Automatic Programming Revisited

Part I: Puzzles and Oracles

Rastislav Bodik

University of California, Berkeley

Once you understand how to write a program, get someone else to write it. *Alan Perlis, Epigram #27*

The Exascale Programming Challenge

The Exascale Programming Challenge

More levels of hierarchy

Accelerators everywhere

The revenge of Ahmdal’s Law

Programmers will be swamped in design choices

The Exascale Programming *Opportunity*

How can CPU cycles help in programming?

The **SKETCH** Language

try it at `bit.ly/sketch-language`

SKETCH: just two constructs

spec:

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

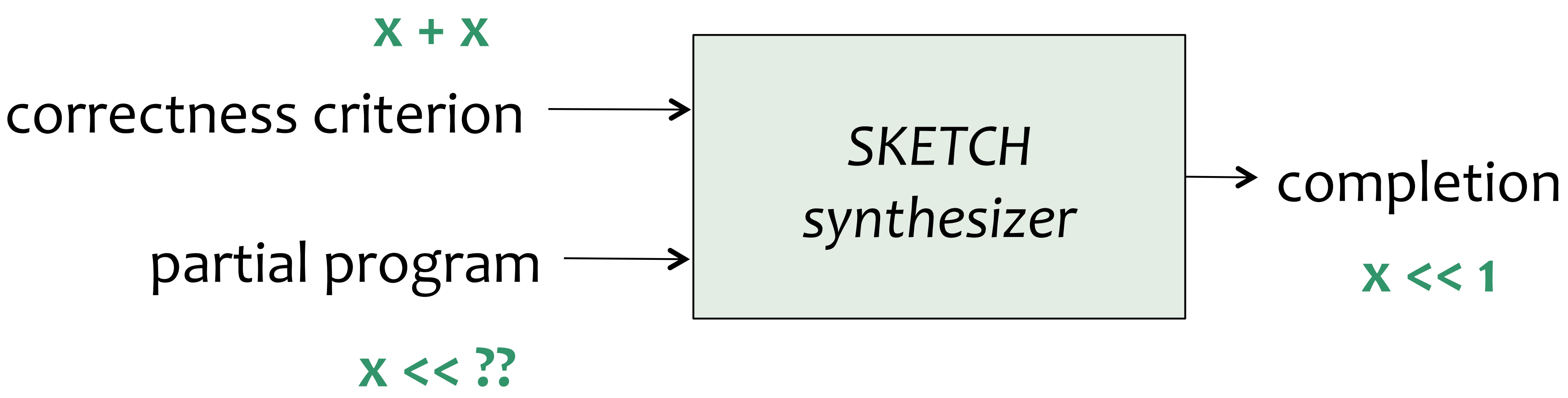
sketch:

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

result:

```
int bar (int x) implements foo {
    return x << 1;
}
```

SKETCH is synthesis from partial programs



No need for a domain theory. No rules needed to rewrite
x+x into **2*****x*** into ***x<<1***

Demo 1: division of a polynomial

```
int spec (int x) {
    return x*x*x-19*x+30;
}

#define Root {| ?? | -?? |}

int sketch (int x) implements spec {
    return (x - Root) * (x - Root) * (x - Root);
}
```

Note: Sketch divides polynomials slowly but it knows nothing about finding roots of polynomials. This generality enables it to do synthesis of arbitrary programs.

