Automatic Programming Revisited

Part II: Synthesizer Algorithms

Rastislav Bodik

University of California, Berkeley

Outline of Part II

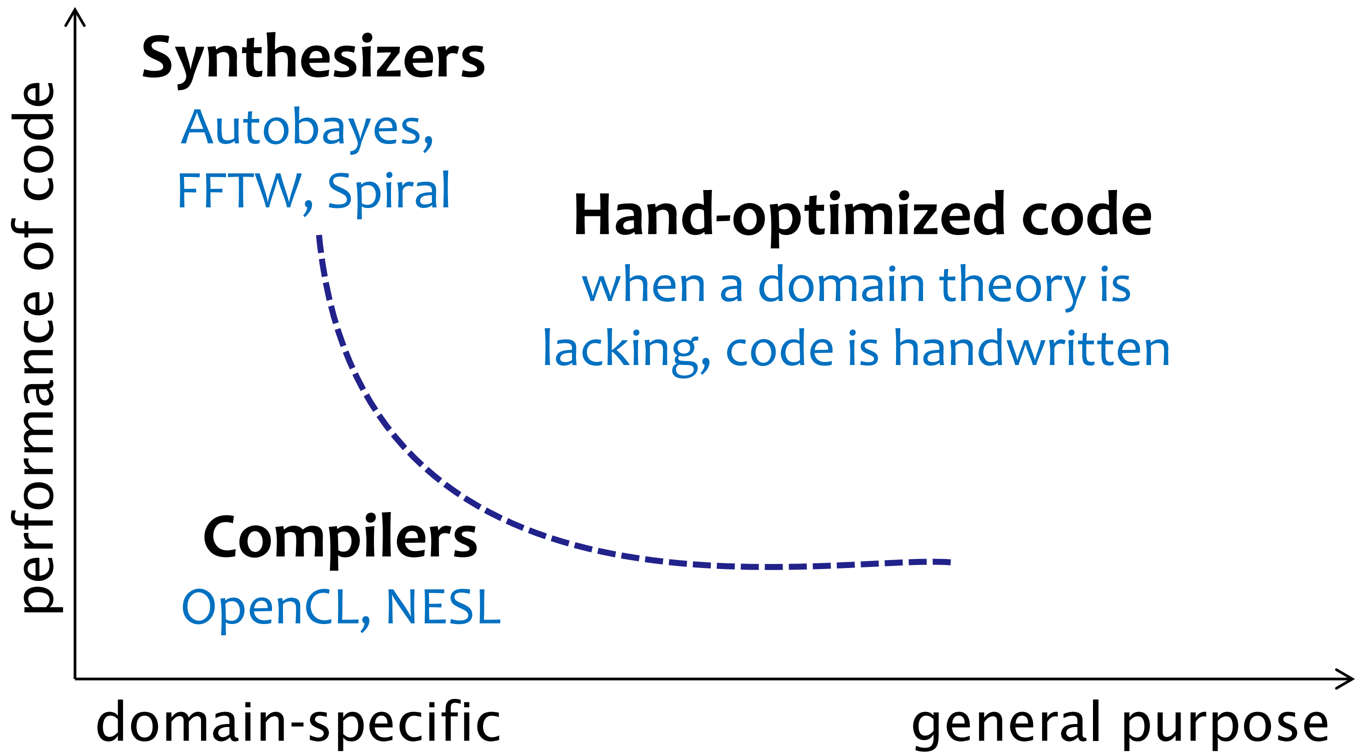
Synthesizer algorithms

Future directions:

- concurrency
- domain-specific synthesis (dynamic programming)

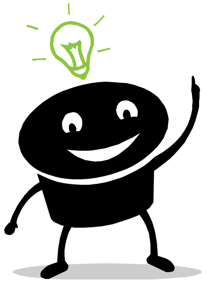
Other partial program synthesizers

What's between compilers and synthesizers?



Our approach: *help programmers auto-write code without (us or them) having to invent a domain theory*

Automating code writing



SKETCH

SKETCH: just two constructs

spec:

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

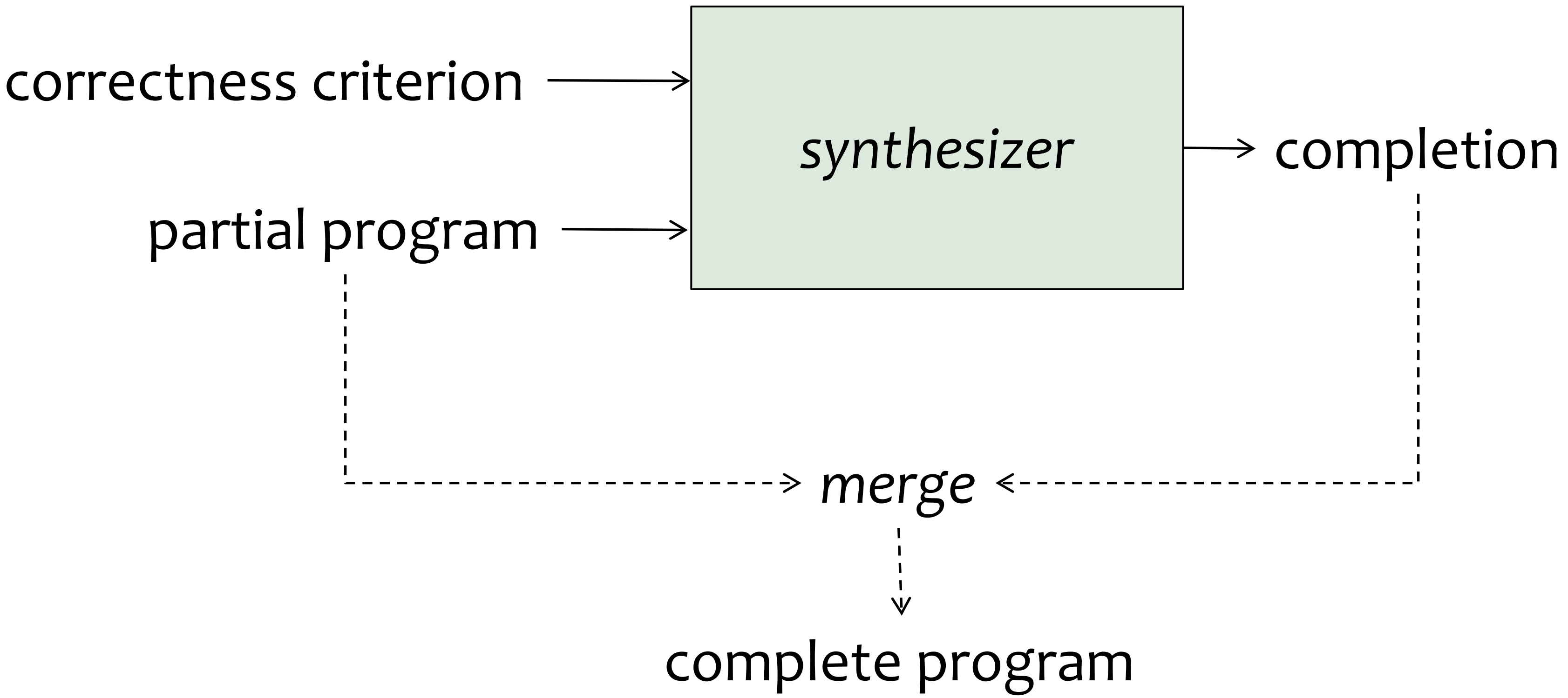
sketch:

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

result:

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

It’s synthesis from partial programs



The price SKETCH pays for generality

What are the limitations behind the magic?

Sketch doesn’t produce a proof of correctness:

SKETCH checks correctness of the synthesized program on all inputs of up to certain size. The program could be incorrect on larger inputs. This check is up to programmer.

Scalability:

Some programs are too hard to synthesize. We propose to use refinement, which provides modularity and breaks the synthesis task into smaller problems.

Counterexample-Guided Inductive Synthesis (CEGIS)

How it works

Step 1: Turn holes into control inputs

Step 2: Translate spec and sketch to boolean functions

Step 3: Formulate synthesis as generalized SAT

Step 4: Solve with counterexample guided search

Step 5: Plug controls into the sketch

Making the candidate space explicit

A sketch syntactically describes a set of candidate programs.

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

```
int f(int x) {
      … ?? … ?? …
}
```

→

```
int f(int x, int c1, c2) {
      … c1 … c2 …
}
```

What about recursion?

