Program Synthesis by Sketching

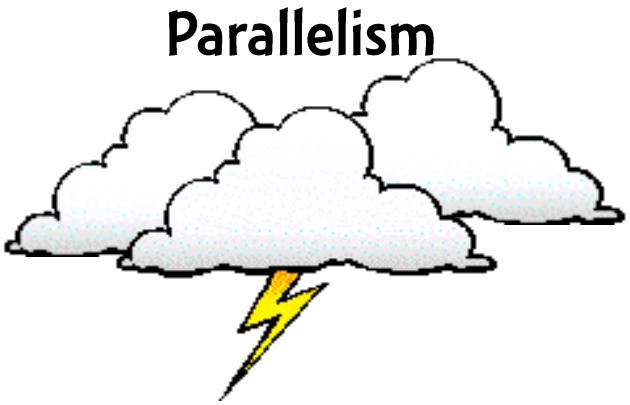
Rastislav Bodik UC Berkeley

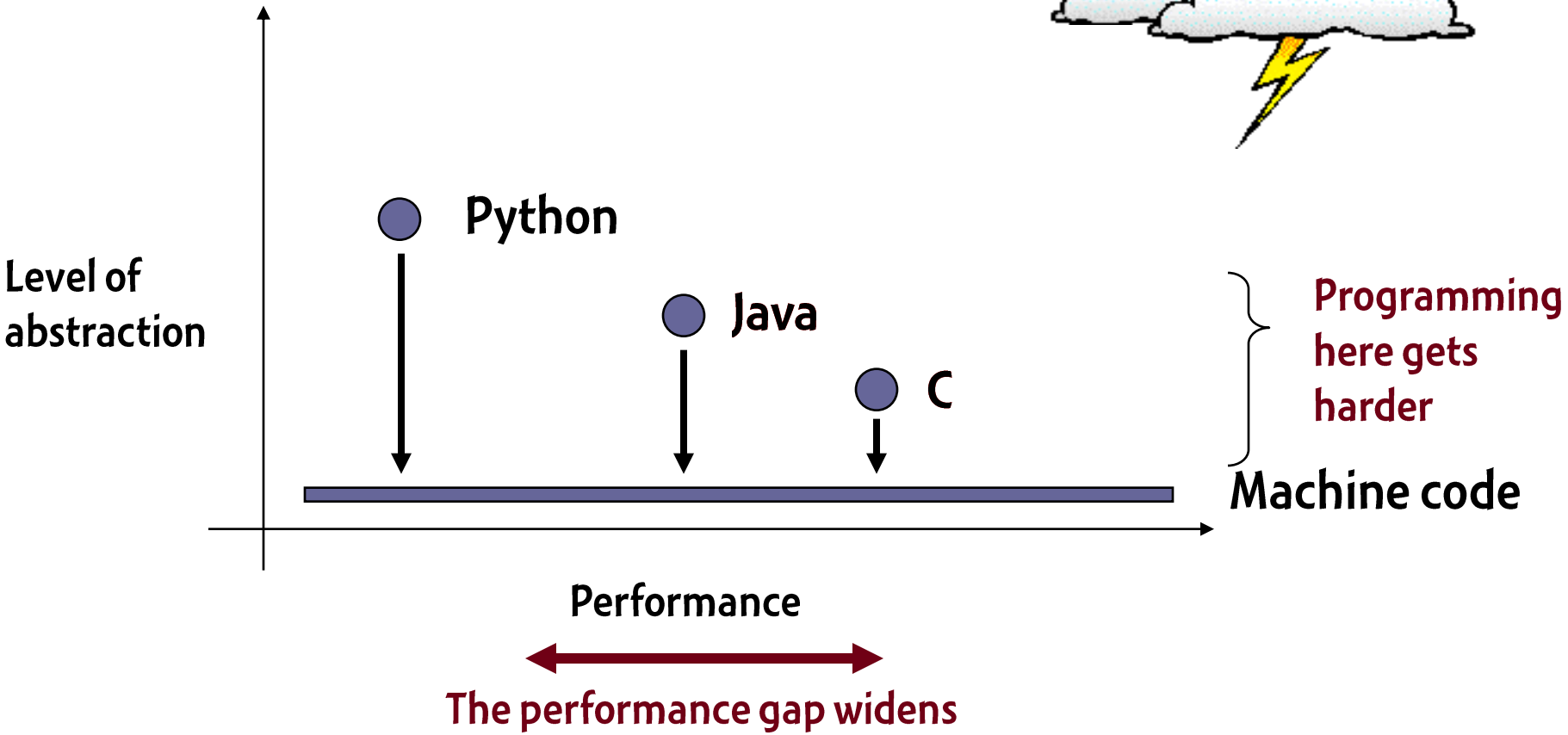
Gilad Arnold, Bob Brayton, Chris Jones, Alan Mishchenko, Armando Solar-Lezama, Koushik Sen, Sanjit Seshia, Liviu Tancau, UC Berkeley Rodric Rabbah MIT
Kemal Ebcioglu, Vijay Saraswat, Vivek Sarkar, Martin Vechev, Eran Yahav IBM
Mooly Sagiv Tel Aviv University