Example: Silver Medal in a SKETCH contest

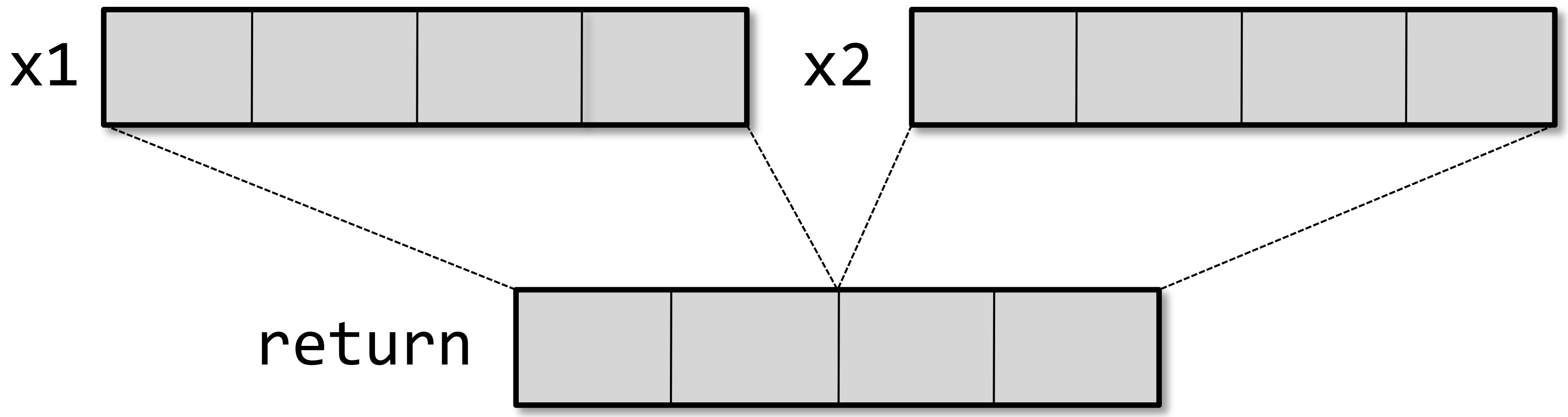
4x4-matrix transpose, the specification:

```
int[16] trans(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

Implementation idea: parallelize with SIMD

Intel shufps SIMD instruction

SHUFP (shuffle parallel scalars) instruction



The SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
int[16] trans_sse(int[16] M) implements trans { // synthesized code
  S[4::4]   = shufps(M[6::4],   M[2::4],  11001000b);
  S[0::4]   = shufps(M[11::4],  M[6::4],  10010110b);
  S[12::4]  = shufps(M[0::4],   M[2::4],  10001101b);
  S[8::4]   = shufps(M[8::4],   M[12::4], 11010111b);
  T[4::4]   = shufps(S[11::4],  S[1::4],  10111100b);
  T[12::4]  = shufps(S[3
  T[8::4]   = shufps(S[4
  T[0::4]   = shufps(S[1
}
```

From the contestant email:

Over the summer, I spent about 1/2 a day manually figuring it out.

Synthesis time: 30 minutes.

Demo 2: 4x4 matrix transpose

```
pragma options "--bnd-unroll-amnt 6 --bnd-inbits 3 --bnd-cbits 6";

int[16] transpose(int[16] mx){
            int x, y;
            for(x = 0; x < 4; x++)
               for(y = 0; y <= x; y++)
                  mx[4*x+y] = mx[4*y+x];
            return mx;
}

generator int[4] shufps(int[4] xmm1, int[4] xmm2, bit[8] imm8){ /* automatically rewritten */
            int[4] ret;
            ret[0] = xmm1[(int)imm8[0::2]];
            ret[1] = xmm1[(int)imm8[2::2]];
            ret[2] = xmm2[(int)imm8[4::2]];
            ret[3] = xmm2[(int)imm8[6::2]];
            return ret;
}

int[16] sse_transpose(int[16] mx) implements transpose {
            int[16] p0 = 0;
            int[16] p1 = 0;
            // Find the extra insight (constraint) that this version communicates to the synthesizer.
            int steps = ??;
            loop(steps){ p0[??::4] = shufps(mx[??::4], mx[??::4], ??); }
            loop(steps){ p1[??::4] = shufps(p0[??::4], p0[??::4], ??); }
            return p1;
}
```

How can synthesis help?

In this example, our programmer possessed enough knowledge to actually write the program himself.

The <u>synthesizer saved him from tedious details</u>, like a compiler.

Note we did not have to teach that compiler any SIMD optimizations, as is usually necessary.

In the next example, the synthesizer will help us <u>find</u> the program (actually, a solution to a puzzle). We could not solve the problem without the synthesizer.

The Hat Game

There are ***n*** players in a room. Someone will soon come by and put hats labeled ***0*** to ***n-1*** on each of their heads. There may be multiple hats with the same number.

Once the hats are in place, the players cannot communicate. Each player must then guess which hat is on their head. A player can see everyone else's hat, but not their own.

The challenge is for the group to come up with a strategy such that ***at least one person correctly guesses their own hat***.