– calls are unrolled (inlined) => distinct ?? in each invocation

⇒ unbounded number of ?? in principle

– but we want to synthesize bounded programs, so unroll until you found a correct program or run out of time

How it works

Step 1: Turn holes into control inputs

Step 2: Translate spec and sketch to boolean functions

Step 3: Formulate synthesis as generalized SAT

Step 4: Solve with counterexample guided search

Step 5: Plug controls into the sketch

Must first create a bounded program

Bounded program:

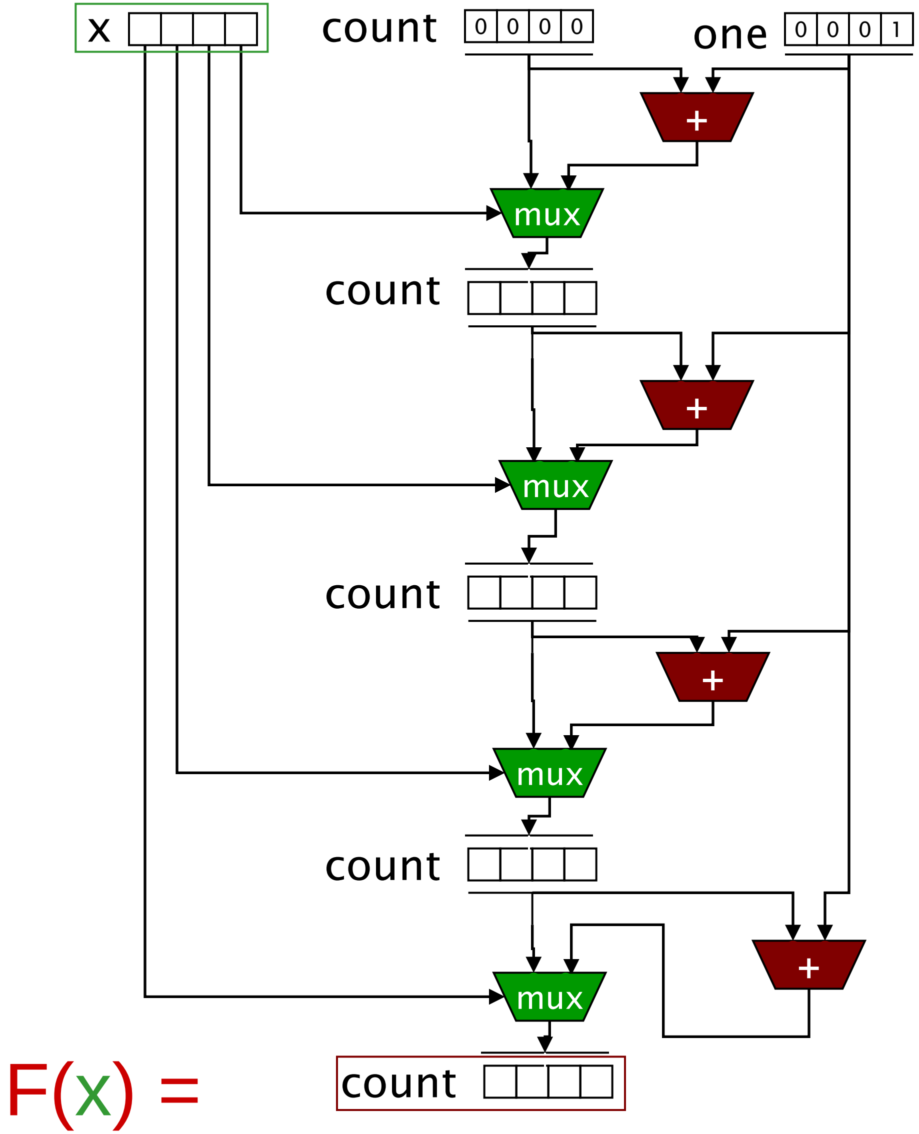
- executes in bounded number of steps

One way to bound a program:

- bound the size of the input, and
- work with programs that always terminate

Ex : bit population count.

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



How it works

Step 1: Turn holes into control inputs

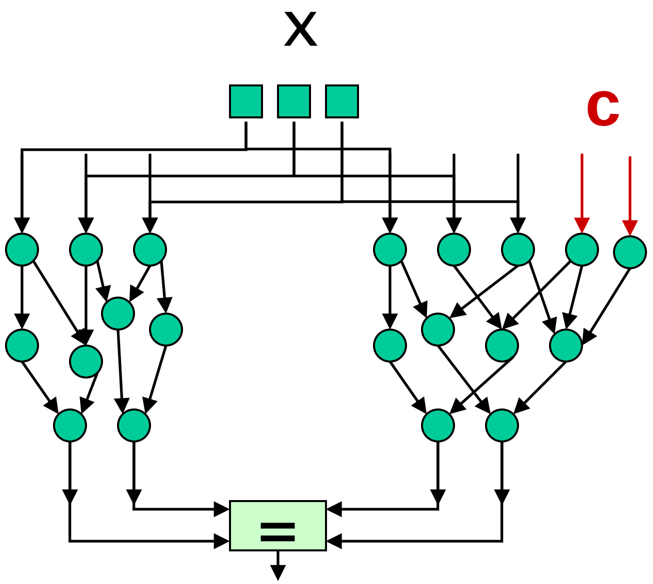
Step 2: Translate spec and sketch to boolean functions

Step 3: Formulate synthesis as generalized SAT

Step 4: Solve with counterexample guided search

Step 5: Plug controls into the sketch

Putting together sketch and spec



Sketch synthesis is constraint satisfaction

Synthesis reduces to solving this satisfiability problem

- synthesized program is determined by c

$$\exists\, c.\quad \forall\, x.\ \ \mathrm{spec}(x) \;=\; \mathrm{sketch}(x, c)$$

Quantifier alternation is challenging. Our idea is to turn to inductive synthesis

How it works

Step 1: Turn holes into control inputs

Step 2: Translate spec and skretch to boolean functions

Step 3: Formulate synthesis as generalized SAT

Step 4: Solve with counterexample guided search

Step 5: Plug controls into the sketch

Inductive Synthesis

Synthesize a program from a set of **input-output observations**

Some history

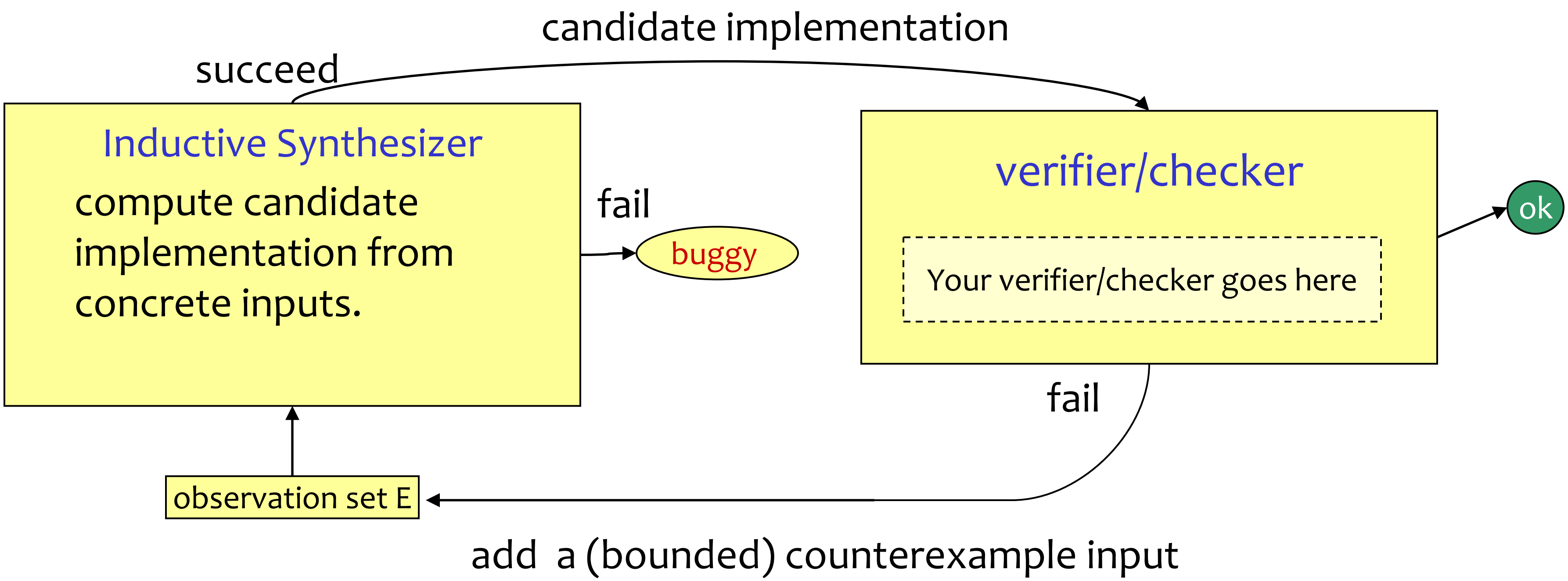
- Algorithmic debugging (Shapiro 1982)
- Inductive logic programming (Muggleton 1991)
- Programming by example (e.g. Lau 2001)

Three big issues

- **Convergence:** How do you know your solution generalizes?
- **Suitable observations:** Where to obtain them?
- **Efficiency**: Computing a candidate correct on a few observations is still hard

CounterExample –Guided Inductive Synthesis

The CEGIS algorithm:



Inductive synthesis step implemented with a SAT solver

CEGIS: Summary

Inductive synthesizer could be **adversarial**

- so we constrain it to space of candidates described by the sketch

Finding **convergence** (is resulting program correct?)