Synthesis

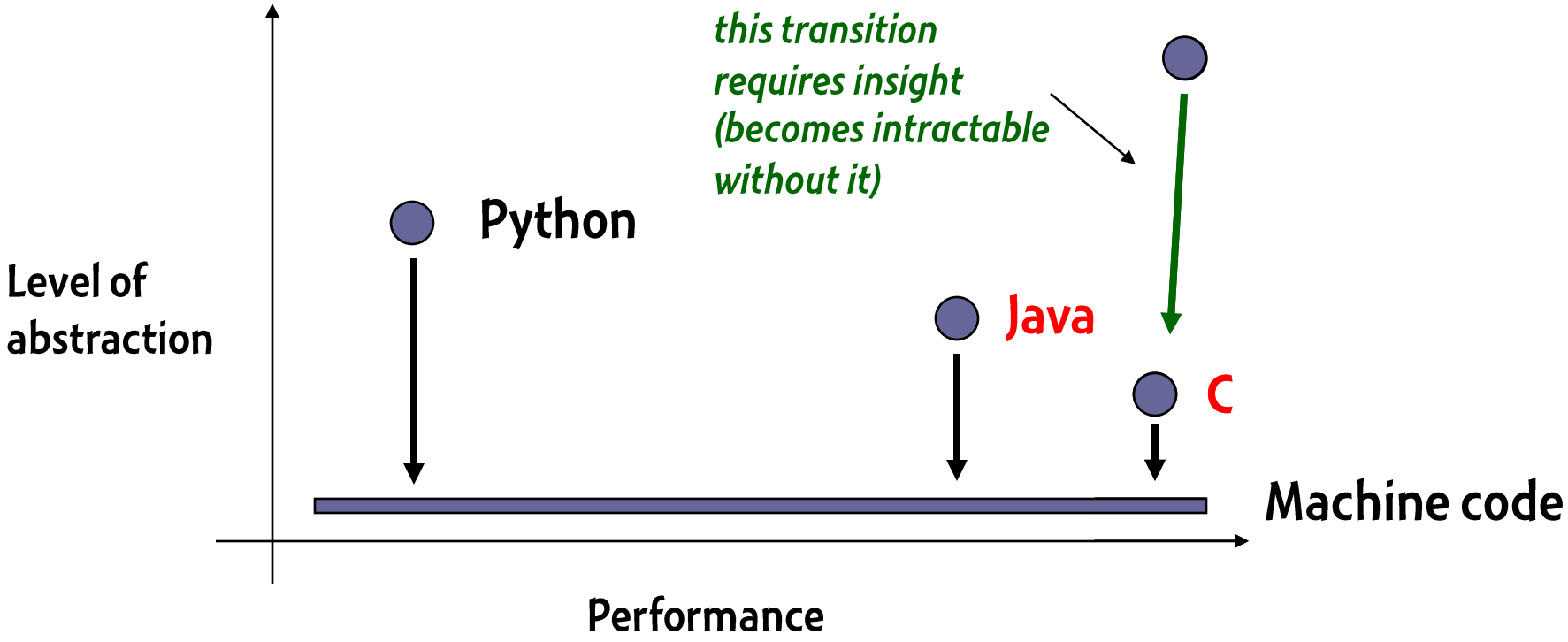
The promise: Automate program development

Abstraction Vs. Performance tradeoff





Software synthesis could help



Software synthesis could help

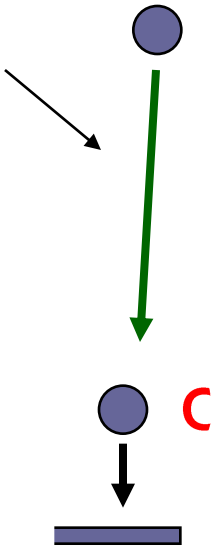
DSLs:
- StreamIt
- AutoBayes

Generators
- FFTW

– hard-coded heuristics
– domain specific transformations

Synthesizers
- PRL
- KIDS

– interactive proofs
– domain theories



Axiomatic definitions of Types and functions that may be used in the solution

Ex: merge(c, d, b) ⇔ (ordered(c) ∧ ordered(d)) ➔ (perm(c·d, b) ∧ ordered(b))

Software synthesis could help

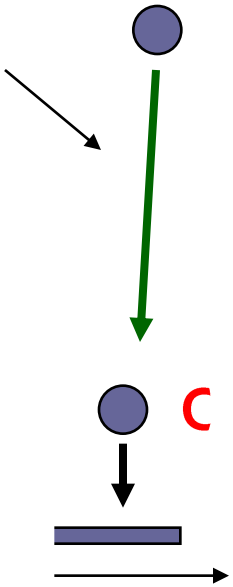
DSLs:
- StreamIt
- AutoBayes

Generators
- FFTW

– hard-coded heuristics
– domain specific transformations

Synthesizers
- PRL
- KIDS

– interactive proofs
– domain theories



It's hard for users to help

– hacking the synthesizer is very hard
– writing domain theories is very hard

The Challenge

Challenge

Establish a synergy between synthesizer and user

Insight

Big picture

Strategy

Exhaustive exploration

Details

Tactics

Research problems

Programmers must be able to contribute **expertise**

- How can the programmer contribute insights?
- How can we use these insights to make synthesis scalable?

Programmers want **control** over the implementation

- Can we avoid exposing programmers to synthesizer's internals?

