# Introduction to Sketching

IAP 2008

Armando Solar-Lezama

# What is sketching?

- A program synthesis system
  - generates small fragments of code
  - checks their validity against a specification

- A programming aid
  - help you write tricky programs
  - cleverness and insight come from you
    - sorry, no programming robots
  - computer helps with low-level reasoning

# The sketching experience

sketch

implementation
(completed sketch)

# Sketch language basics

- Sketches are programs with holes
  - write what you know
  - use holes for the rest

- 2 semantic issues
  - specifications
    - How does SKETCH know what program you actually want?
  - holes
    - Constrain the set of solutions the synthesizer may consider

# Specifications

- Specifications constrain program behavior
  - assertions

```
assert x > y;
```

  - function equivalence

```
blockedMatMul(Mat a, Mat b) implements matMul
```

Is this enough?

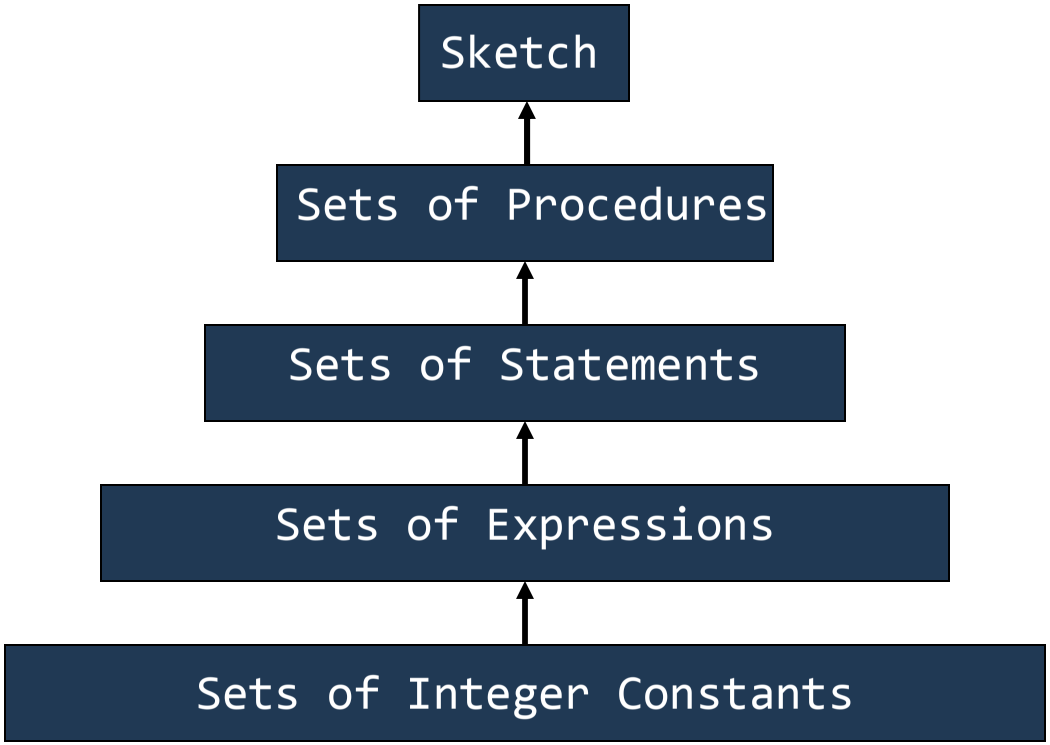
# Holes

- Holes are placeholders for the synthesizer
  - synthesizer replaces hole with concrete code fragment
  - fragment must come from a set defined by the user

**Defining sets of code fragments is the key to Sketching effectively**

# Defining sets of code fragments

- Sets are defined hierarchically
  - a sketch is just a set of possible programs
  - the synthesizer will chose a correct program from the set



# Integer hole

- Define sets of integer constants

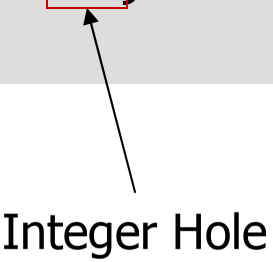
Example: Hello World of Sketching

*spec:*

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

*sketch:*

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



# Integer Holes → Sets of Expressions

- Example: 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 zero to a 1
  - i.e. 000111 + 1 = 001000