- we charge a checker with detecting convergence

Counterexamples make good empirical observations

- new counterexample covers a new "corner case"

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

Synthesis as generalized SAT

- The sketch synthesis problem is an instance of 2QBF:

$$\exists\, c.\quad \forall\quad x.\;\; \text{spec}(x) \;=\; \text{sketch}(x, c)$$

- Counter-example driven solver:

```
I = {}
x = random()
do
   I = I U {x}
   c = synthesizeForSomeInputs(I)
   if c = nil then exit("buggy sketch")
   x = verifyForAllInputs(c)          // x: counter-example
while x != nil
return c
```

$S(x_1, c)=F(x_1)$ & … & $S(x_k, c)=F(x_k)$

$I =\{ x_1, x_2, \ldots, x_k \}$

$S(x, c) \mathrel{!=} F(x)$

How it works

Step 1: Turn holes into control inputs

Step 2: Translate spec and sketch to boolean functions

Step 3: Formulate synthesis as generalized SAT

Step 4: Solve with counterexample guided search

Step 5: Plug controls into the sketch

Exhaustive search not scalable

Option 0: Exploring all programs in the language

- for the concurrent list: space of about 10^{30} candidates
- if each candidate tested in 1 CPU cycle: ~age of universe

Option 1: Reduce candidate space with a sketch

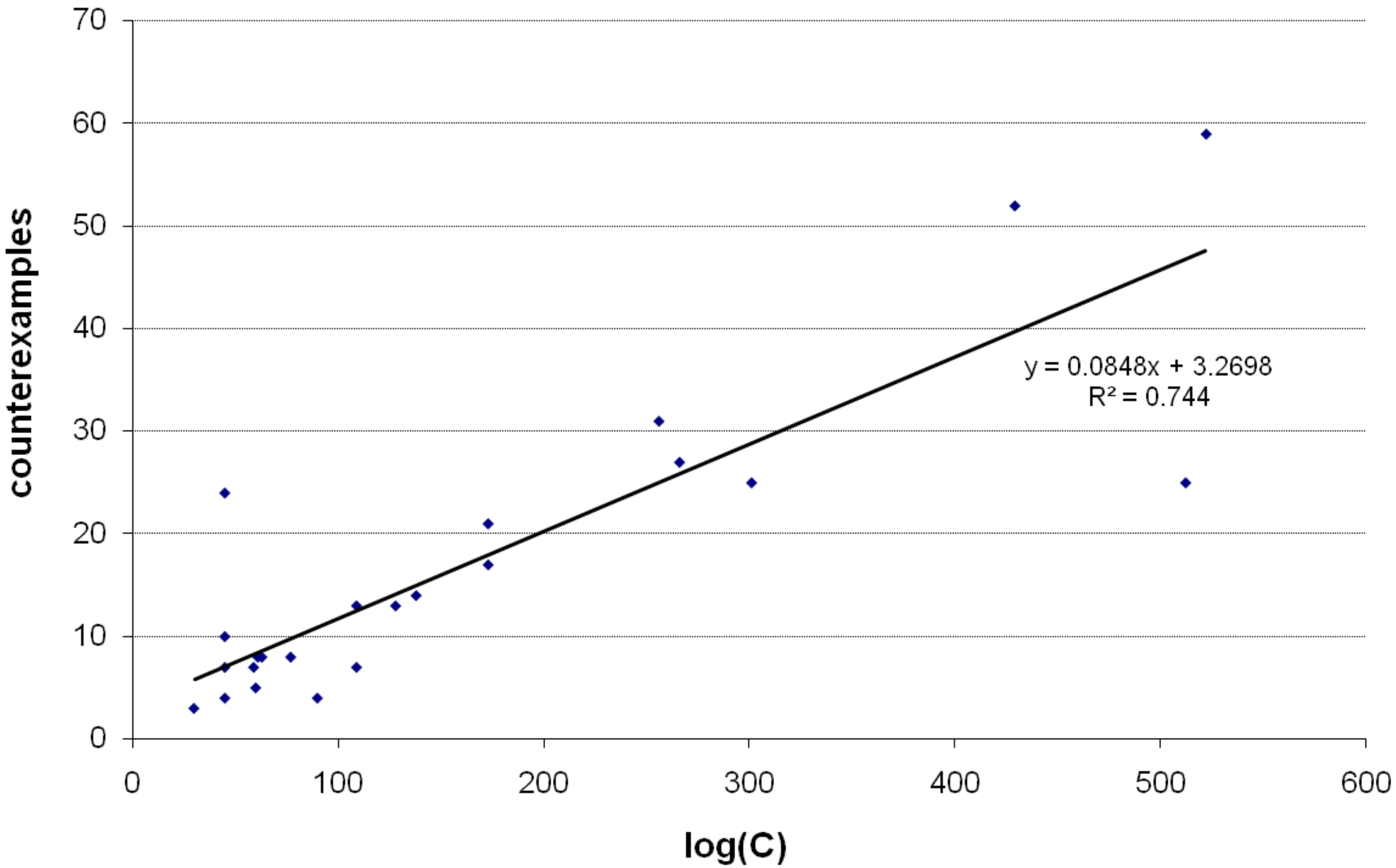
- concurrent list sketch: candidate space goes down to 10^{9}
- 1sec/validation ==> about 10-100 days (assuming that the space contains 100-1000 correct candidates)
- but our spaces are sometimes 10^{800}

Option 2: Find a correct candidate with CEGIS

- concurrent list sketch: 1 minute (3 CEGIS iterations)

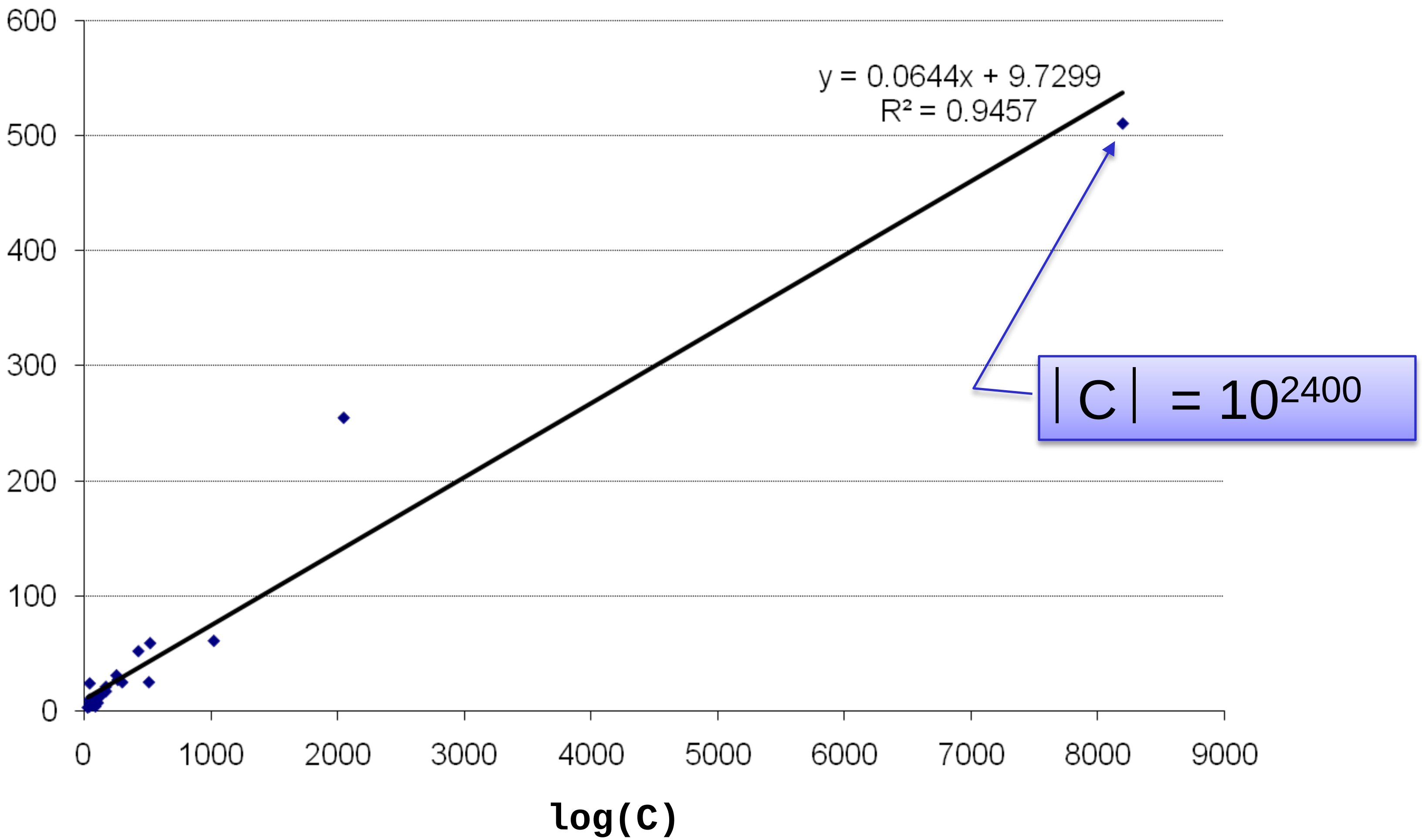
Number of counterexample vs. log(C)