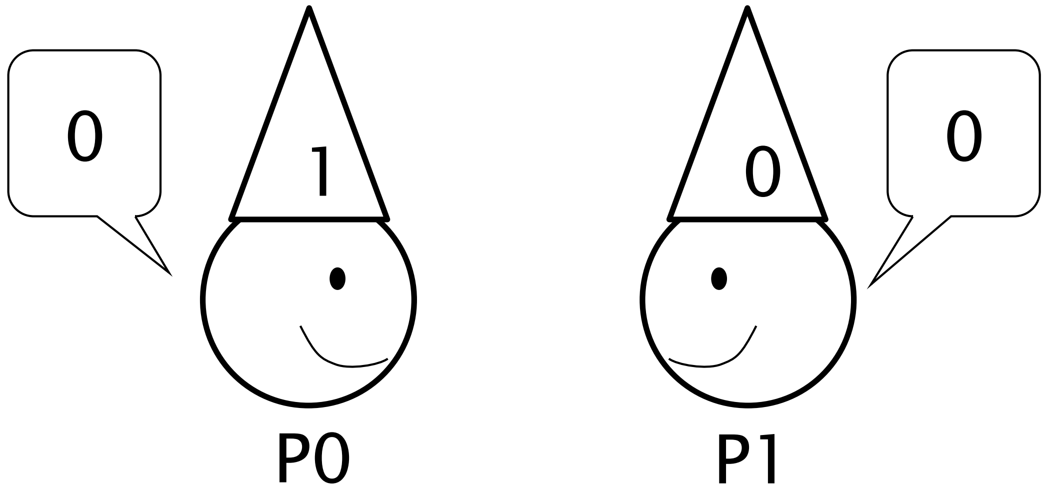
Assume the group knows ***n*** before they strategize.

Finding a winning strategy for n=2

There are only 16 strategies to consider.

We can find a winning one manually.

Color of hat the player can see	What player P0 will guess	What player P1 will guess
0	0	1
1	1	0



Finding a winning strategy for n=3

There are now 7,625,597,484,987 possible strategies.

We gave up on finding a winning one manually.

Colors of hats the player sees	What player P0 will guess	What player P1 will guess	What player P2 will guess
0,0			
0,1			
0,2			
1,0			
1,1			
1,2			
2,0			
2,1			
2,2			

The synthesis correctness condition (n=3)

```
p0_strategy(p1_hat, p2_hat) : int {
   p0 : int[3][3] = { ??(0,1,2), ??(0,1,2) … }
   return p0[p1_hat][p2_hat];
}


…


forall (i, j, k) from i, j, k  in [0,2]
  assert i = p0_strategy(j, k)
      or j = p1_strategy(i, k)
      or k = p2_strategy(i, j)
```

Computing a winning strategy for n=3

We asked an oracle to compute a winning strategy. There are 10,752 of them.

Colors of hats the player sees	What player P0 will guess	What player P1 will guess	What player P2 will guess
0,0	0	1	2
0,1	1	0	1
0,2	2	2	0
1,0	1	2	0
1,1	2	1	2
1,2	0	0	1
2,0	2	1	0
2,1	0	0	2
2,2	1	2	1

The Hat Game, Revisited

Now assume that the players do **not** know the total number of players, ***n***, or their own id, ***k***, until the hats are placed.

Their winning strategy thus must be a function *f(k, n, hats)*.

Our goal is to devise such a function *f*. This is our “program”.

We (humans) will observe the (oracle’s) winning strategies for ***n***=3 and generalize them for arbitrary *n*.

Generalizing from *n*=3 to arbitrary *n*.

Here is one of the 10,752 winning strategies.

Sadly, the algorithmic pattern is not visible.

Colors of hats the player sees	**What player P0 will guess**	**What player P1 will guess**	**What player P2 will guess**
0,0	0	1	2
0,1	1	0	1
0,2	2	2	0
1,0	1	2	0
1,1	2	1	2
1,2	0	0	1
2,0	2	1	0
2,1	0	0	2
2,2	1	2	1

Idea 1: **Interact** with the oracle

Fix a strategy for P0 and ask what P1 and P2 strategies yield a winning group strategy. There are 8 of them.

Colors of hats the player sees	**What player P0 will guess**	**What player P1 will guess**	**What player P2 will guess**
0,0	0	1	2
0,1	1	0	1
0,2	2	2	0
1,0	1	2	0
1,1	2	1	2
1,2	0	0	1
2,0	2	1	0
2,1	0	0	2
2,2	1	2	1

Coordination between oracles

One oracle's choice can affect the value of another

- Some coordination is required
- In hat game, coordination occurred within a substrategy

Independence is the lack of coordination

- In hat game, independence existed across substrategies

The oracles can exhibit unintended coordination

Idea 2: **Mine** oracle's alternative solutions

It turns out that a winning strategy can be composed from any combination of smaller strategies.