Programmers **insight** will sometimes be **wrong**

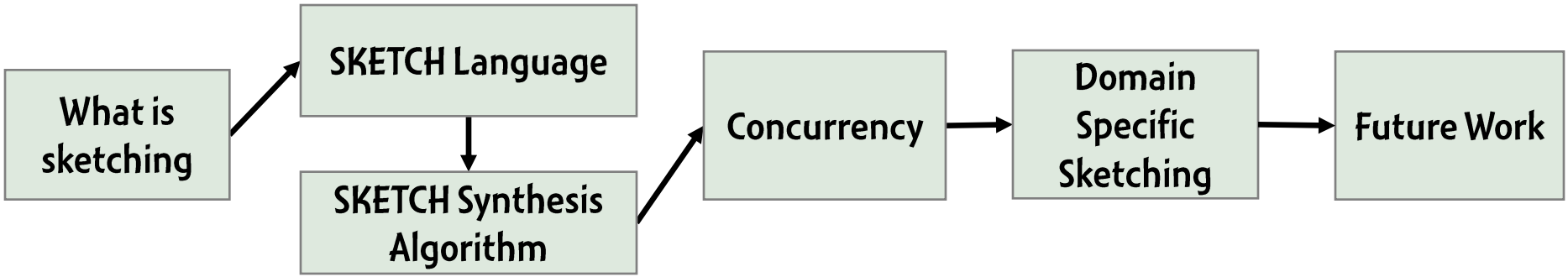
- How can we support debugging

Sketching

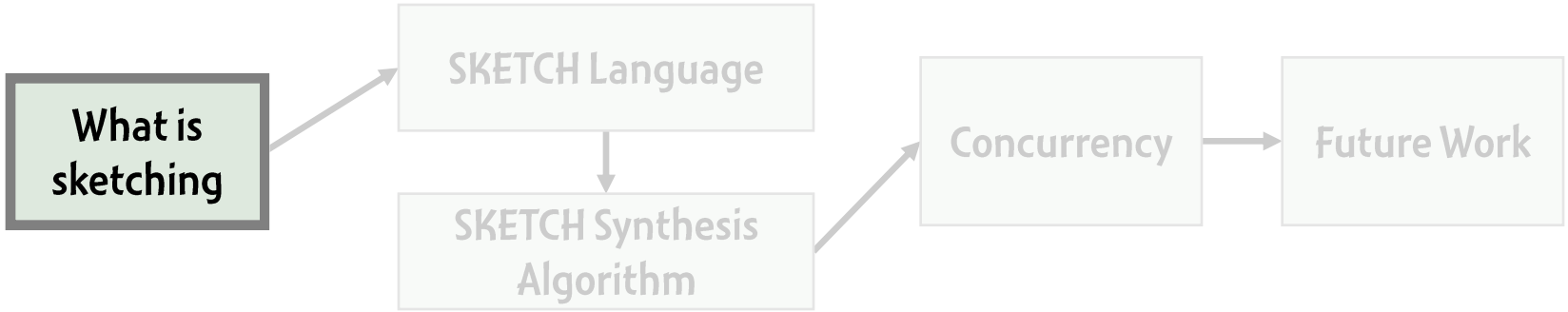
Our answer to the challenges of practical synthesis

Key Contributions:

- Synthesis from partial programs — PLDI 05
- Design of the SKETCH language — ASPLOS 06
- SKETCH Synthesis algorithm — ASPLOS 06
- Generalization to Concurrent Programs — PLDI 08
- Domain Specific Sketching through Program Reduction — PLDI 07



The Sketching Approach

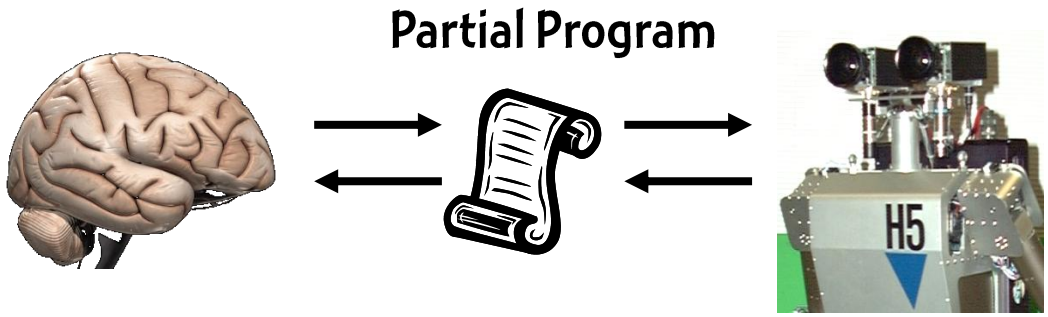


Key Observation

Insight and Mechanics are both reflected in the source code

The Sketch solution:

- Write only the code corresponding to insight
- Let the synthesizer derive the mechanics



Merge sort: first, by hand

```
int[] mergeSort (int[] input, int n) {
   if ( n == 1 ) return input;
   return merge(     mergeSort (input[0:n/2-1], n/2),
                     mergeSort (input[n/2:n-1], n-n/2), n);
}
int[] merge (int[] a, int b[], int n) {
    int j=0; int k=0;
    for (int i = 0; i < n; i++)
        if ( a[j] < b[k] )
            result[i] = a[j++];
        else
            result[i] = b[k++];
    return result;
}
```

looks simple to code, but there is a bug

Merge sort: corrected, by hand

```
int[] mergeSort (int[] input, int n) {
   if ( n == 1 ) return input;
   return merge(     mergeSort (input[0:n/2-1], n/2),
                     mergeSort (input[n/2:n-1], n-n/2), n);
}
int[] merge (int[] a, int b[], int n) {
    int j=0; int k=0;
    for (int i = 0; i < n; i++)
        if ( j<n/2 && ( !(k<n-n/2) || a[j]<b[k]) )
            result[i] = a[j++];
        else
            result[i] = b[k++];
    return result;
}
```

Merge sort: sketched

```
int[] mergeSort (int[] input, int n) {
   if ( n == 1 ) return input;
   return merge(    mergeSort (input[0:n/2-1], n/2),
                    mergeSort (input[n/2:n-1], n-n/2), n);
}
int[] merge (int[] a, int b[], int n) {
    int j=0; int k=0;
    for (int i = 0; i < n; i++)
        if (                    hole                    )
            result[i] = a[j++];
        else
            result[i] = b[k++];
    return result;
}
```

