SKETCH TUTORIAL

EXCAPE SEMINAR SEPT 10, 2012

Armando Solar-Lezama

MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY
What is Sketch

A programming language with synthesis capabilities

A platform for synthesis research

• Rapidly try out ideas involving constraint based synthesis
• Avoid reinventing the wheel
SYNTHESIS WITH PARTIAL PROGRAMS
The synthesis problem

Synthesis as a game

For every move of the environment

Synthesized program makes a counter move
The challenge of synthesis

For functions, the environment sets the inputs
- i.e. whatever we synthesize must work for all inputs

Modeled with a doubly quantified constraint

\( \exists P \forall in \ (in, P \models Spec) \)

- What does it mean to quantify over programs?
Quantifying over programs

Synthesis as curve fitting

- we want a function that satisfies some properties

It’s hard to do curve fitting with arbitrary curves

- Instead, people use parameterized families of curves
- Quantify over parameters instead of over functions

\[ \exists c \forall in \ (in, P[c] \models Spec) \]

Key idea:
Let user define parameterized functions with partial programs
DESIGNING A LANGUAGE FOR PARTIAL PROGRAMS
Language Design Strategy

Extend base language with **one** construct

**Constant hole:** ??

```c
int bar (int x)
{
    int t = x * ??;
    assert t == x + x;
    return t;
}
```

Synthesizer replaces ?? with a constant

High-level constructs defined in terms of ??
Expressions with ?? == sets of expressions

- linear expressions: \( x^{??} + y^{??} \)
- polynomials: \( x^{x^{??}} + x^{??} + ?? \)
- sets of variables: \( ?? \ ? x : y \)
Example: Registerless Swap

Swap two words without an extra temporary

```c
int W = 32;

void swap(ref bit[W] x, ref bit[W] y){
    if(??){ x = x ^ y;}else{ y = x ^ y; }
    if(??){ x = x ^ y;}else{ y = x ^ y; }
    if(??){ x = x ^ y;}else{ y = x ^ y; }
}

harness void main(bit[W] x, bit[W] y){
    bit[W] tx = x; bit[W] ty = y;
    swap(x, y);
    assert x==ty && y == tx;
}
```
From simple to complex holes

We need to compose ?? to form complex holes

Borrow ideas from generative programming
- Define generators to produce families of functions
- Use partial evaluation aggressively
Generators

Look like a function

- but are partially evaluated into their calling context

Key feature:

- Different invocations $\rightarrow$ Different code
- Can recursively define arbitrary families of programs
Example: Least Significant Zero Bit

- 0010 0101 → 0000 0010

```c
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
Sample Generator

/**
 * Generate the set of all bit-vector expressions
 * involving +, &, xor and bitwise negation (~).
 * the bnd param limits the size of the generated expression.
 */

generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return ~gen(x, bnd-1);
    if(??){
        return { | gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) |};
    }
}

Example: Least Significant Zero Bit

generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return ~gen(x, bnd-1);
    if(??){
        return { | gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) |};
    }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
    return gen(x, 3);
}
High order generators

```c
/*
 * Generate code from f n times
 */
generator void rep(int n, fun f) {
    if (n > 0) {
        f();
        rep(n - 1, f);
    }
}
```
Example: Reversing bits

```c
bit[W] reverse (bit[W] in) {
    bit [W] out=0;
    for (int i = 0; i < W; i++) {
        out[i] = in[W-1 - i];
    }
    return out;
}
```
Closures + High Order Generators

generator void rep(int n, fun f) {
    if (n > 0) {
        f();
        rep(n - 1, f);
    }
}

bit[16] reverseSketch(bit[16] in) implements reverse {
    bit[16] t = in;
    int s = 1;
    generator void tmp() {
        bit[16] m = ??;
        t = ((t << s) & m) | ((t >> s) & (~m));
        s = s * ??;
    }
    rep(??, tmp);
    return t;
}
Syntactic Sugar

{{| 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
repeat

Avoid copying and pasting

- \( \text{repeat}(n) \{ \ s \} \rightarrow s;s;...s; \)

- each of the \( n \) copies may resolve to a distinct stmt
- \( n \) can be a hole too.
**Example: Reversing bits**