Colors of hats the player sees	**What player P0 will guess**	**What player P1 will guess**	**What player P2 will guess**
0,0	0	1	2
0,1	1	0	1
0,2	2	2	0
1,0	1	2	0
1,1	2	1	2
1,2	0	0	1
2,0	2	1	0
2,1	0	0	2
2,2	1	2	1

Idea 3: Ask the system to **synthesize** *f*

We tell the system “synthesize *f* that uses +,- and % ”

```
    f(k,n,hats) = “a program with +,-,%,sum”
```

and the system produces the function

```
    f(k,n,hats) = (k - 1 - sum(hats)) % n
```

which is a winning strategy parametric in *k*, *n*.

Summary

Ask oracle to compute all strategies (programs) for n=3

Interact with the oracle by constraining it and observing what solutions remain.

Decompose the solutions to see if a strategy can be composed from smaller strategies.

Synthesize the function that is the parametric strategy.

Beyond synthesis of constants

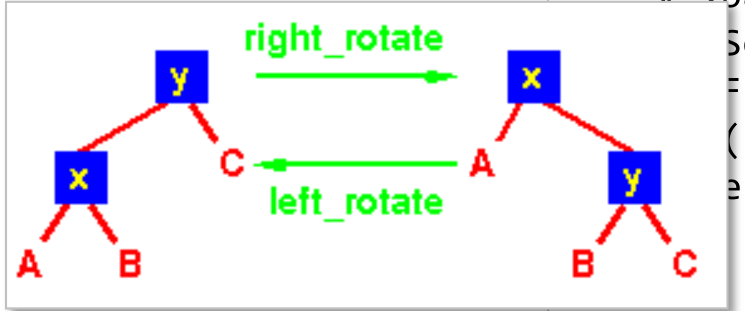
Sometimes the insight is "*I want to complete the hole with an of particular syntactic form.*"

- Array index expressions: `A[ ??*i+??*j+?? ]`
- Polynomial of degree 2: `??*x*x + ??*x + ??`
- Initialize a lookup table: `int strategy[N] = {??,??,??,??}`

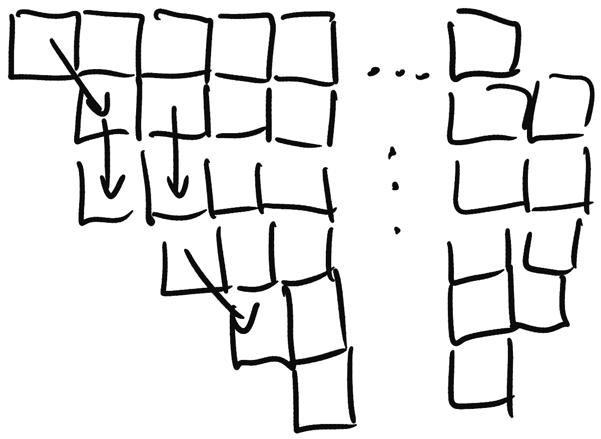
Angelic Programming

What's your memory of Red-Black Tree?

```
left_rotate( Tree T, node x ) {
    node y;
    y = x->right;
    /* Turn y's left sub-tree into x's right sub-tree */
    x->right = y->left;
    if ( y->left != NULL )
        y->left->parent = x;
    /* y's new parent was x's parent */
    y->parent = x->parent;
       Set the parent to point to y instead of x */
       First see whether we're at the root */
       ( x->parent == NULL ) T->root = y;
       e
        if ( x == (x->parent)->left )
            /* x was on the left of its parent */
            x->parent->left = y;
        else
            /* x must have been on the right */
            x->parent->right = y;
    /* Finally, put x on y's left */
    y->left = x;
    x->parent = y;
    }
```



Jim Demmel's napkin



Programmers often think with examples

They often design algorithms by devising and studying examples demonstrating steps of algorithm at hand.

If only the programmer could ask for a demonstration of the desired algorithm!

The demonstration (a trace) reveals the insight.

We create demonstration with an executable oracle.

Angelic choice

Angelic nondeterminism.

Oracle makes an angelic (clairvoyant) choice.