C = size of candidate space = exp(bits of controls)



Number of counterexample vs. log(C)

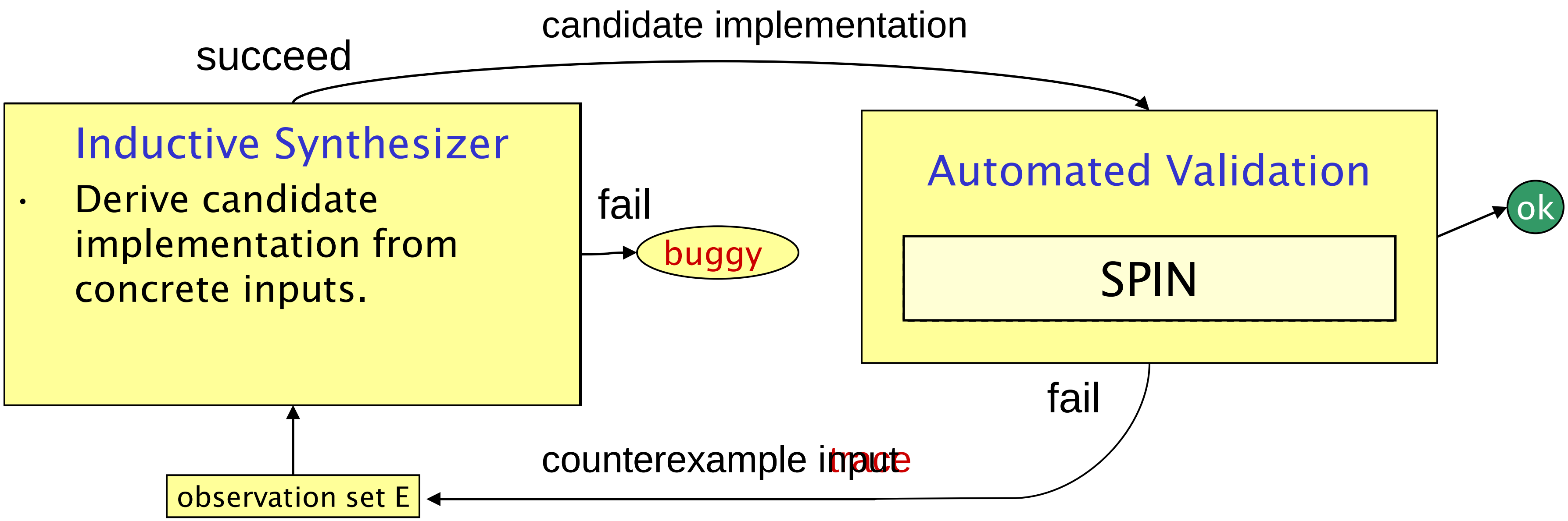
C = size of candidate space = exp(bits of controls)



Synthesis of Concurrent Programs

CEGIS for Concurrent Programs

~~Sequential~~ Concurrent

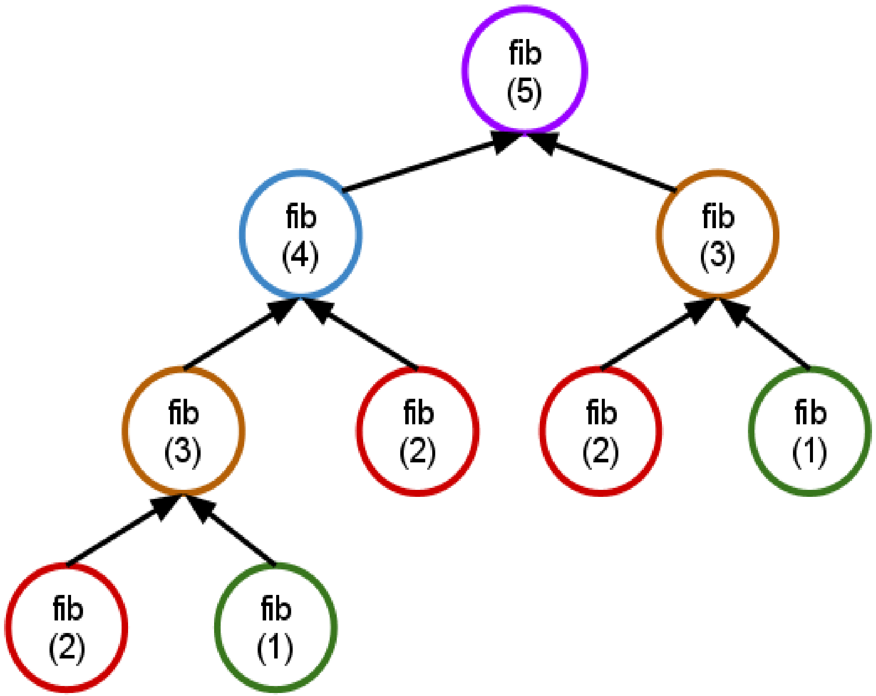


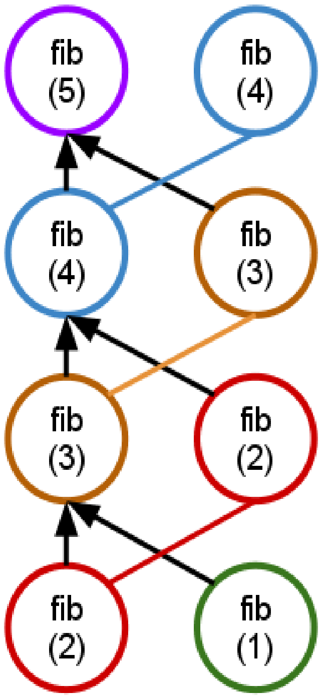
Synthesis of Dynamic Programming

Dynamic Programming

Compute $O(2^n)$ algorithms in $O(n^k)$ time

Example: *fib(n)*





Challenges in DP algorithm design

The divide problem: Suitable sub-problems often not stated in the original problem. We may need to invent different subproblems.

The conquer problem: Solve the problem from subproblems by formulate new recurrences over discovered subproblems.

Maximal Independent Sum (MIS)

Given an array of positive integers, find a non-consecutive selection that returns the best sum and return the best sum.

Examples:

mis([4,2,1,4]) = 8
mis([1,3,2,4]) = 7

Exponential Specification for MIS

The user can define a specification as an clean exponential algorithm:

```
mis(A):
    best = 0
    forall selections:
        if legal(selection):
            best = max(best, eval(selection, A))
    return best
```

Sketch = "shape" of the algorithm

```
def linear_mis(A):
  tmp1 = array()
  tmp2 = array()
  tmp1[0] = initialize1()
  tmp2[0] = initialize2()
  for i from 1 to n:
    tmp1 = prop1(tmp1[i-1],tmp2[i-1],A[i-1])
    tmp2 = prop2(tmp1[i-1],tmp2[i-1],A[i-1])
  return term(tmp1[n],tmp2[n])
```

Synthesize propagation functions

```
def prop (x,y,z) :=
    switch (??)
    case 0: return x
    case 1: return y
    case 2: return z
    case 3: return unary(prop(x,y,z))
    ...
    case r: return binary(prop(x,y,z),
                          prop(x,y,z))
```

MIS: The synthesized algorithm

```
linear_mis(A):
  tmp1 = array()
  tmp2 = array()
  tmp1[0] = 0
  tmp2[0] = 0
  for i from 1 to n:
    tmp1[i] = tmp2[i-1] + A[i-1]
    tmp2[i] = max(tmp1[i-1],tmp2[i-1])
  return max(tmp1[n],tmp2[n])
```

A guy walks into a Google Interview …

Given an array of integers $A=[a_1, a_2, \ldots, a_n]$,
return $B=[b_1, b_2, \ldots , b_n]$
such that: $b_i = a_1 + \ldots + a_n - a_i$

Time complexity must be $O(n)$

Can't use subtraction

Google Interview Problem: Solution

```
puzzle(A):
  B = template1(A)
  C = template2(A,B)
  D = template3(A,B,C)
  return D

template1(A):
  tmp1 = array()
  tmp1[0] = 0
  for i from 1 to n-1:
    tmp1[i] = tmp[i-1]+A[n-1]
  return tmp1
```

```
template2(A,B):
  tmp2 = array()
  tmp2[n-1] = 0
  for i from 1 to n-1:
    tmp2[n-i-1]
          = tmp2[n-i]+A[n-i]

template3(A,B,C):
  tmp3 = array()
  for i from 0 to n-1:
    tmp3[i] = B[i] + C[i]
  return tmp3
```

aLisp

[Andre, Bhaskara, Russell, … 2002]

aLisp: learning with partial programs

Problem:

- implementing AI game opponents (state explosion)
- ML can't efficiently learn how agent should behave
- programmers take months to implement a decent player

Solution:

- programmer supplies a skeleton of the intelligent agent
- ML fills in the details based on a reward function

Synthesizer:

- hierarchical reinforcement learning

What's in the partial program?

