Using Program Synthesis for Program Analysis

Cristina David [1] Daniel Kroening [1] Matt Lewis [1,2]

[1]University of Oxford [2]Improbable

Outline

What is this talk about?

$$\exists F.\forall x.\sigma(F, x)$$

Solving this formula.

$\exists F.\forall x.\sigma(F,x)$ for Program Analysis

```
assume(Init);
while(G) {
  T;
}
assert(A);
```

$\exists F.\forall x.\sigma(F,x)$ for Program Analysis

```
assume(Init);
while(G) {
  T;
}
assert(A);
```

$$\exists I.\forall x, x'.\mathit{Init}(x) \rightarrow I(x) \wedge$$
$$I(x) \wedge G(x) \wedge T(x,x') \rightarrow I(x') \wedge$$
$$I(x) \wedge \neg G(x) \rightarrow A(x)$$

$\exists F.\forall x.\sigma(F,x)$ for Program Analysis

```
assume(Init);
while(G) {
  T;
}
assert(A);
```

$$\exists N, x_0.\forall x.\exists x'.\mathit{Init}(x_0) \wedge N(x_0) \wedge$$
$$N(x) \rightarrow G(x) \wedge$$
$$N(x) \rightarrow T(x,x') \wedge N(x')$$

$\exists F.\forall x.\sigma(F,x)$ for Program Analysis

```
assume(Init);
while(G) {
  T;
}
assert(A);
```

$$\exists N, x_0.\forall x.\exists x'.\mathit{Init}(x_0) \wedge N(x_0) \wedge N(x) \rightarrow G(x) \wedge N(x) \rightarrow T(x,x') \wedge N(x')$$

$$\exists N, S, x_0.\forall x.\mathit{Init}(x_0) \wedge N(x_0) \wedge N(x) \rightarrow G(x) \wedge N(x) \rightarrow T(x,S(x)) \wedge N(S(x))$$

Program Analysis Problems

- Safety proving (41 lines)
- Bug finding (102 lines)
- Termination (70 lines)
- Non-termination (55 lines)
- Complexity analysis (56 lines)
- Refactoring
- Concurrency
- Heap
- ...

All of these analyses are precise (no false alarms, no false negatives)!

$\exists F.\forall x.\sigma(F, x)$ is Program Synthesis

$$\underbrace{\exists F}_{\text{There is a function...}} \;.\; \underbrace{\forall x}_{\text{which for all inputs...}} \;.\; \underbrace{\sigma(F, x)}_{\text{satisfies some spec}}$$

Example

$$\exists F.\forall x \in \mathbb{N}.F(x) > x$$

Example

$$\exists F.\forall x \in \mathbb{N}.F(x) > x \qquad F(x) = x + 1$$

$$F = \{(0,1),(1,2),\ldots\}$$

Example

$$\exists F.\forall x \in \mathbb{N}.F(x) > x \qquad F(x) = x + 1$$
$$F = \{(0,1),(1,2),\ldots\}$$

Instead of directly dealing with functions, we search for *programs* that compute the functions in our solution.

```
int F(int x){
  return x + 1;
}
```

What Does a Specification Look Like?

$$\exists P.\forall x, y.(x \geq y) \Rightarrow P(x, y) = x \wedge$$
$$(x < y) \Rightarrow P(x, y) = y$$