`!!(S)` evaluates to a value chosen from set S such that the execution terminates without violating an assertion

Angelic Programming

Angelic construct: **choose(val1, val2, …)**

- The oracle returns one of the parameters
- Choice is carefully made so that all assertions pass
- In the hat game, the oracle filled in the table

```
p0 : int[3][3] = {choose(0,1,2), choose(0,1,2) … }
p1 : int[3][3] = {choose(0,1,2), choose(0,1,2) … }
p2 : int[3][3] = {choose(0,1,2), choose(0,1,2) … }

forall (i, j, k) from i in [0,2], j in [0,2], k in [0,2]
  assert i = p0[j][k] or j = p1[i][k] or k = p2[i][j]
```

Programming with oracles (DFS)

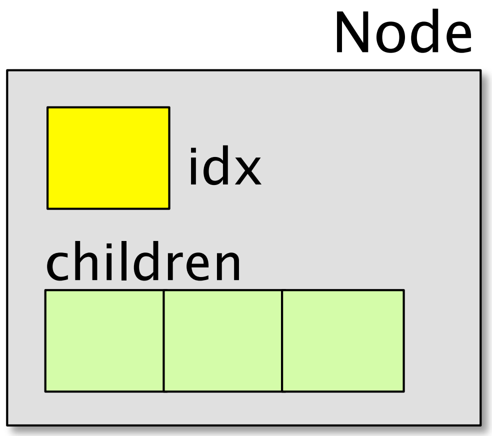
Design DFS traversal that does not use a stack.

Used in garbage collection: when out of memory, you cannot ask for O(N) memory to mark reachable nodes

We want DFS that uses O(1) memory.

Depth-first search with explicit stack

```
vroot = new Node(g.root)
push(vroot); current = g.root

while (current != vroot) {
   if (!current.visited) current.visited = true
   if (current has unvisited children) {
         current.idx := index of first unvisited child
         child = current.children[current.idx]
         push(current)
         current = child
   } else {
         current = pop()
   }
```



Parasitic Stack

Borrows storage from its host (the graph)

accesses the host graph via pointers present in traversal code

A two-part interface:

stack: usual push and pop semantics

parasitic channel: for borrowing/returning storage

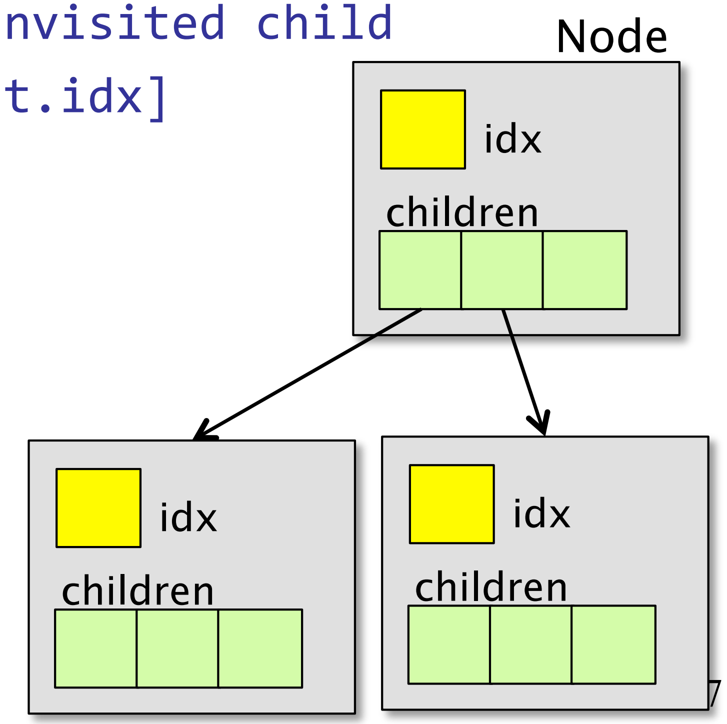
```
push(x,(node1,node2,…))      stack can (try to) borrow fields in node_i
pop(node1,node2,…)       value node_i may be handy in returning storage
```

Parasitic stack expresses an optimization idea

But can DFS be modularized this way? Angels will tell us.

Replace regular stack with parasitic stack

```
vroot = new Node(root)
push(null); current = vroot

while (current != vroot) {
   if (!current.visited) current.visited = true
   if (current has unvisited children) {
         current.idx := index of first unvisited child
         child = current.children[current.idx]
         push(current, (current, child))
         current = child
   } else {
         current = pop((current))
   }
```



Angels perform deep global reasoning

Which location to borrow?

  traversal must not need until it is returned

How to restore the value in the borrowed location?

  the stack does not have enough locations to remember it

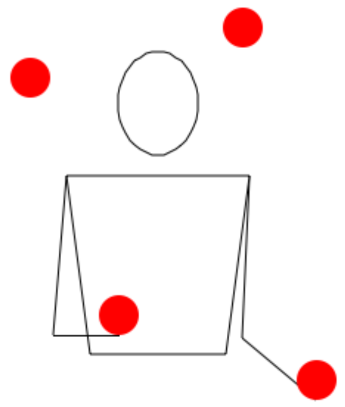
How to use the borrowed location?