```plaintext
pragma options "--bnd-cbits 3 ";

int W = 32;

bit[W] reverseSketch(bit[W] in) implements reverse {

    bit[W] t = in;
    int s = 1;
    int r = ??;
    int r = ??;

    repeat(??){

        bit[W] tmp1 = (t << s);
        bit[W] tmp2 = (t >> s);
        t = tmp1 || tmp2;
        // Syntactic sugar for m=??, (tmp1&m | tmp2&~m).
        s = s*r;
    }

    return t;

}
```
KARATSUBA MULTIPLICATION
Karatsuba Multiplication

Recursive grade-school multiplication

\[
x = x_2 \cdot p^{2^{\frac{n}{2}}} + x_1
\]
\[
y = y_2 \cdot p^{2^{\frac{n}{2}}} + y_1
\]
\[
a = x_1 \cdot y_1
\]
\[
b = x_2 \cdot y_2
\]
\[
c = x_1 \cdot y_2
\]
\[
d = x_2 \cdot y_1
\]

\[
x \cdot y = a + b \cdot p^{2^{\frac{n}{2}}} + c \cdot p^{2^{\frac{n}{2}}} + d \cdot p^n
\]
Karatsuba Multiplication

Smarter Karatsuba Multiplication

\[
x = x_1 \cdot p^{\frac{n}{2}} + x_2
\]

\[
y = y_1 \cdot p^{\frac{n}{2}} + y_2
\]

\[
a = x_1 \cdot y_1
\]

\[
b = x_2 \cdot y_2
\]

\[
c = \ldots
\]

\[x \cdot y = \ldots\]

Key requirement:
- Only one multiplication

Compute \(x \cdot y\) using only sums and differences of \(a, b, c\).
(Multiplication times \(p^k\) are ok)
Karatsuba Multiplication

```c
int [2*N] karatsuba(int N, int[N] x, int[N] y) implements mult {
    if(N % 2 != 0) { return mult(N, x, y); }
    int No2 = N/2;
    int[No2] x1, x2, y1, y2;
    int[N] a=0, b=0, c=0;
    int[2*N] out = 0;
    x1=x[0::No2];  x2=x[No2::No2];
    y1=y[0::No2];  y2=y[No2::No2];
    a = karatsuba(No2, x1, y1);
    b = karatsuba(No2, x2, y2);
    c = karatsuba(No2, poly1(No2, x1,x2,y1,y2),
                  poly1(No2, x1,x2,y1,y2));
    repeat(??){
        int[N] t = { | a | b | c |};
        out = plus(2*N, out,
                   shiftVect(2*N, { | t | minus(N, t) |} , { |N | No2 | 0|} )  )
    }
    out = normalize(2*N, out);
    return out;
}
```
UNINTERPRETED FUNCTIONS
Uninterpreted functions

Uninterpreted functions are just another input
- Sketch must be correct for all functions $f$

Useful for modeling complex functionality
- also useful to model the environment
File Reading/Writing with UFuns

```c
int NDCNT=0;
int getND_private(int i);
int getND()
{
    return getND_private(NDCNT++);
}
```

**Non Deterministic Value Producer**

Every time this function is called it produces a new non-deterministic value.
struct FileHandle{
    int maxReads;
}

FileHandle getFile(){
    return new FileHandle(maxReads=getND());
}

bit moreValues(FileHandle fh){
    assert fh.maxReads >= 0;
    return fh.maxReads!=0;
}

int readInt(FileHandle fh){
    assert fh.maxReads > 0;
    --fh.maxReads;
    return getND();
}
**Reading a file**

Synthesizer can use file model to synthesize code that reads a file