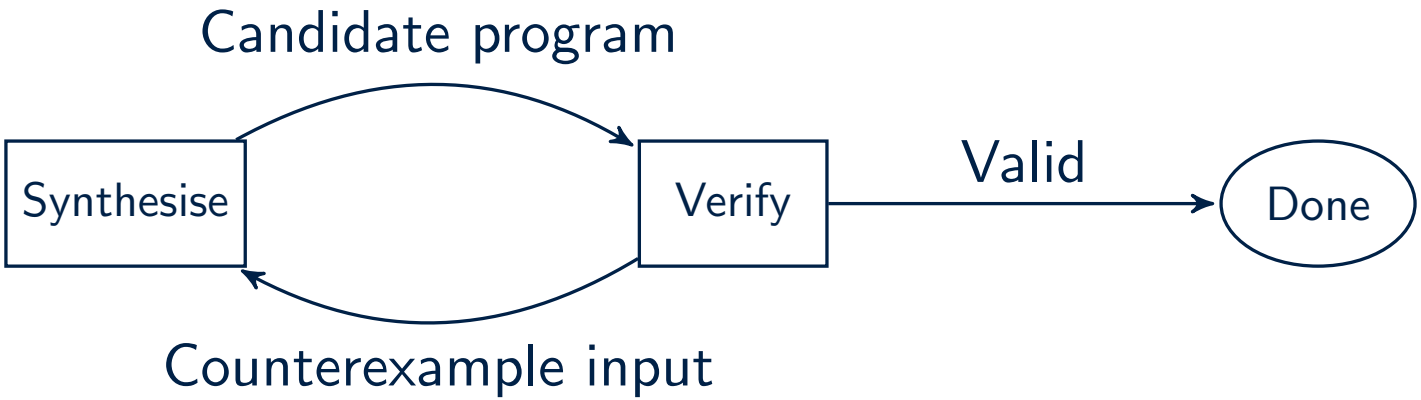
What Does a Specification Look Like?

$$\exists P.\forall x, y.(x \geq y) \Rightarrow P(x, y) = x \wedge$$
$$(x < y) \Rightarrow P(x, y) = y$$

```
bool check(prog_t *p,
           int x,
           int y) {
  int z = exec(p, x, y);

  if (x >= y) {
    return z == x;
  } else {
    return z == y;
  }
}
```

CounterExample Guided Inductive Synthesis



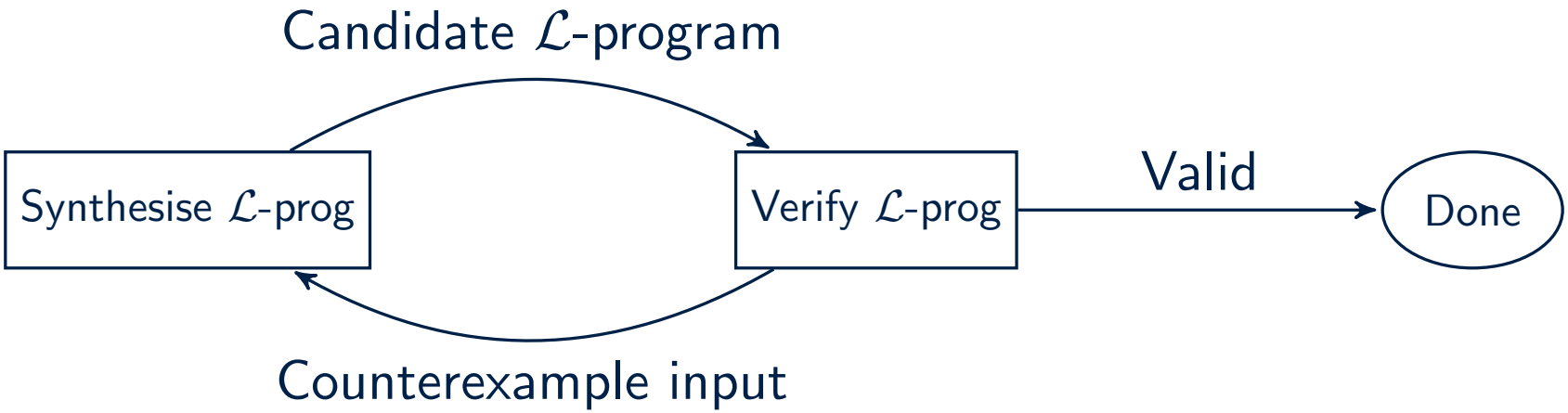
Synthesise: $\exists P.\forall x \in \text{INPUTS}.\sigma(P, x)$

Verify: $\exists x.\neg\sigma(P, x)$

Our instantiation of CEGIS (1)

- We synthesise *finite state programs*.
- Arithmetic is bitvector accurate.
- Is NEXPTIME-complete.