  it must implement a stack

Angels will clairvoyantly made these decisions for us

- in principle, human could set up this parasitic “wiring”, too, but we failed without the help of the angels

ParasiticStack.push



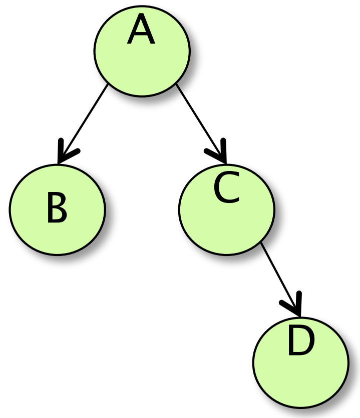
```
class ParasiticStack {
    var e // allow ourselves one extra storage locatic

    push(x, nodes) {
        // borrow memory location n.children[c]
        n = choose(nodes)
        c = choose(0 until n.children.length)

        // value in the borrowed location; will need to be restored
        v = n.children[c]

        // we are holding 4 values but have only 2 memory locations
        // select which 2 values to remember, and where
        e, n.children[c] = angelicallyPermute(x, n, v, e)
    }
```

ParasiticStack.pop

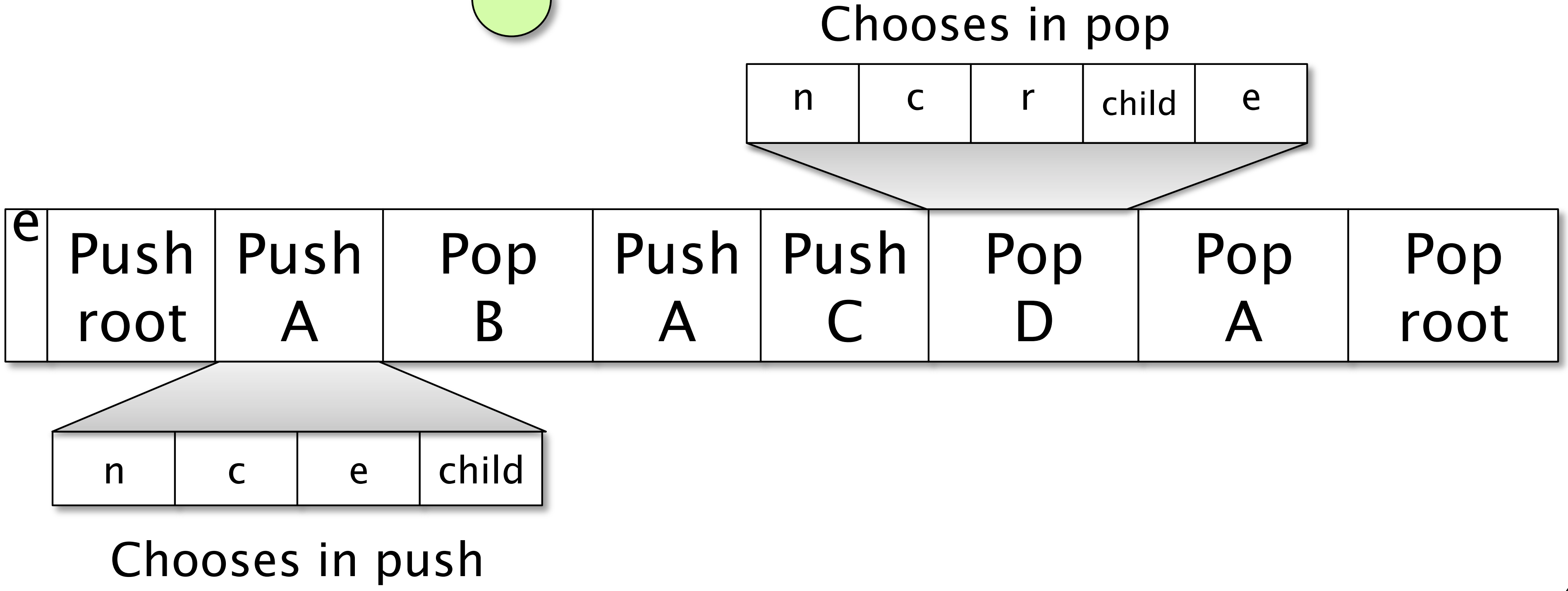
```
pop(values) {
   // ask the angel which location we borrowed at time of push
   n = choose(e, values)
   c = choose(0 until n.children.length)

   // v is the value stored in the borrowed location
   v = n.children[c]

   // (1) select return value
   // (2) restore value in the borrowed location
   // (3) update the extra location e
   r, n.children[c], e = angelicallyPermute(n,v,e,values)

   return r
}
```