Which would a programmer prefer?

```
int[] mergeSort (int[] input, int n) {
  if ( n == 1 )
      return input;
  return merge(
           mergeSort (input[0:n/2-1], n),
           mergeSort (input[n/2:n-1], n-n/2)
         , n);
}
int[] merge (int[] a, int b[], int n) {
  int j=0; int k=0;
  for (int i = 0; i < n; i++)
    if ( [ hole ] )
      result[i] = a[j++];
    else
      result[i] = b[k++];
  return result;
}
```

```
int[] sort (int[] input, int n) {
    for (int i=0; i<n; ++i)
        for (int j=i+1; j<n; ++j)
            if (input[j] < input[i])
                swap(input, j, i);
    return input;
}
```

Theory $Sorting(\langle \alpha, \leq \rangle : linear - order)$
Imports $integer,\ bag(\alpha), seq(\alpha)$
Operations
$Ordered : seq(\alpha) \rightarrow Boolean$

Axioms
$\forall(S : seq(\alpha))\ (Ordered(S)$
$\Leftrightarrow \forall(i)(i \in \{1..length(S) - 1\} \implies S(i) \leq S(i+1)))$

Theorems
$Ordered([]) = true$
$\forall(a : \alpha)\ (Ordered([a]) = true)$
$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$
$(Ordered(y_1 +\!\!+ y_2) \Leftrightarrow Ordered(y_1)$
$\wedge\ Seq\text{–}to\text{–}bag(y_1) \leq Seq\text{–}to\text{–}bag(y_2)$
$\wedge\ Ordered(y_2))$

end–theory

Theory *Divide–and–Conquer*
Sorts D, R
Operations
$I : D \rightarrow Boolean$
$O : D \times R \rightarrow Boolean$
$primitive : D \rightarrow Boolean$
$O_{Decompose} : D \times D \times D \rightarrow Boolean$
$O_{Compose} : R \times R \times R \rightarrow Boolean$
$\succ : D \times D \rightarrow Boolean$
Soundness Axiom
$O_{Decompose}(x_0, x_1, x_2)$
$\wedge\ O(x_1, z_1)\ \wedge\ O(x_2, z_2)$
$\wedge\ O_{Compose}(z_0, z_1, z_2)$
$\implies O(x_0, z_0)$
...
end–theory

$Seq\text{–}to\text{–}bag([]) = \{\!\{\}\!\}$

$\forall(a : \alpha)\ Seq\text{–}to\text{–}bag([a]) = \{\!\{a\}\!\}$

$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$
$Seq\text{–}to\text{–}bag(y_1 +\!\!+ y_2) = Seq\text{–}to\text{–}bag(y_1)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(y_1)$

$\forall(y_1 : seq(\alpha), y_2 : seq(\alpha))$
$Seq\text{–}to\text{–}bag(y_1\ ilv\ y_2) = Seq\text{–}to\text{–}bag(y_1)\ \mathrm{d}\ Seq\text{–}to\text{–}bag(y_1)$

$O_{Decompose} \mapsto \lambda(b_0, b_1, b_2)\ b_0 = b_1\ \mathrm{d}\ b_2$
$O \mapsto \lambda(b, z)\ b = Seq\text{–}to\text{–}bag(z)\ \wedge\ Ordered(z)$

...

The sketching experience

spec +  → 

specification | sketch | implementation (completed sketch)

The spec: bubble sort

```
int[] sort (int[] input, int n) {
    for (int i=0; i<n; ++i)
        for (int j=i+1; j<n; ++j)
            if (input[j] < input[i])
                swap(input, j, i);
    return input;
}
```

Merge sort: sketched

```
int[] mergeSort (int[] input, int n) {
   if ( n == 1 ) return input;
   return merge(    mergeSort (input[0:n/2-1], n/2),
                    mergeSort (input[n/2:n-1], n-n/2), n);
}
int[] merge (int[] a, int b[], int n) {
    int j=0; int k=0;
    for (int i = 0; i < n; i++)
        if (                    hole                    )
            result[i] = a[j++];
        else
            result[i] = b[k++];
    return result;
}
```

Merge sort: sketched

```
int[] mergeSort (int[] input, int n) {
   if ( n == 1 )
   return merge
                    mergeSort (input[n/          -n/2)  n);
}
int[] merge (int[] a, ir              nt n) {
    int j=0; int k=0;
    for (int i = 0; i   n; i++)
        if ( expr(<,&&,||,!,-,[])(a, b, j, k, n, n/2 ) )
            result[i] = a[j++];
        else
            result[i] = b[k++];
    return result;
}
```

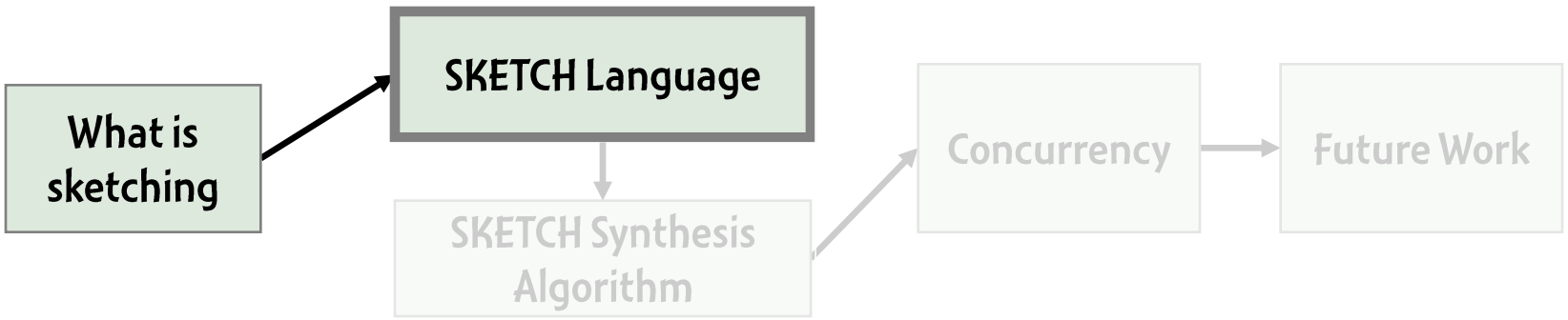
Challenge: how to help scalability?