```java
harness void foo(){
    FileHandle fh = getFile();
    while (true | false | moreValues(fh) |) {
        int x = readInt(fh);
        printInt(x);
    }
    assert !moreValues(fh);
}
```
Sketch annotation system

Like Java annotations.
Make it easy to experiment with language extensions

Ex: Native annotation

• Override the standard code generator

```java
@Native(
"{ FileHandle* f = new FileHandle("input.in"); _out = f; }"
) FileHandle getFile(){
    return new FileHandle(maxReads=getND());
}
```
SKETCH AS A TOOL FOR SYNTHESIS RESEARCH
Sketch as a bridge
Sketch as a backend

High-level notation for synthesis problems
Much easier than translating to SMT

Ex:
- Storyboard Programming
- Synthesis of SQL Queries
- Autograder
Example from Autograder

DEMO
THE SKETCH SYNTHESIS
INFRASTRUCTURE
Synthesizer Structure

Sketch Program → Parsing and Lowering → Sketch IR

Sketch IR → Completion and partial evaluation → CEGIS Solver

CEGIS Solver → Resolved Sketch

Resolved Sketch → Quantified Constraints

Partially desugared IR
Installation: The easy way

Get the Pre Built Tar Ball
- Contains pre-built JAR file for the frontend
- Source distribution for the backend
- Versions available with pre-built backends as well

Build the backend if necessary

Run!
Installation: The hard way

Do this if you plan to develop Sketch

Requires many tools to be installed
  - Maven, Mercurial, Python, ...

Follow instructions in bitbucket site
Environment Variables

Set PATH if you want to be able to run from other directories

Set SKETCH_HOME to run compiled code
  - From sketch-frontend run export
    SKETCH_HOME="`pwd`/runtime"
Don’t reinvent the wheel
BACKUP
IN PLACE LIST REVERSAL
Problem statement

Given a list like this:

Produce a list like this:
Constraints

Your algorithm must be $O(n)$

Your algorithm must use a constant space

- It cannot use arrays or recursion
harness void main(int n){
    if(n >= MAXN){ n = MAXN-1; }
    node[n] nodes = null;
    list l = newList();

    popList(n, l, nodes);

    reverseSK(l);

    check(n, l, nodes);
}
The Spec

```csharp
void popList(int n, list l, ref node[n] nodes) {
    node tail = null;
    for (int i = 0; i < n; ++i) {
        node t = newNode();
        if (i > 0) {
            tail.next = t;
            tail = t;
        } else {
            l.head = t;
        }
        nodes[i] = t;
    }
}
```
**The Spec**

```c
void check(int n, list l, node[n] nodes)
{
    node cur = l.head;
    int i=0;
    while(cur != null)
    {
        assert cur == nodes[n-1-i];
        cur = cur.next;
        i = i+1;
    }
    assert i == n;
    if(n > 0){
        assert l.head == nodes[n-1];
    }else{
        assert l.head == null;
    }
}
```
The Sketch

```c
void reverseSK(ref list l){
    node tmp1 = null; node tmp2 = null;
    // some assignments go here
    while(  ){
        // some more go in here.
    }
}

{| (tmp1 | tmp2 | l.head)().next)? |}
```
A CONSTRAINT BASED SYNTHESIS PRIMER
Framing the synthesis problem

Goal: Find a function from holes to values
  - Easy in the absence of generators

\[
\text{bit}[W] \ \text{isolateSk} \ (\text{bit}[W] \ x) \ \text{implements} \ \text{isolate0} \ {\ \\
\text{\hspace{1cm} return} \ ! (x + \phi \{?_1\}) (x + \phi \{?_2\}) ;
\}
\]

- Finite set of holes so function is just a table
Framing the synthesis problem

Generators need something more

generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??1) return x;
    if(??2) return ??5;
    if(??3) return gen(x, bnd-1);
    if(??4){
        ...
    }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
    return gen0(x, 3);
}
Framing the synthesis problem

Generators need something more

- The value of the holes depends on the context