Strategic decisions, for example:

- first train a few peasant
- then, send them to collect resources (wood, gold)
- when enough wood, reassign peasants to build barracks
- when barracks done, train footmen
- better to attack with <u>groups</u> of footmen rather than send a footman to attack as soon as he is trained

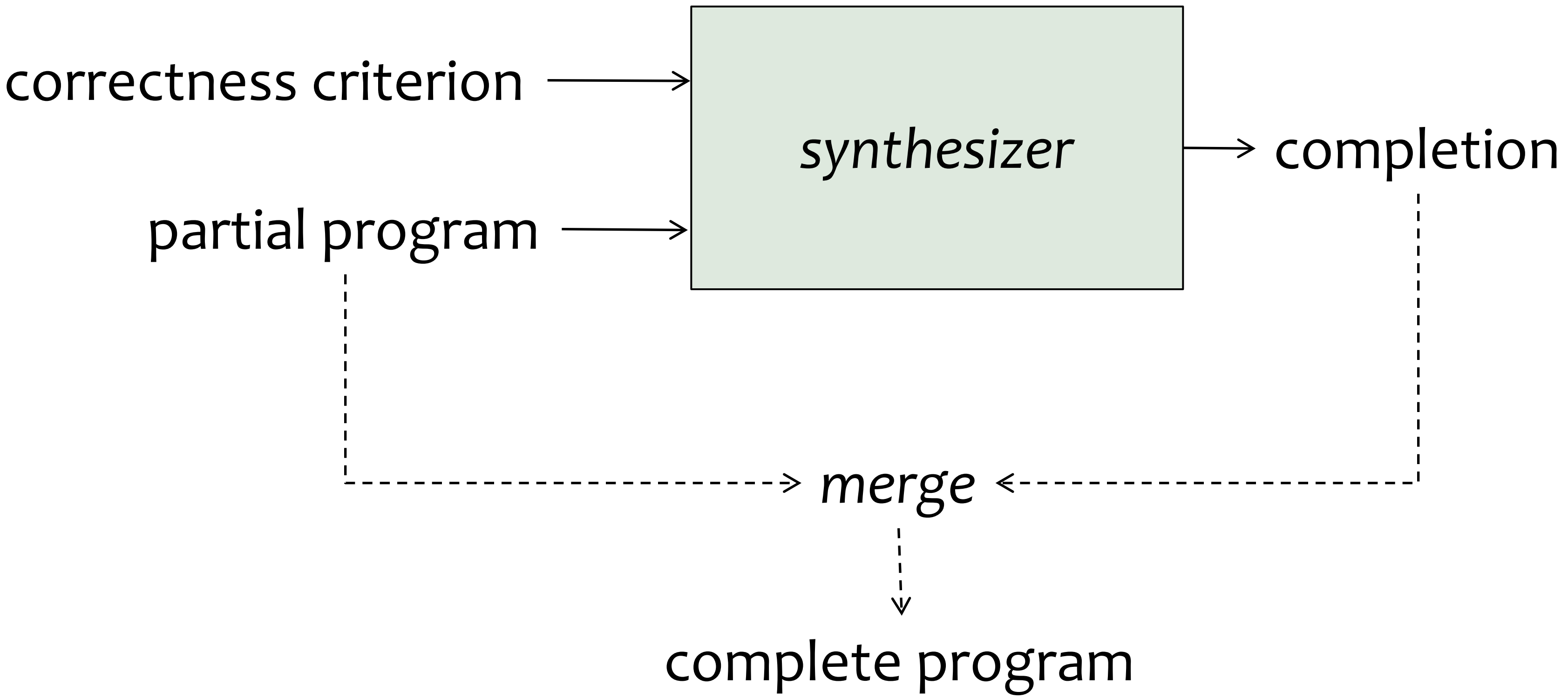
[from Bhaskara et al IJCAI 2005]

Fragment from the aLisp program

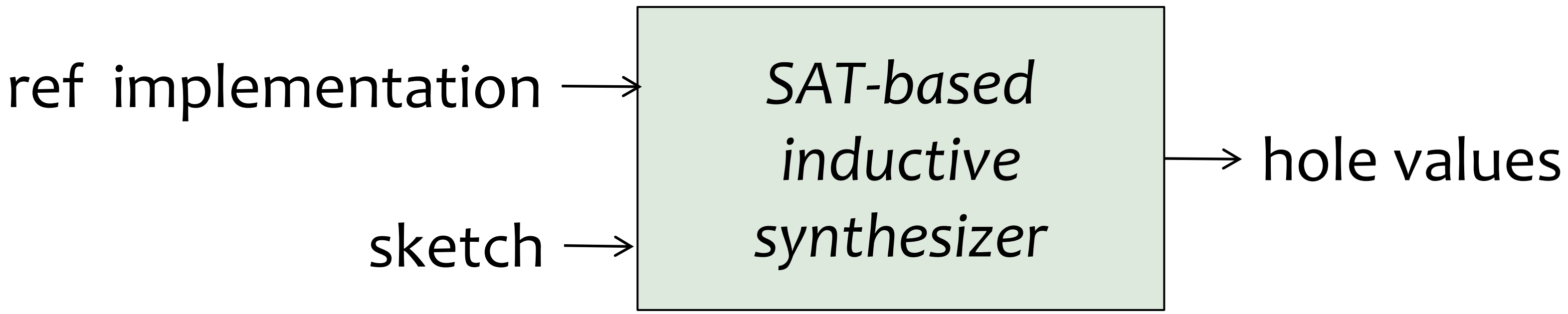
```
(defun single-peasant-top ()
   (loop do
         (choose '((call get-gold) (call get-wood)))))

(defun get-wood ()
   (call nav (choose *forests*))
   (action 'get-wood)
   (call nav *home-base-loc*)
   (action 'dropoff))

(defun nav (l)
   (loop until (at-pos l) do
   (action (choose '(N S E W Rest)))))
```

this.x > l.x then go West
check for conflicts
…

It's synthesis from partial programs

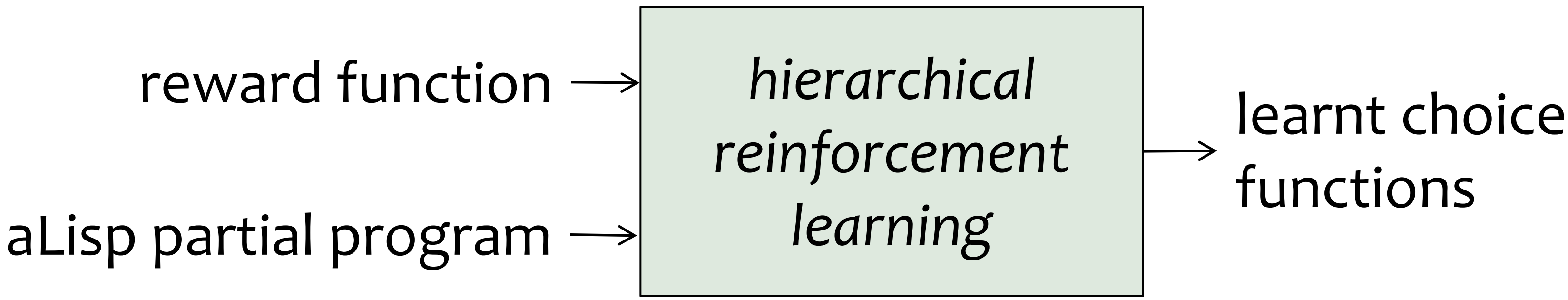


SKETCH



ref implementation
sketch
SAT-based inductive synthesizer
hole values

aLisp


reward function
aLisp partial program
hierarchical reinforcement learning
learnt choice functions

First problem with partial programming

Where does specification of correctness come from?

Can it be developed faster than the program itself?

Unit tests (input,output pairs) sometimes suffice.

Next two projects go in the direction of saying even less.