Our instantiation of CEGIS (2)



- $\mathcal{L}$: an SSA language.
- An $\mathcal{L}$-machine, which interprets an $\mathcal{L}$-program.
- A C specification (supplied by the user) which checks that an $\mathcal{L}$-program was "correct" on a particular input.

The $\mathcal{L}$ Language

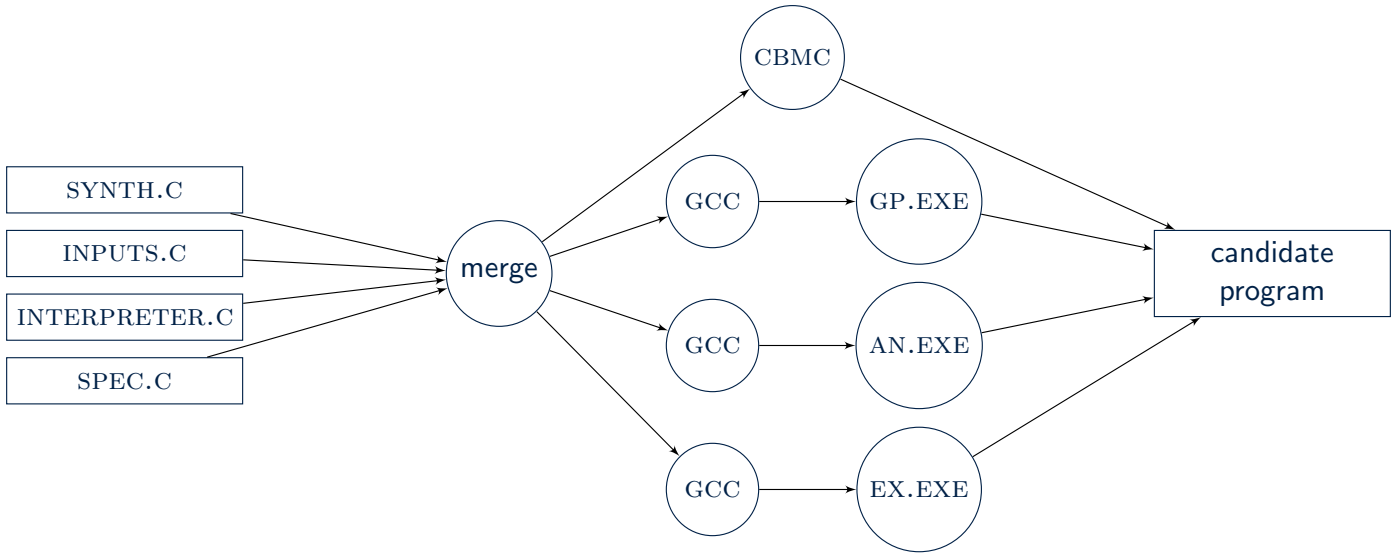
```
t0 = LT x y
t1 = ADD x 10
t2 = MUL x y
t3 = ITE t0 t1 t2
RETURN t3
```

```
int f(int x, int y) {
  if (x < y) {
    return x + 10;
  } else {
    return x * y;
  }
}
```

Properties of $\mathcal{L}$

- $\mathcal{L}$ is *finite-universal*: every function over finite domains is computed by at least one $\mathcal{L}$-program.
- $\mathcal{L}$ is *concise*: for any other language $\mathcal{L}'$, there is at least one function f for which the shortest $\mathcal{L}'$-program computing f is at least as long as the shortest $\mathcal{L}$-program computing f.
- Checking safety of an $\mathcal{L}$-program is NP-complete.