```c
#include "cmdl.h"

generator bit[W] gen(context τ, bit[W] x, int bnd) {
   assert bnd > 0;
   if(φ(τ, ??_1)) return x;
   if(φ(τ, ??_2)) return φ(τ, ??_5);
   if(φ(τ, ??_3)) return ~gen_q1(τ ∙ g_1, x, bnd-1);
   if(φ(τ, ??_4)){
      ...
   }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
   return gen_q0(g_0, x, 3);
}
```
Framing the synthesis problem

```
generator bit[W] gen(context τ, bit[W] x, int bnd) {
    assert bnd > 0;
    if(φ(τ, ??₁)) return x;
    if(φ(τ, ??₂)) return φ(τ, ??₃);
    if(φ(τ, ??₄)) return ~gen₉₁(τ · g₁, x, bnd-1);
    if(φ(τ, ??₅)) {
        return { | gen₉₂(τ · g₂, x, bnd-1) (+| &| ^) gen₉₃(τ · g₃, x, bnd-1) |};
    }
}

bit[W] isolate0sk (bit[W] x) implements isolate0 {
    return gen₉₀(g₀, x, 3);
}
```

- Potentially unbounded set of unknowns
- We can bound the depth of recursion
  - That means again φ is just a table

φ(g₀, ??₉₀)
φ(g₀g₁, ??₉₀)
φ(g₀g₂, ??₉₀)
φ(g₀g₁g₂, ??₉₀)
φ(g₀g₁g₃, ??₉₀)
φ(g₀g₁g₂g₁, ??₉₀)
...
Solving the synthesis problem

Many ways to represent \((\text{in}, P[c] = \text{Spec})\)

- Important area of research
- At the abstract level, it’s just a predicate
Many different options

a) Eliminate $\forall$ symbolically
   - You can use Farkas Lemma
   - You can use an abstract domain
   - You can use plain-vanilla elimination (not recommended)

b) Sample the space of inputs intelligently
Hypothesis

Sketches are not arbitrary constraint systems
  - They express the high level structure of a program

A small number of inputs can be enough
  - focus on corner cases

\[ \exists c \ \forall \text{in} \in E \ Q(\text{in}, c) \]
  where \( E = \{x_1, x_2, \ldots, x_k\} \)

This is an inductive synthesis problem!
Ex: Sketch

\[
\exists c \text{ s.t. } \text{Correct}(P_c, \text{in}_i)
\]

Synthesize

\{\text{in}_i\}

Check

Insert your favorite checker here
CEGIS in Detail

$\exists P \forall in \ (in, P \in \text{Spec})$

**Synthesize**

$Q(c, in_0)$

$Q(c, in_2) \land Q(c, in_3)$

**Check**

$\exists i\forall Q(i(c(in_2))c)$
Example

You want to partition $N$ elements over $P$ procs

- How many elements should a processor get?

Obvious answer is $N/P$

Obvious answer is wrong!
Synthesizing a partition function

What do we know?
- The interface to the function we want
- Not all processors will get the same # of elements
- The kind of expressions we expect

```c
void partition(int p, int P, int N, ref int ibeg, ref int iend) {
    N P N\%P
    P * + N/P
}
```
Synthesizing a partition function

How does the system know what a partition is?

```c
harness void testPartition(int p, int N, int P) {
  
  if (p >= P || P < 1) { return; }
  int ibeg, iend;
  partition(p, P, N, ibeg, iend);
  assert iend - ibeg < (N/P) + 2;
  if (p+1 < P) {
    int ibeg2, iend2;
    partition(p+1, P, N, ibeg2, iend2);
    assert iend == ibeg2;
  }
  if (p==0) { assert ibeg == 0; }
  if (p==P-1) { assert iend == N; }
  
}
```

- Partitions should be balanced
- Adjacent partitions should match
- First and last partition should go all the way to the ends