SMARTedit*

[Lau, Wolfman, Domingos, Weld 2000]

SMARTedit*

Problem:

- creation of editor macros by non-programmers

Solution:

- user demonstrates the steps of the desired macro
- she repeats until the learnt macro is unambiguous
- *unambiguous* = all plausible macros transform the provided input file in the same way

Solver:

- version space algebra

An editing task: EndNote to BibTex

%0 Journal Article
%1 4575
%A ^Richard C. Waters
%T The Programmer's Apprentice: A Session with KBEmacs
%J IEEE Trans. Softw. Eng.
%@ 0098-5589
%V 11
%N 11
%P 1296-1320
%D 1985
%R http://dx.doi.org/10.1109/TSE.1985.231880
%I IEEE Press

→

```
@article{4575,
    author = {Waters, Richard C.},
    title = {The Programmer's Apprentice: A Session with KBEmacs},
    journal = {IEEE Trans. Softw. Eng.},
    volume = {11}, number = {11}, year = {1985},
    issn = {0098-5589},
    pages = {1296--1320},
    doi = {http://dx.doi.org/10.1109/TSE.1985.231880},
    publisher = {IEEE Press}, address = {Piscataway, NJ, USA},
}
```

Demonstration = sequence of program states:

```
1) cursor in (0,0)    buffer = "%0 …"    clipboard = ""
2) cursor in ^        buffer = "%0 …"    clipboard = ""
3) …
```

Desired macro:

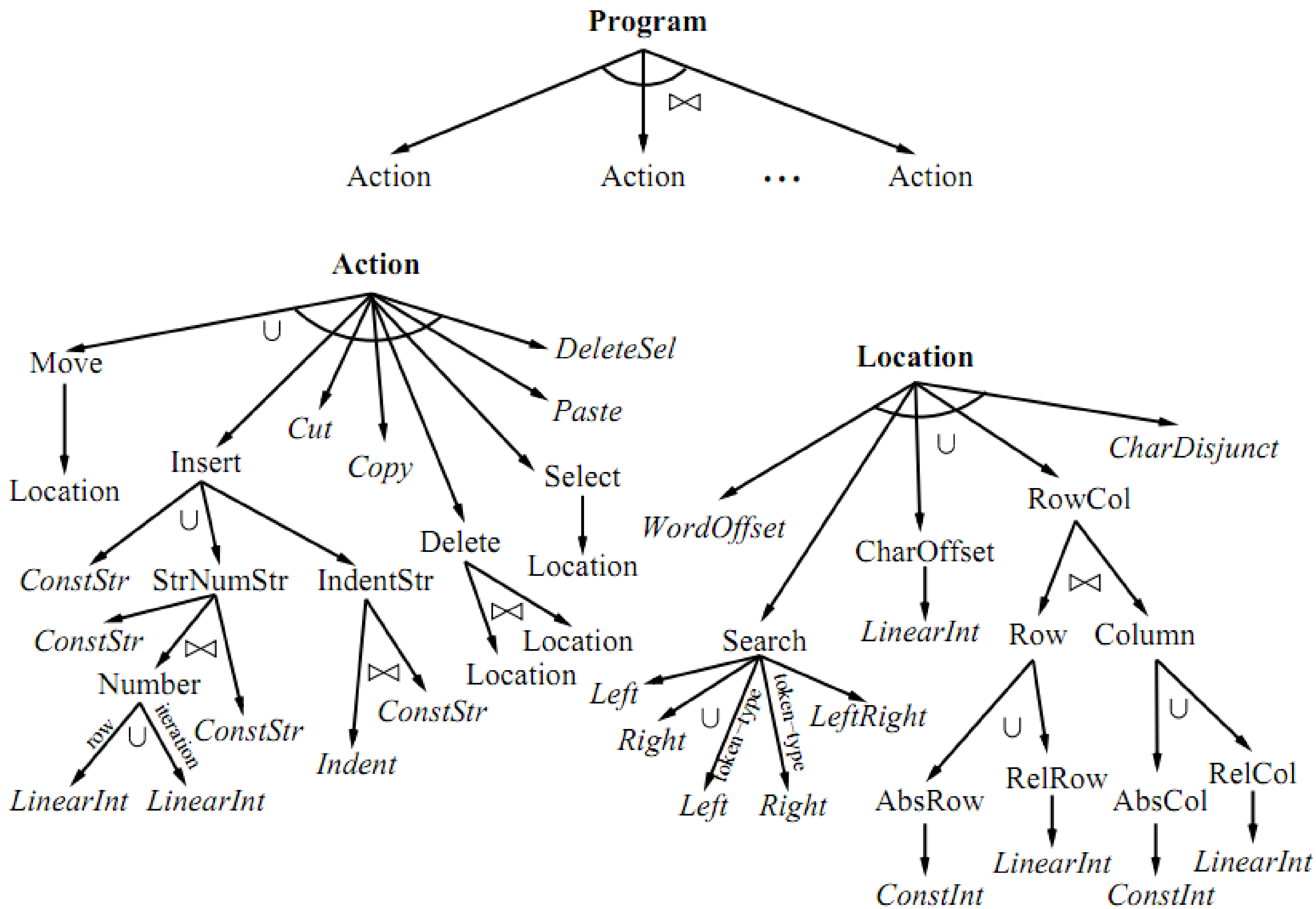
```
move(to after string "%A ")
…
```

Version space = space of candidate macros

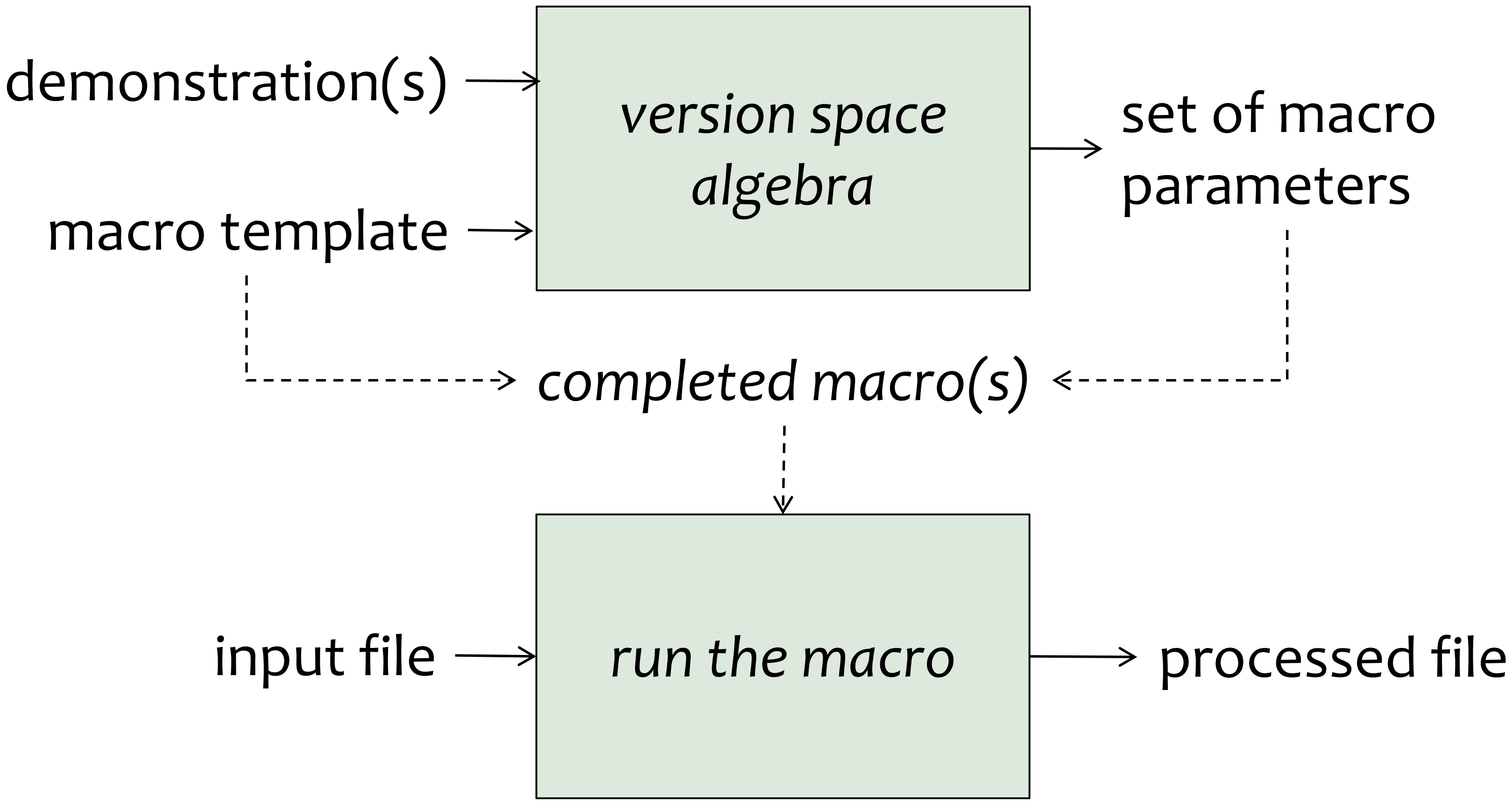
Version space expressed in SKETCH (almost):

```
#define location {| wordOffset(??) | rowCol(??,??)
                  | prefix("??")   |  … |}

repeat ?? times {
  switch(??) {
  0:   move(location)
  1:   insert({| "??" | indent(??,"??") |}))
  2:   cut()
  3:   copy()
       …
  }
}
```

Version Space for SMARTedit



SMARTedit*


demonstration(s)
macro template
version space algebra
set of macro parameters
completed macro(s)
input file
run the macro
processed file

Prospector

[Mandelin, Bodik, Kimelman 2005]

Software reuse: the reality

Using Eclipse 2.1, parse a Java file into an AST

```
IFile file = …
ICompilationUnit cu = JavaCore.createCompilationUnitFrom(file);
ASTNode node = AST.parseCompilationUnit(cu, false);
```

Productivity < 1 LOC/hour   Why so low?