Generating candidate $\mathcal{L}$-programs



We use four strategies in parallel:

- Bounded model checking (with CBMC).
- Simulated annealing.
- Genetic programming.
- Explicit enumeration.

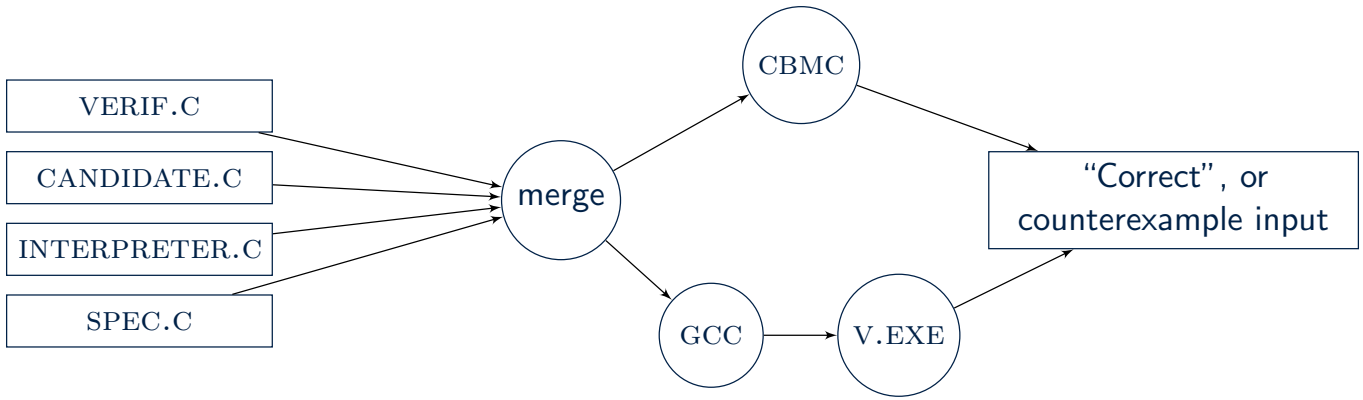
Searching the program space

The space of $\mathcal{L}$-programs is very large.

In order to search it efficiently, we parametrise the space in various dimensions (program length, word width, number of constants) and systematically increase these parameters to cover the whole space.

An important strategy is *generalisation* – we find simple solutions that solve a restricted case of the specification, then try to generalise to a full solution.

Verifying candidate $\mathcal{L}$-programs



We use two strategies in parallel:

- Symbolic execution (with CBMC) – fast for proving a candidate is correct.
- Explicitly enumerating inputs (with a natively compiled binary) – fast for finding counterexamples.

Complexity

Our algorithm is NEXPTIME-complete, but its parameterized complexity is NP[k] where k is the length of the shortest program satisfying the spec (k is bounded by 2^n where n is the number of program variables).

If the spec is UNSAT, the algorithm is very slow (but we have a trick for that).

Example

$$\exists P.\forall x, y.(x \geq y) \Rightarrow P(x, y) = x \wedge$$
$$(x < y) \Rightarrow P(x, y) = y$$

```
bool check(prog_t *p,
           int x,
           int y) {
  int z = exec(p, x, y);

  if (x >= y) {
    return z == x;
  } else {
    return z == y;
  }
}
```

```
t0 = LE x y
t1 = SUB x y
t2 = MUL t1 t0
t3 = SUB x t2
RETURN t3
```

Experimental results

Category	#Benchmarks	#Solved	Avg. solution size	Avg. iterations	Avg. time (s)	Total time (s)
Superoptimisation	29	22	4.1	2.7	7.9	166.1
Termination	47	35	5.7	14.4	11.2	392.9
Safety	20	18	8.3	7.1	11.3	203.9
Total	96	75	5.9	9.2	10.3	762.9

Comparison to SyGuS on the Safety category:

	#Benchmarks	#Solved	#TO	#Crashes	Avg. time (s)	Spec. size
Kalashnikov	20	18	2	0	11.3	341
eSolver	20	7	5	8	13.6	3140
CVC4	20	5	13	2	61.7	3140

Conclusions

Existential second order logic is useful for program analysis.