Running the angelic program

Input:



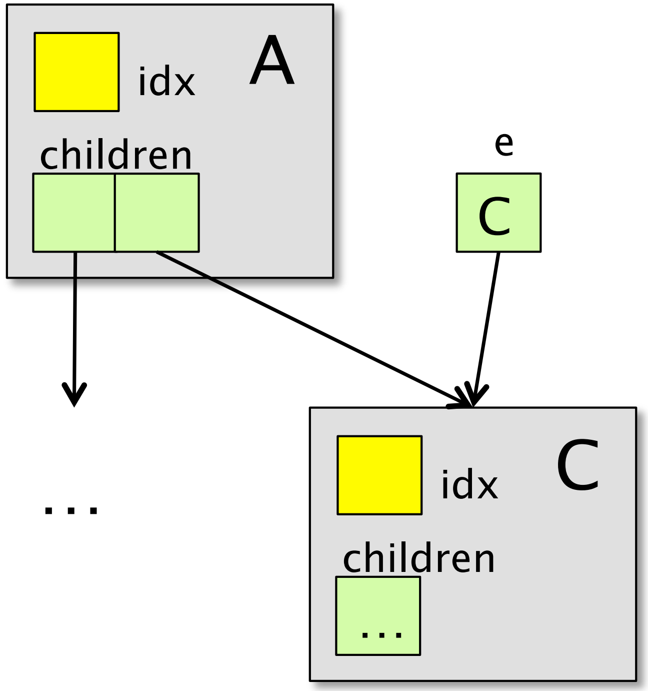
8040 solutions synthesized



Example of an undesirable trace

Undesirable traces meet the spec but do not demonstrate a desirable algorithm

```
// choose initial value for extra storage
e = choose(nodes)
:
push(..)
:
n.children[c] = angelPermute(x,n,v,e) // e
```



Good and bad traces

Trace with strange behavior

```
// choose initial value for extra storage
e = Node(c)

// call to push(x = Node(origin), nodes = {Nod
// choose node whose child pointer will be borrowed
n = choose(nodes) // Node(a)
// choose which child pointer will be borrowed
c = choose(0 until n.children.length) // 1
// choose how to set storage nodes
// original values: Node(c), Node(c)
e, n.children[c] = angelicallyPermute(x, n, v, e)
    // x = Node(origin), e = Node(c)
```

e is chosen only because e = Node(c) in the initializer

Removing coordination

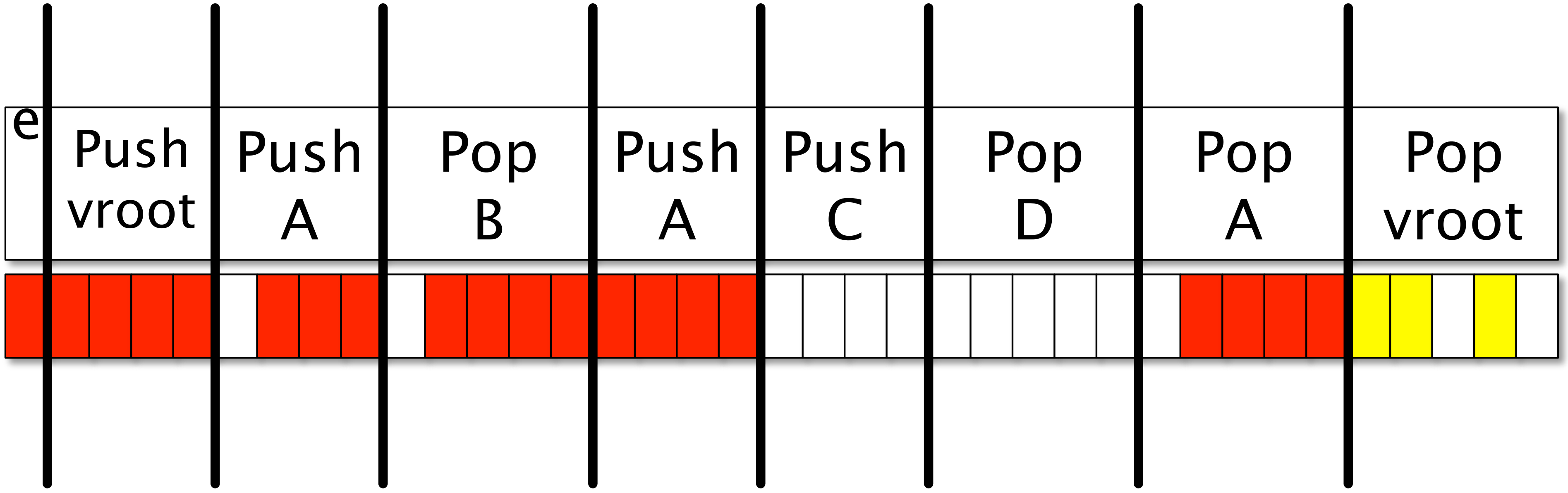
Bad traces usually display unintended coordination