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

→

```
!(x + 1) & (x + 0)
```

```
!(x + 1) & (x + 0xFFFF)
```

```
!(x + 0) & (x + 1)
```

```
!(x + 0xFFFF) & (x + 1)
```

# Integer Holes → Sets of Expressions

- Example: 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 → Sets of Expressions

- Least Significant One Bit
  - 0010 0100 → 0000 0100

```
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;  }
}
```

- Will the same trick work?
  - try it out

# Integer Holes → 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

- {| 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 One revisited

- How did I know the solution would take the 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

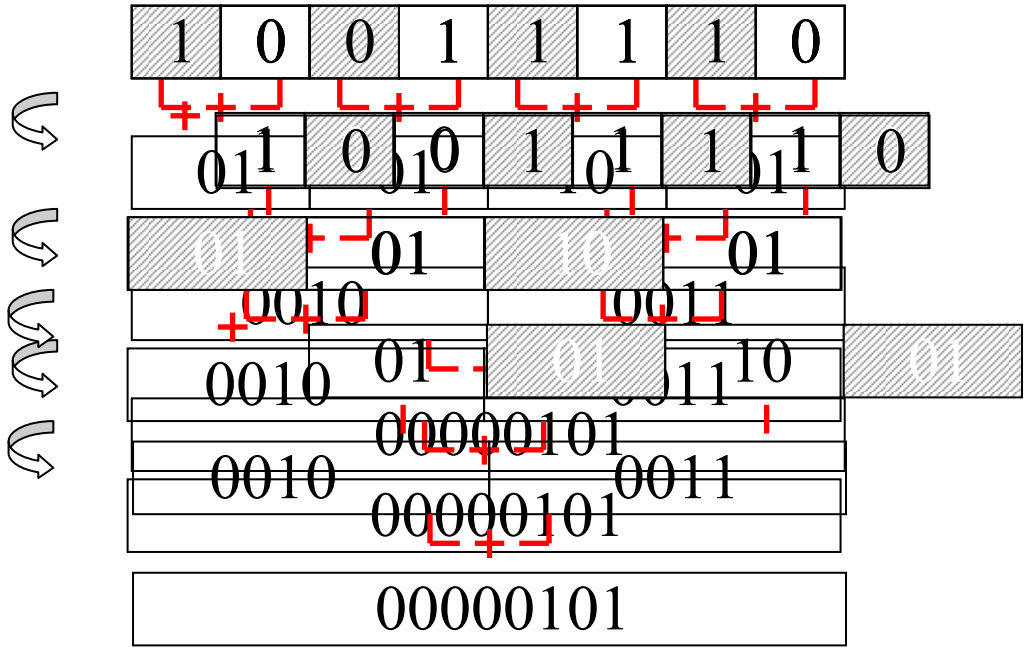
- Avoid copying and pasting
  - `repeat(n){ s}` ➔ $\underbrace{s;s;\ldots s;}_{n}$
  - each of the n copies may resolve to a distinct stmt
  - n can be a hole too.

```
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 --unrollamnt to set the maximum value of n

# Example: logcount

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



# Procedures and Sets of Procedures

- 2 types of procedures
  - standard procedures
    - represents a single procedure
    - all call sites resolve to the same procedure
    - identified by the keyword **`static`**
  - generators
    - represents a set of procedures
    - each call site resolves to a different procedure in the set
    - default in the current implementation

# Example:

```
int rec(int x, int y, int z){
   int t = ??;
   if(t == 0){return x;}
   if(t == 1){return y;}

   if(t == 2){return z;}

   if(t == 3){return rec(x,y,z) * rec(x,y,z);}

   if(t == 4){return rec(x,y,z) + rec(x,y,z);}

   if(t == 5){return rec(x,y,z) - rec(x,y,z);}
}

int sketch( int x, int y, int z ) implements spec{
   return rec(x,y, z);
}
```

```
int spec( int x, int y, int z ){
   return (x + x) * (y - z);
}
```

# Closing Remarks

- Problems will be posted on the website at the end of class.

- Tomorrow:
  - how sketching works
  - advanced use of the synthesizer
  - debugging your sketches