Existential second order logic can be solved with program synthesis.

Thank you! Questions?

The synthesis algorithm

Algorithm 1 Abstract refinement algorithm

```
1:  function SYNTH(inputs)
2:      (i_1, ..., i_N) ← inputs
3:      query ← ∃F.σ(i_1, P) ∧
    ... ∧ σ(i_N, P)
4:      result ← decide(query)
5:      if result.satisfiable then
6:          return result.model
7:      else
8:          return UNSAT

9:  function VERIF(P)
10:     query ← ∃x.¬σ(x, P)
11:     result ← decide(query)
12:     if result.satisfiable then
13:         return result.model
14:     else
15:         return valid

16: function REFINEMENT LOOP
17:     inputs ← ∅
18:     loop
19:         candidate ← SYNTH(inputs)
20:         if candidate = UNSAT
    then
21:             return UNSAT
22:         res ← VERIF(candidate)
23:         if res = valid then
24:             return candidate
25:         else
26:             inputs ← inputs ∪
    res
```

Constant Generalisation

$$
\begin{aligned}
\mathcal{BV}(m,m) &\rightarrow \mathcal{BV}(n,n) \\
\mathcal{BV}(m-1,m) &\rightarrow \mathcal{BV}(n-1,n) \\
\mathcal{BV}(m+1,m) &\rightarrow \mathcal{BV}(n+1,n) \\
\mathcal{BV}(x,m) &\rightarrow \mathcal{BV}(x,n) \\
\mathcal{BV}(x,m) &\rightarrow \mathcal{BV}(x,m)\cdot\mathcal{BV}(0,n-m) \\
\mathcal{BV}(x,m) &\rightarrow \underbrace{\mathcal{BV}(x,m)\cdot\ldots\cdot\mathcal{BV}(x,m)}_{\frac{n}{m}\,\text{times}}
\end{aligned}
$$

SYNTH and VERIF programs

```
void synth() {
  prog_t p = nondet();
  int in[N], out[M];

  assume(wellformed(p));

  in = test1;
  exec(p, in, out);
  assume(check(in, out));
  ...
  in = testN;
  exec(p, in, out);
  assume(check(in, out));

  assert(false);
}
```

```
void verif(prog_t p) {
  int in[N] = nondet();
  int out[M];

  exec(p, in, out);
  assert(check(in, out));
}
```

The $\mathcal{L}$ language

Integer arithmetic instructions:

add a b	sub a b	mul a b	div a b
neg a	mod a b	min a b	max a b

Bitwise logical and shift instructions:

and a b	or a b	xor a b
lshr a b	ashr a b	not a

Unsigned and signed comparison instructions:

le a b	lt a b	sle a b
slt a b	eq a b	neq a b

Miscellaneous logical instructions:

implies a b	ite a b c

Floating-point arithmetic:

fadd a b	fsub a b	fmul a b	fdiv a b

A Cool Trick

$$\phi = \begin{pmatrix} \exists N, x_0. \forall x. \exists x'. \mathit{Init}(x_0) \wedge N(x_0) \wedge \\ N(x) \rightarrow G(x) \wedge \\ N(x) \rightarrow T(x, x') \wedge N(x') \end{pmatrix}$$

$$\psi = \begin{pmatrix} \exists R. \forall x, x'. G(x) \rightarrow R(x) > 0 \wedge \\ G(x) \wedge T(x, x') \rightarrow R(x) > R(x') \end{pmatrix}$$

Then...

$$\phi \equiv \neg\psi$$

So...

$\phi \vee \psi$ is always SAT for any T, G, Init.