- In DFS, oracles coordinated across procedure boundaries to produce bad traces

We can use the programmer's understanding of coordination to filter out undesired traces

Without coordination, we can break traces into independent subtraces

Interactions in DFS



Each box represents one oracle

- All red oracles are coordinating with each other
- All yellow oracles are coordinating with each other
- All white oracles are completely independent

Let's refine the angelic program

```
class ParasiticStack {
   var e : Node
   push(x, nodes) {
        n = choose(nodes)
        c = choose(0 until n.children.length)
        e, n.children[c] = angelicallyPermute(x,n,v,e)
   }
   pop(values) {
        n = choose(e, values)
        c = choose(0 until n.children.length)
        v = n.children[c]
        r, n.children[c],e = angelicallyPermute(n,v,e,values)
        return r
}  }
```

First we observe what these angels do

```
class ParasiticStack {
   var e : Node
   push(x, nodes) {
        n = choose(nodes)
        c = choose(0 until n.children.length)
        e, n.children[c] = angelicallyPermute(x,n,v,e)
   }
   pop(values) {
        n = choose(e, values)
        c = choose(0 until n.children.length)
        v = n.children[c]
        r, n.children[c],e = angelicallyPermute(n,v,e,values)
        return r
} }
```

Refinement #1

```
class ParasiticStack {
  var e : Node
  push(x, nodes) {
      n = choose(nodes)
      c = choose(0 until n.children.length)
      e, n.children[c] = x, e
  }
  pop(values) {
      n = e
      c = choose(0 until n.children.length)
      v = n.children[c]
      r, n.children[c],e = e, values[0], v
      return r
}  }
```

Refinement #1

```
class ParasiticStack {
   var e : Node
   push(x, nodes) {
        n = choose(nodes)
        c = choose(0 until n.children.length)
        e, n.children[c] = x, e
   }
   pop(values) {
        n = e
        c = choose(0 until n.children.length)
        v = n.children[c]
        r, n.children[c],e = e, values[0], v
        return r
}  }
```

Refinement #2

```
class ParasiticStack {
   var e : Node
   push(x, nodes) {
        n = nodes[0]
        c = choose(0 until n.children.length)
        e, n.children[c] = x, e
   }
   pop(values) {
        n = e
        c = choose(0 until n.children.length)
        v = n.children[c]
        r, n.children[c],e = e, values[0], v
        return r
}  }
```

Refinement #2

```
class ParasiticStack {
   var e : Node
   push(x, nodes) {
        n = nodes[0]
        c = choose(0 until n.children.length)
        e, n.children[c] = x, e
   }
   pop(values) {
        n = e
        c = choose(0 until n.children.length)
        v = n.children[c]
        r, n.children[c],e = e, values[0], v
        return r
}  }
```

Refinement #2

```
class ParasiticStack {
   var e : Node
   push(x, nodes) {          invariant: c == n.idx
        n = nodes[0]
        c = choose(0 until n.children.length)
        e, n.children[c] = x, e
   }
   pop(values) {
        n = e
        c = choose(0 until n.children.length)
        v = n.children[c]
        r, n.children[c],e = e, values[0], v
        return r
}  }
```

Final refinement

```
class ParasiticStack {
   var e : Node
   push(x, nodes) {
        n = nodes[0]

        e, n.children[n.idx] = x, e
   }
   pop(values) {
        n = e

        v = n.children[n.idx]
        r, n.children[n.idx],e = e, values[0], v
        return r
}  }
```

Our results: what we synthesized

Concurrent Data Structures [PLDI 2008]

lock free lists and barriers

Stencils [PLDI 2007]

highly optimized matrix codes

Dynamic Programming Algorithms [OOPSLA 2011]

O(N) algorithms, including parallel ones

To be continued after lunch

How to implement the oracles (synthesis algorithms)

Hiding sketches from programmers

Similar synthesizers and the space of synthesis ideas