1. follow expected design? two levels of file handlers
2. class member browsers? two unknown classes used
3. grep for ASTNode? parser returns subclass of ASTNode

Prospector

Problem:

APIs have 100K methods. How to code with the API?

Solution:

Observation 1: many reuse problems can be described with a **have-one-want-one query** $q=(h,w)$, where h,w are static types, eg ASTNode.

Observation 2: most queries can be answered with a **jungloid**, a chain of single-parameter "calls". Multi-parameter calls can be decomposed into jungloids.

Synthesizer:

Jungloid is a path in a directed graph of types+methods.

Observation 3: shortest path more likely the desired one

Integrating synthesis with IDEs

- How do we present jungloid synthesis to programmers?
- Integrate with IDE "code completion"

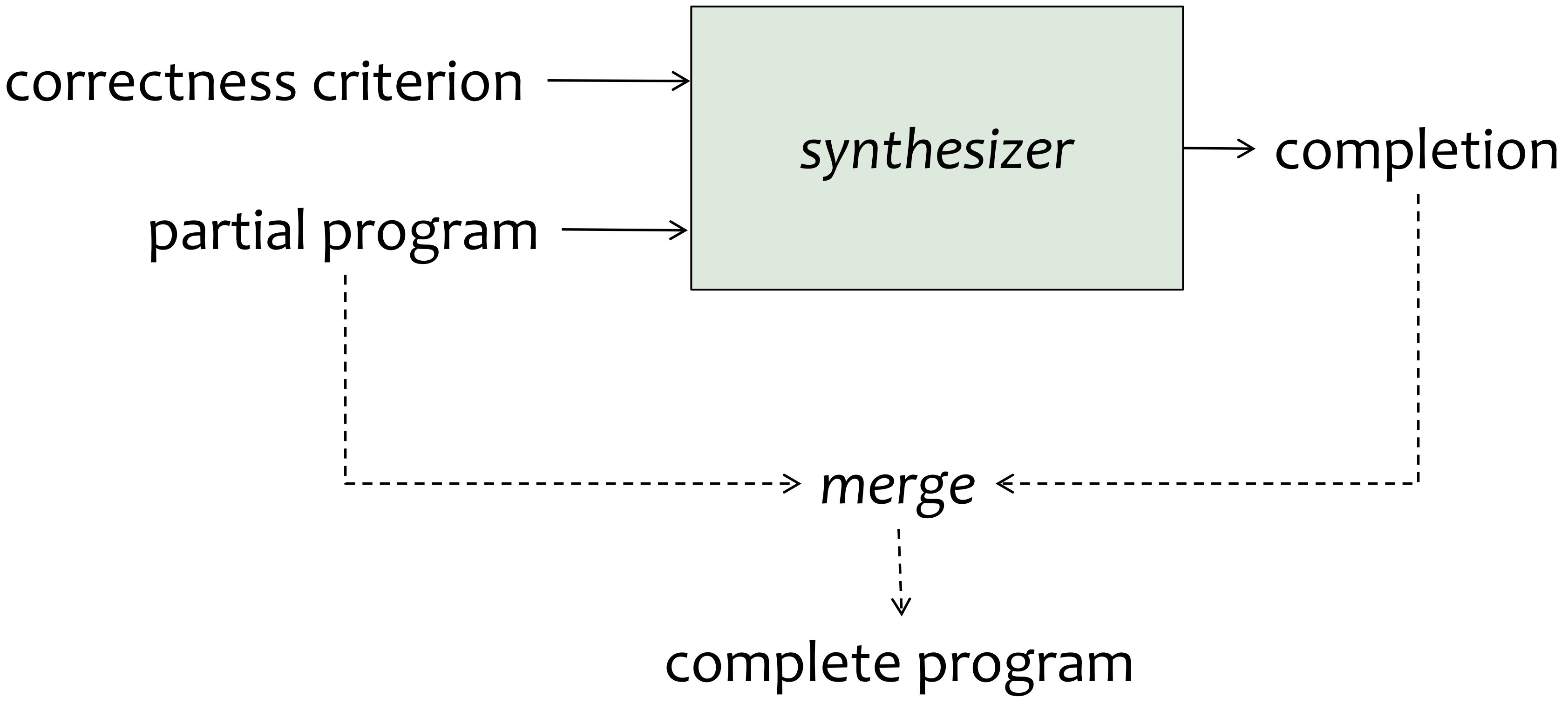
```
IEditorPart editor;

public void parse(IFile file) {
    ASTNode ast =
```

have types

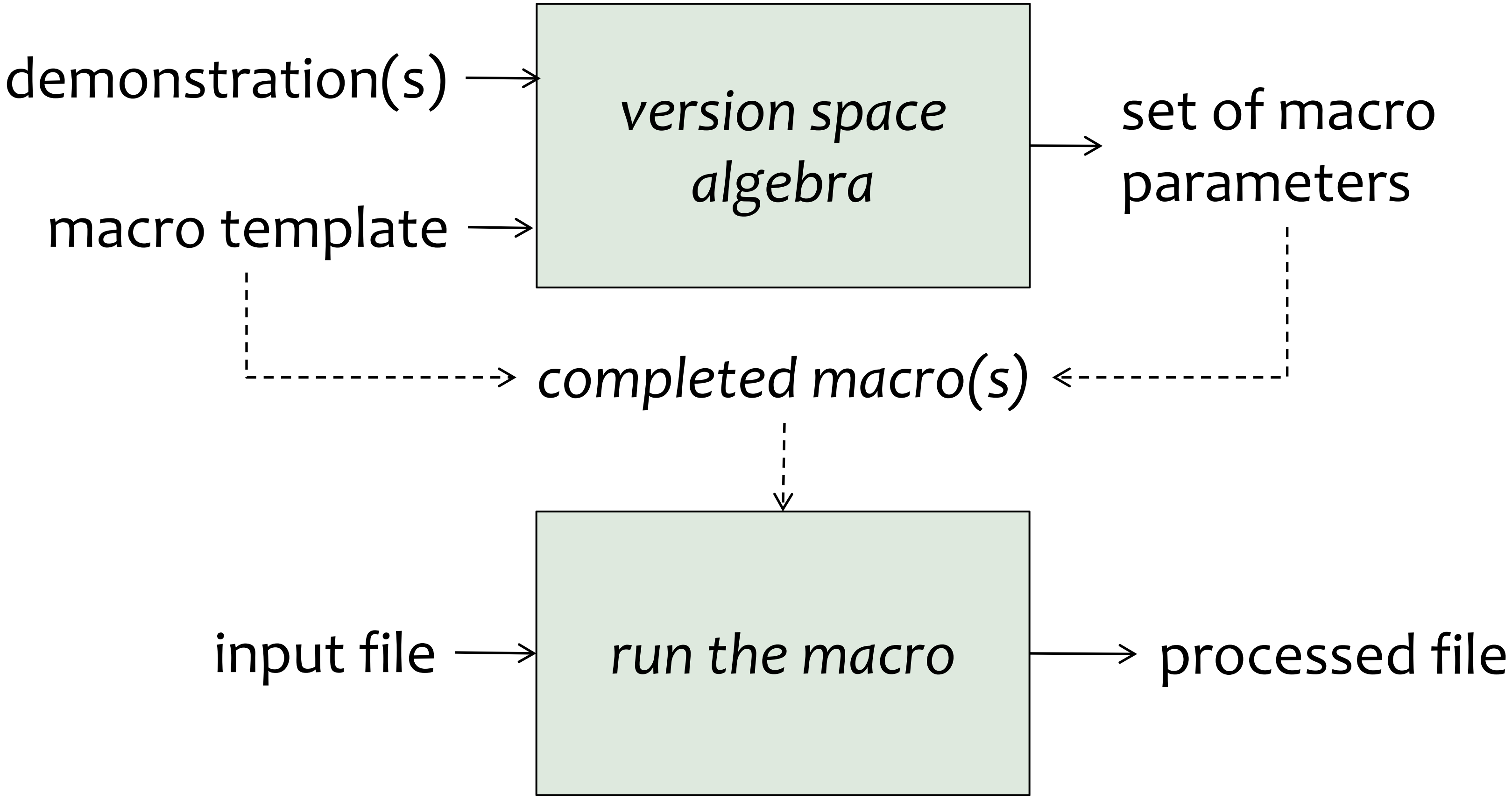
want type

Queries: (IFile, ASTNode)
(IEditorPart, ASTNode)

Are these two also about partial programs?