Challenge: control over synthesized code

Merge sort: synthesized

```
int[] mergeSort (int[] input, int n) {
   if ( n == 1 ) return input;
   return merge(    mergeSort (input[0:n/2-1], n),
                    mergeSort (input[n/2:n-1], n-n/2), n);
}
int[] merge (int[] a, int b[], int n) {
    int j=0; int k=0;
    for (int i = 0; i < n; i++)
        if ( j<n/2 && ( !(k<n-n/2) || a[j]<b[k]) )
            result[i] = a[j++];
        else
            result[i] = b[k++];
    return result;
}
```

The SKETCH Language
PLDI 05, ASPLOS 06



Language design goals

Learnable

- Embed easily into a known language

Expressive

- allow programmer to express different insights about holes

Induces a simple synthesis problem

- ideally, domain independent

SKETCH: two simple 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;
}
```

Assertions can also be used to define behavior

Case study 1: Silver Medal in a SKETCH contest

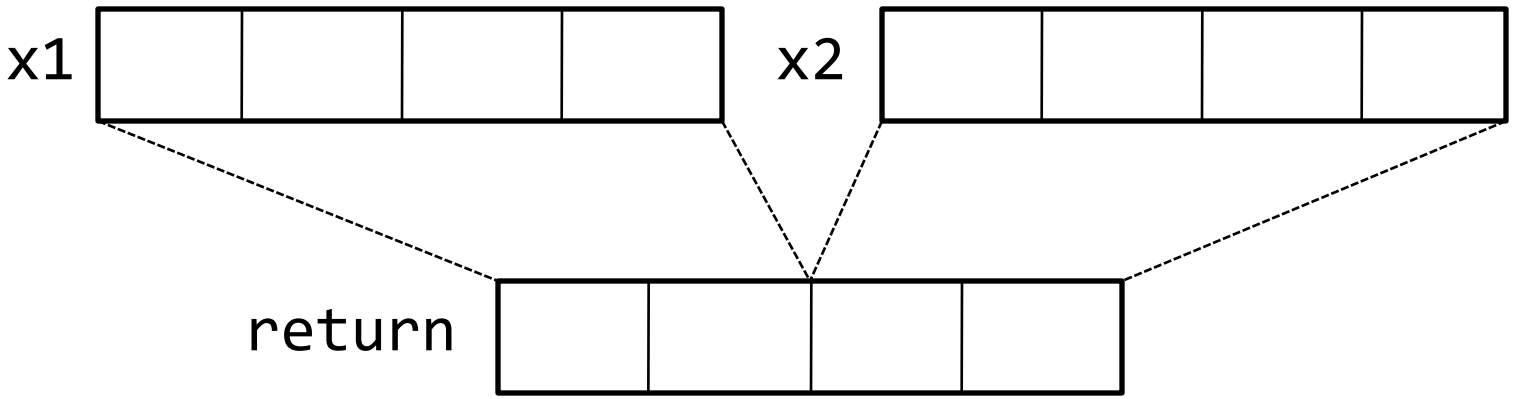
The 4x4-matrix transpose, the specification:

```
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) expressed in SKETCH:



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[
  T[8::4]   = shufps(S[
  T[0::4]   = shufps(S[
}
```

From the contestant email: Over the summer, I spent about 1/2 a day manually figuring it out. Synthesis time: 30 minutes.

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 over x: `??*x*x + ??*x + ??`

Primitive holes can be used synthesize arbitrary expressions, statements, …

- we also can make these "generators" reusable

Reusable expression generators

Following function synthesizes to one of a, b, a+b, a-b, a+b+a, ...

```
inline int expr(int a, int b){  // generator
   switch(??) {
      case 0: return a;
      case 1: return b;
      case 2: return expr(a,b) + expr(a,b);
      case 3: return expr(a,b) - expr(a,b);
   }
}
```

Synthesizing polynomials

