Dimensions in program synthesis

(Gulwani 2010)

Behavioral constraints
How do you tell the system what the program should do?

Structural constraints
What is the space of programs to explore?

Programming by Example
Rich Functional Specifications

Enumeration
Search strategy
How does the system find the program you want?

Beyond Examples

• There has to be some kind of specification language involved
  ▸ E.g. we already saw the use of first-order logic in the SyGuS project:

  \[
  \begin{align*}
  f(0, 1) &= 1 \\
  f(1, 0) &= 1 \\
  f(1, 1) &= 1 \\
  f(2, 0) &= 2
  \end{align*}
  \]

  With free variables:

  \[
  \begin{align*}
  &x \in f(x, y) \\
  &y \in f(x, y) \\
  &f(x, y) = x \lor f(x, y) = y
  \end{align*}
  \]

  represents arbitrary
  some examples

Beyond Examples

• Rich Specifications
  ▸ Reference implementation
  ▸ Assertions
  ▸ Pre- and post-conditions ("contracts")
  ▸ Refinement types
Reference Implementation

- Problem: isolate the least significant zero bit in a word
  - 0010 1011 → 0000 0100 0110 0111
  - 0000 0010 → 0000 0001 1111 1111
  - Easy to implement with a loop

```c
int W = 32; // W = word size
bit[W] isolate0 (bit[W] x) {
  bit[W] ret = 0;
  for (int i = 0; i < W; i++)
    if (x[i] == 0) { ret[i] = 1; return ret; }
}
```

Assertions

- When it’s hard to compute the result but easy to check that it’s valid

```c
split_seconds (int totsec) {
  int h = ...;
  int m = ...;
  int s = ...;
  assert totsec == h*3600 + m*60 + s;
  assert 0 <= h && 0 <= m < 60 && 0 <= s < 60;
}
```

Pre-/post-condition

- When it’s hard to compute the result but easy to express its properties in logic

```c
sort (int[]) in, int n) returns (int[]) out)
  requires n > 0 ∧ len(in) = n
  ensures \( \forall i, j. 0 \leq i < j < n \rightarrow out[i] \leq out[j] \)
  \( \forall i. 0 \leq i < n \rightarrow \exists j. 0 \leq j < n \wedge in[i] = out[j] \)
  \( n \geq 0 \wedge \text{len(in)} = n \)

{P} c {Q}
```

Refinement Types (at a glance)

- Same as pre-/post-conditions, but logic goes inside the types

```c
data SList e where
  Nil :: SList e
  Cons :: h : e : SList e ! SList e
intersect :: xs:SList e + ys:SList e ! (\forall x:SList e | \text{elems xs} \cap \text{elems ys})
```

Why go beyond examples?

- Might need too many
  - Example: Myth needs 12 for insert_sorted, 24 for list_n_th
  - Examples contain too little information
    - Successful tools use domain-specific ranking, e.g. FlashFill
- Output difficult to construct
  - Example: AES cipher, red-black tree
  - Examples also contain too much information (concrete outputs)
- Need strong guarantees
  - Example: one-way hash; aircraft controller
- Reasoning about non-functional properties
  - Example: time/space complexity, security, privacy
Why is this hard?

\[
gcd \ (\text{int} \ a, \ \text{int} \ b) \ \text{returns} \ \text{int} \ c
\]

\[
\begin{align*}
\text{requires} & : \ a \geq 0 \land b \geq 0 \\
\text{ensures} & : \ \forall d \ (a \mid d \land b \mid d) \Rightarrow c \leq d
\end{align*}
\]

\[
\begin{align*}
\text{int} & \ x, \ y \ := \ a, \ b; \\
\text{while} & \ (x \neq y) \\
& \ \{ \\
& \ \text{if} \ (x > y) \ x := \ \cdots; \\
& \ \text{else} \ y := \ \cdots; \\
& \ \}
\end{align*}
\]

\[
\begin{align*}
\text{infinitely many inputs} & \ \text{cannot validate by testing} \\
\text{infinitely many paths!} & \ \text{hard to generate constraints}
\end{align*}
\]

Why is this hard?

Synthesis from examples

Synthesis from specifications

Dimensions in program synthesis

(Gulwani 2010)

Behavioral constraints

Structural constraints

Search strategy

So... for the remainder of the course:

- Constraint-based synthesis
  - How to solve constraints about infinitely many inputs?
  - How to encode semantics of looping / recursive programs?
    - Bounded reasoning
    - Unbounded / deductive reasoning

- Deductive synthesis
  - Enumeration+deduction:
    - How to use deductive reasoning to guide the search?

Constraint-Based Synthesis

- Let’s start easy... from examples
Constraint-Based Synthesis

• Let’s start easy… from examples

\[ E = \{ i \rightarrow o \} \]

\[ \exists C. \forall i, o. \phi(C, i, o) \Rightarrow \psi(i, o) \]

**Synthesis** = \( \exists \forall \)

† Let’s try a simple program sketch.

```java
harness
void main(float x) {
    int y := ?? \ast x + ??;
    assert y - 1 == x + x;
}
```

encoding

\[ \exists C. \forall C, y. y = c_1 \times x + c_2 \Rightarrow y - 1 = x + x \]

```
simplify
```

How do we find \( X \) in general?

Counterexample-Guided Inductive Synthesis (CEGIS)

\[ \exists c. \forall x. Q(c, x) \]

**Idea 1**: Bounded Observation Hypothesis

• Assume that there exists a small set

\[ X = \{ x_1, x_2, \ldots, x_n \} \]

such that

\[ \bigwedge Q(c, x) \Rightarrow \forall x. Q(c, x) \]

Bounded Observation Example

\[ \exists C, x. \forall c. \bigwedge_1 x + c_2 - 1 = x + x \]

• This is a linear constraint.

• Two inputs are enough;

• Furthermore: any two inputs!

\[ X = \{ 0, 1 \} \]

How do we find \( X \) in general?
CEGIS Loop

\[ Q(\mathbf{0}, \mathbf{0}, x) \]

Oracle

\[ (\text{for validity of } \forall \text{ formulas}) \]

Counterexample

\[ x_1 = 0 \quad x_2 = 1 \]

Bag of instances \( \{x\} \)

Solver

Bag of instances \( \{x\} \)

CEGIS in Sketch

- Reference implementation:

```java
int W = 32;
bit[W] isolate0(bit[W] x) {
    bit[W] ret = 0;
    for (int i = 0; i < W; i++)
        if (x[i] == 0) { ret[i] = 1; return ret; }
}
```

- Program with holes (say: parametric program space):

```java
bit[W] being_smart(bit[W] x) implements isolate0 {
    return ~x + ??) & (x + ??);
}
```

Lab #9

- Use SKETCH to synthesize a reverse function that works on singly-linked lists.

```
\[ v_1 = v_2 \]
\[ v_1 = v_2.next \]
```

- Use SKETCH to synthesize a reverse function that works on singly-linked lists.

```
\[ \text{while } \cdots \text{ do } \]
\[ v_1 = v_2 \]
\[ v_1 = v_2.next \]
```

Lab #9

- How do we write the specification (= assertions)?

```
\[ \forall l, \text{is_sorted } l (<) \rightarrow \text{is_sorted } (\text{reverse } l) (>) \]
```

- But here is a rather simple and elegant solution:

```
\text{TMTOWTDI!}
```

- Remember from Coq class? 😱

```
\forall l, \text{is_sorted } l (<) \rightarrow \text{is_sorted } (\text{reverse } l) (>)
```

- This will be our specification!