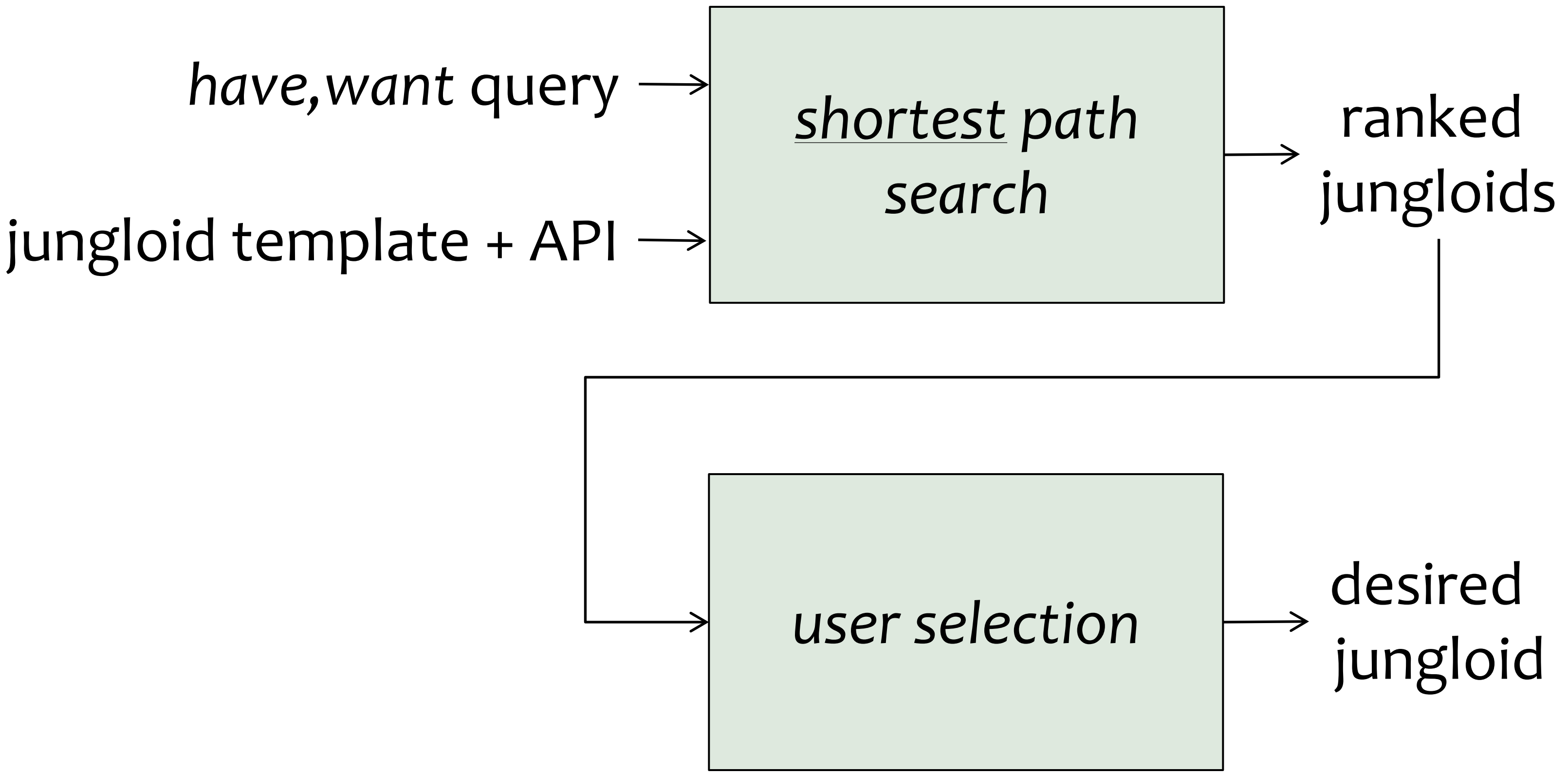


SMARTedit*



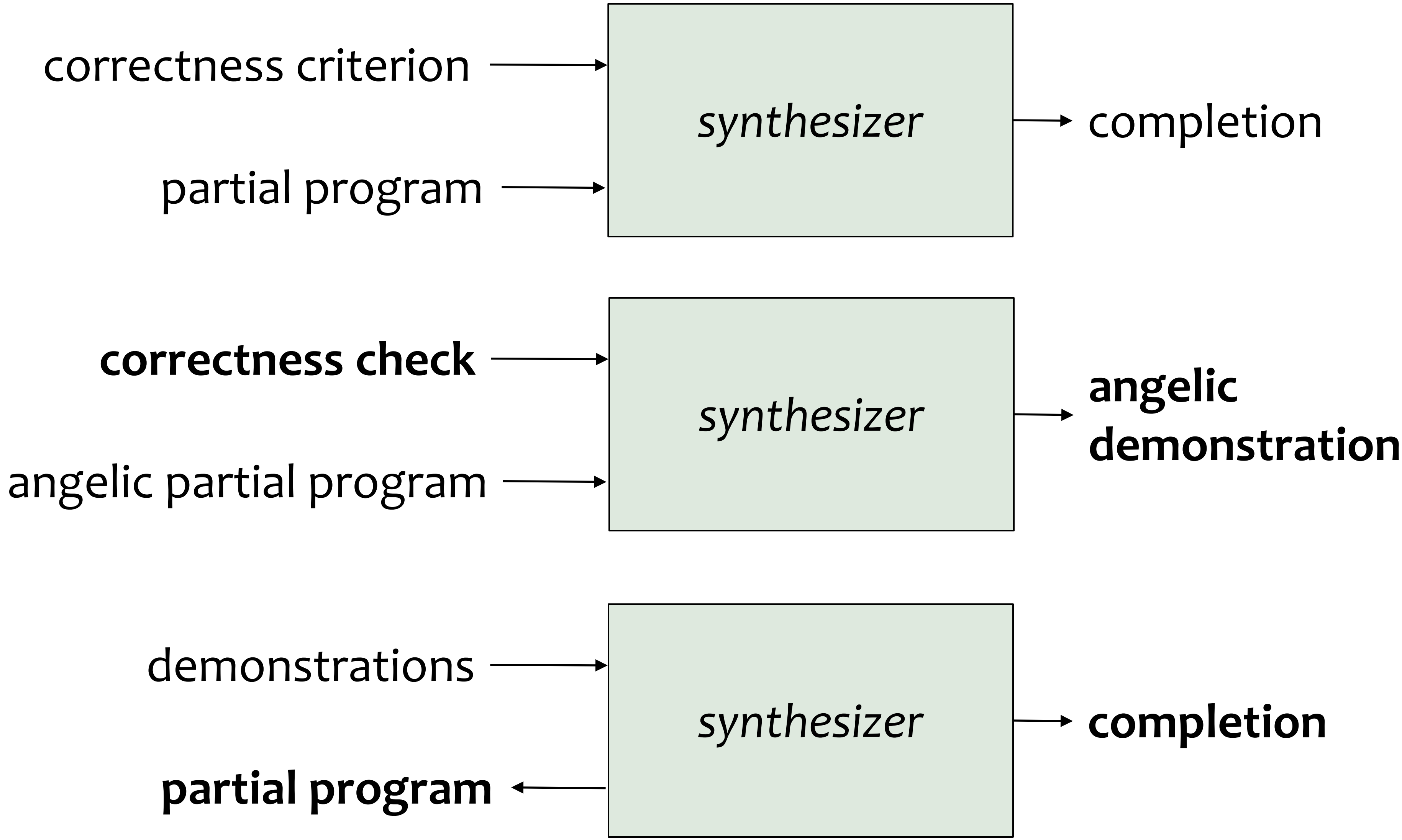
demonstration(s)
macro template
version space algebra
set of macro parameters
completed macro(s)
input file
run the macro
processed file

Prospector



have,want query
jungloid template + API
shortest path search
ranked jungloids
user selection
desired jungloid

Turn partial synthesis around?



Synthesis with partial programs

Partial programs can communicate programmer insight

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

Suitable synthesis algorithm completes the mechanics.

End-user programming, API-level coding are also decomposable into partial program and completion.

Acknowledgements

UC Berkeley

Gilad Arnold
Shaon Barman
Prof. Ras Bodik
Prof. Bob Brayton
Joel Galenson
Sagar Jain
Chris Jones
Evan Pu
Casey Rodarmor
Prof. Koushik Sen
Prof. Sanjit Seshia
Lexin Shan
Saurabh Srivastava
Liviu Tancau
Nicholas Tung

MIT

Prof. Armando Solar-Lezama
Rishabh Singh
Kuat Yesenov
Jean Yung
Zhiley Xu

IBM

Satish Chandra
Kemal Ebcioglu
Rodric Rabbah
Vijay Saraswat
Vivek Sarkar