```
int spec (int x) {
  return 2*x*x*x*x + 3*x*x*x + 7*x*x + 10;
}

int p (int x) implements spec {
  return (x+1)*(x+2)*poly(3,x);
}

inline int poly(int n, int x) {
  if (n==0) return ??;
  else return x * poly(n-1, x) + ??;
}
```

Notice the absence of any meta-variables. The generator poly() is an ordinary function.

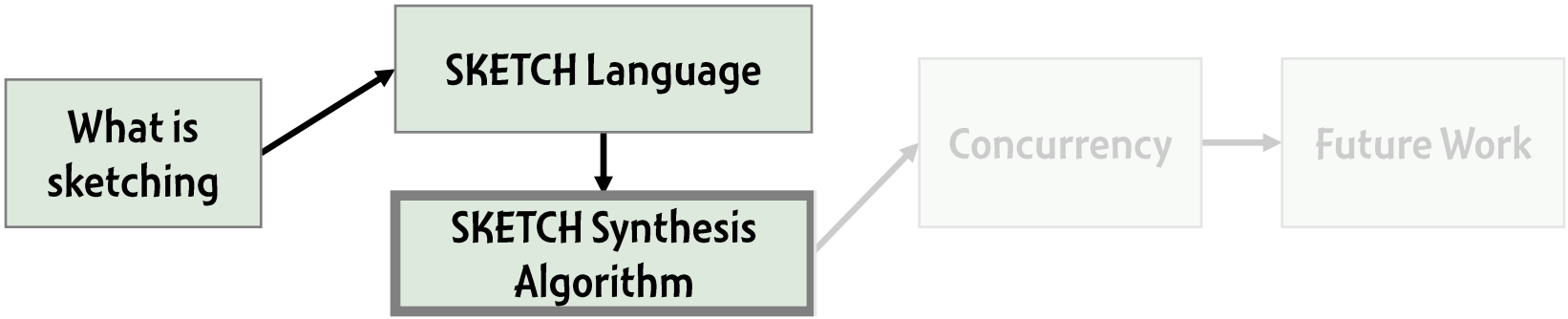
Here, SKETCH performs polynomial division. Result of division is what poly(3,x) is synthesized into.

Syntactic Sugar

Easy to add new constructs as syntactic sugar

- reorder{ s1; s2; . . . ; sn; }
- x = {| (a | b | c)(.next)? |}

The SKETCH Synthesis Algorithm



Sketch is a set of programs

A sketch syntactically describes a set of candidate programs.

- The ?? operator is modeled as a special input, called **control**:

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

The set is defined in terms of the values of the controls

S = { Sk(c) where c is an assignment to the holes}

Sketch synthesis = Search

Synthesis reduces to a search for the correct candidate

- Search for control values satisfying the following equation:

$$\exists\ c.\quad \forall\ x.\quad Spec(x) = Sk(x,c)$$

Adding additional insight reduces the search space

for the user, adding insight reduces to writing more code

programmers know how to do that

Inductive Synthesis

Synthesize from a set of observations

A little history

- Algorithmic debugging (Shapiro 1982)
- Inductive logic programming (Muggleton 1991)
- Programming by example (e.g. Lau 1999)

Two big issues

- Convergence: How do you know your solution generalizes
- Efficiency: Deriving a candidate from observations is hard

Convergence

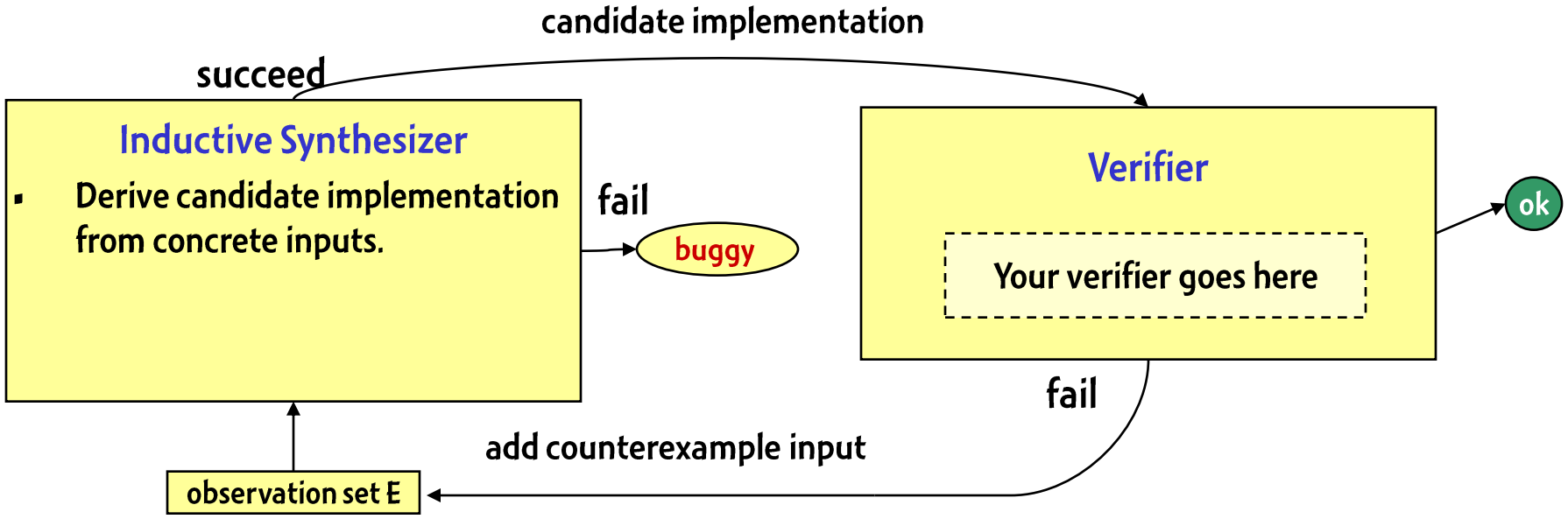
Idea: Couple Inductive synthesizer with a verifier

- Verifier is charged with detecting convergence

Counterexamples make great empirical observations

- new counterexample ➔ new information

Convergence



Convergence

Example: remove an element from a doubly linked list.

```
void remove(list l, node n){
   if (cond(l,n)) { assign(l, n); }
   if (cond(l,n)) { assign(l, n); }
   if (cond(l,n)) { assign(l, n); }
   if (cond(l,n)) { assign(l, n); }
}

int N = 6;
void test(int p){
    nodes[N] nodes;
    list l;
    initialize(l, nodes);   //… add N nodes to list
    remove(l, nodes[p]);
    checkList(nodes, l, p);
}
```

Ex: Doubly Linked List Remove

```
void remove(list l, node n)
{
  if(n.prev != l.head)
    n.next.prev = n.prev;

  if(n.prev != n.next)
    n.prev.next = n.next;

}
```

Counterexamples
p = 3

Ex: Doubly Linked List Remove

```
void remove(list l, node n)
{
  if(n.prev != null)
    n.next.prev = n.prev;

  if(l.head == n)
    l.head = n.next;

  l.tail = l.tail;

  if(l.head!=n.next)
    n.prev.next = n.next;
}
```

Counterexamples
p = 3
p = 0

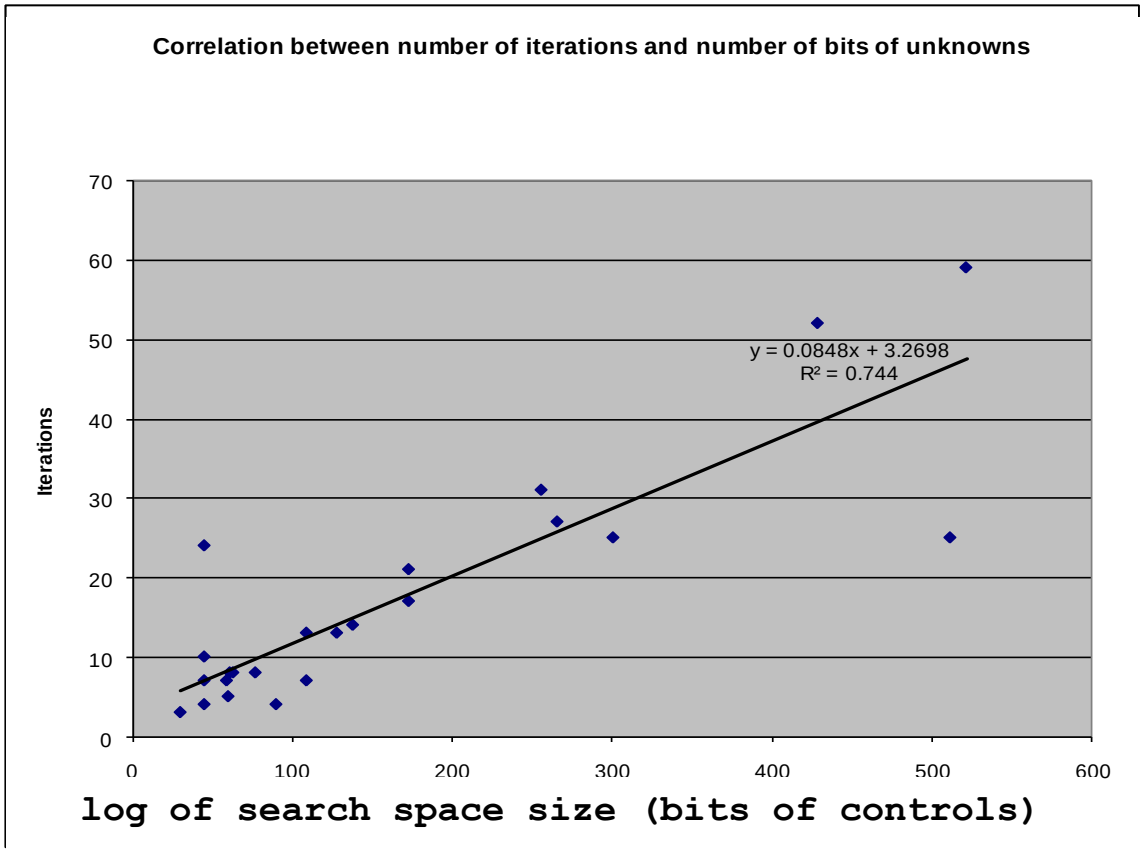
Ex: Doubly Linked List Remove

```
void remove(list l, node n)
{
  if(n.prev == null)
     l.head = n.next;

  if(n.next == null)
    l.tail = n.prev;

  if(n.next != l.head)
    n.prev.next = n.next;

  if(n.next != null)
    n.next.prev = n.prev;
}
```

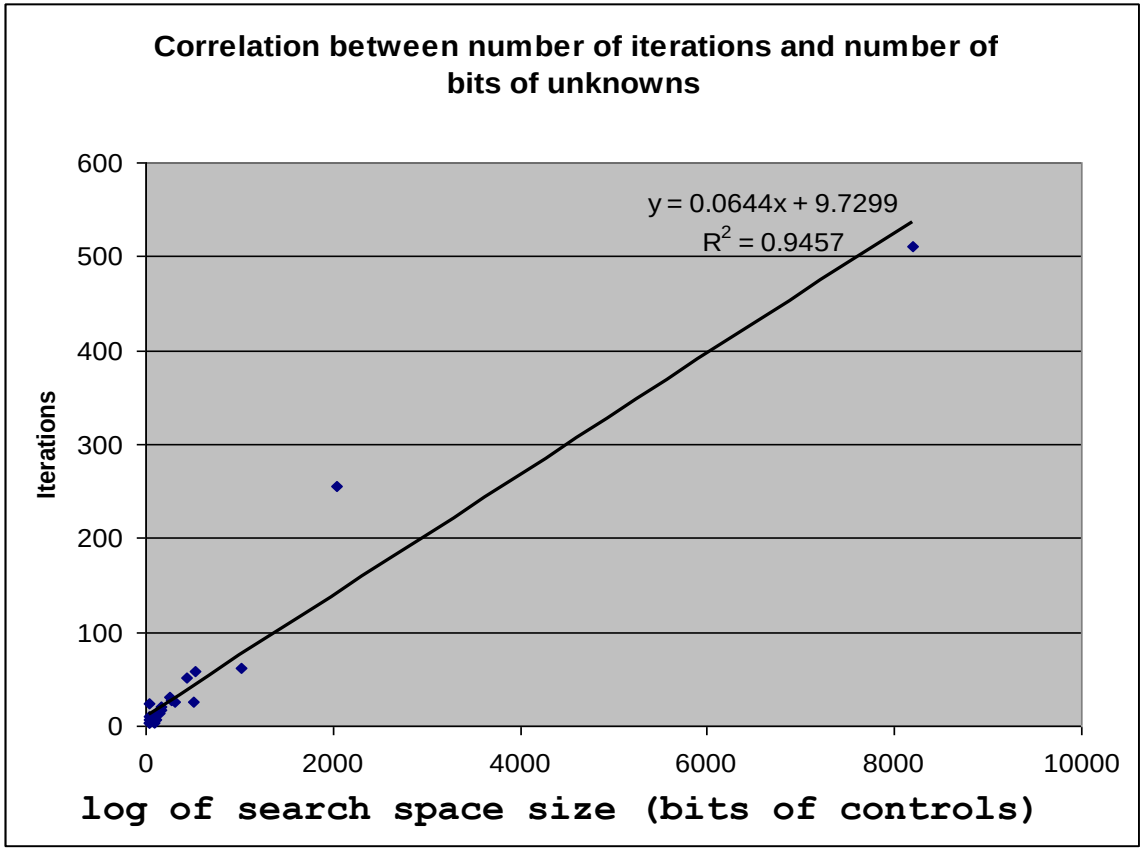
Counterexamples
p = 3
p = 0
p = 5

Process takes < 1 second

The number of counterexample inputs is small



The number of counterexample inputs is small



Inductive Synthesis

Deriving a candidate from a set of observations

Key:

- Frame as a constraint satisfaction problem
- Avoid enumeration; use algebraic techniques instead

Encode candidate space as a bit-vector

- Natural encoding given the integer holes

Encode synthesis as boolean constraints on bit-vector

$$\exists\ c.\quad \forall\ x \text{ in } E.\ \mathrm{Spec}(x)\ =\ \mathrm{Sk}(x,c)$$

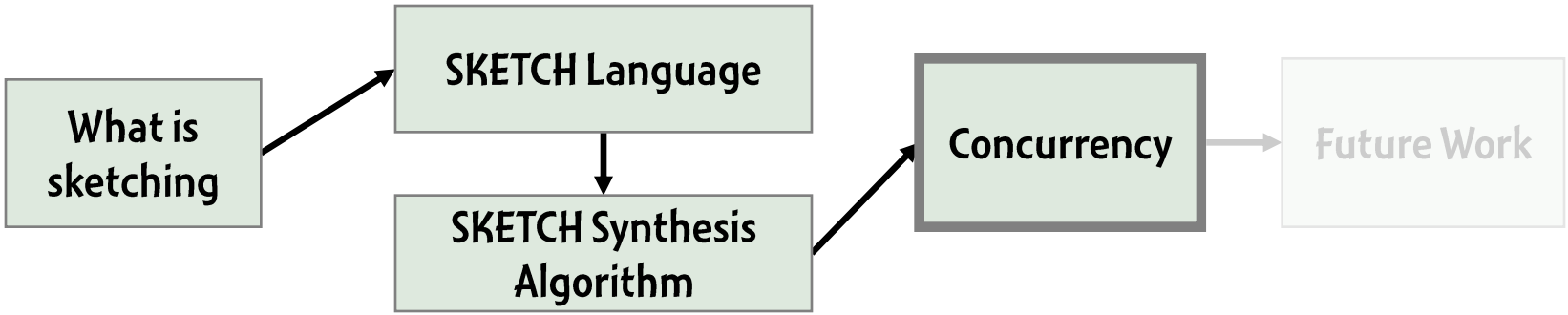
where $E = \{x_1, x_2, \ldots, x_k\}$

Solve constraints using SAT solver

Interesting Benchmarks

Benchmark	Unknowns	Solution Time
AES	32769 bits	78.8 min
SSE Matrix Transpose	24 integers + 64 bits	30 min
CRC	8192 bits	13.5 min
Doubly Linked List Remove	60 bits	< 1 sec
Enqueue	271 int & bit	1 min
16 bit Morton Numbers	332 int & bit	20 min
32 bit fast parity	45 bits	11 sec
Sort (bounded)	363 int & bit	3.5 min

Concurrency

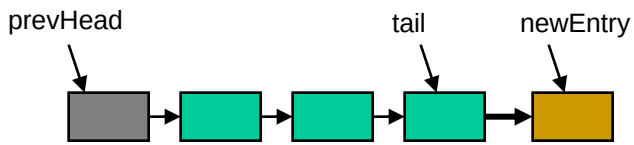


Concurrent programs

Ex: Concurrent Enqueue using AtomicSwap

```
Object AtomicSwap(ref Object loc, Object entry)
atomic {
    Object old = loc;
    loc = entry;
    return old;
}
```

```
class Queue {
  QueueEntry prevHead =
                        new QueueEntry(null);
  QueueEntry tail = prevHead;

  void Enqueue(Object newobject) {
    Node tmp = null;
    newEntry = new QueueEntry(newobject);

    tmp = AtomicSwap(tail , newEntry );
    tmp.next = newEntry ;

  }
}
```



Generalization to Parallelism

Output now dependent on thread interleaving

- Make interleaving schedule part of the observations
- Most verifiers can provide a counterexample trace

Using the observations becomes harder

- Schedule generated as witness for a given candidate
- How do we use it to rule out other incorrect candidates?