Synthesis-Enabled Translation

CSE 501
Spring 15
Announcements

• Office hour today 3-4, CSE 530

• Project presentations on Thursday
  – 10 min presentation for each group
  – 2 min for questions

• Project final report and HW 2 due on June 9th
Outline for today

• Synthesis background
• Using synthesis to build compilers

• Two domain studies
  – Database applications
  – Stencils
What is synthesis
The promise

Automate the task of writing programs
What do we mean by synthesis

• We want to get code from high-level specs
  – Python and VB are pretty high level, why is that not synthesis?

• Support compositional and incremental specs
  – Python and VB don’t have this property
    • If I don’t like the way the python compiler/runtime is implementing my program, I am out of luck.
  – Logical specifications do
    • I can always add additional properties that my system can satisfy
  – Specs are not only functional
    • Structural specifications play a big role in synthesis
    • How is my algorithm going to look like.
The fundamental challenge

- The fundamental challenge of synthesis is dealing with an uncooperative environment
  - For reactive systems, people model this as a game
    - For every move of the adversary (ever action of the environment), the synthesized program must make a counter-move that keeps the system working correctly.
    - The game can be modeled with an automata
The fundamental challenge

- The fundamental challenge of synthesis is dealing with an uncooperative environment
  - If we are synthesizing functions, the environment provides the inputs
  - i.e. whatever we synthesize must work correctly for all inputs

- This is modeled with a doubly quantified constraint
  - if the spec is given as pre and post conditions, then:
    - $\exists P \forall \sigma. (\sigma \models \{\text{pre}\}) \Rightarrow (\sigma \models \text{WP}(P, \{\text{post}\})$

- But what does it mean to quantify over the space of programs??
Quantifying over programs

• Synthesis in the functional setting can be seen as curve fitting
  – i.e. we want to find a curve that satisfies some properties

• It’s very hard to do curve fitting when you have to consider arbitrary curves
  – Instead, people use parameterized families of curves
  – This means you quantify over parameters instead of over functions

• This is the first fundamental idea in software synthesis
  – People call these Sketches, scaffolds, templates, ...
  – They are all the same thing
Formalizing the synthesis problem

- $\exists P. \forall \sigma. (\sigma \models \{\text{pre}\}) \Rightarrow (\sigma \models WP(P, \{\text{post}\})$  
- $\exists P. \forall \text{in}. P(\text{in}) \models \phi$  
  - $\phi$ represents the specification  
- $\exists c. \forall \text{in}. Sk(c, \text{in}) \models \phi$  
- $\exists c. \forall \text{in}. Q(c, \text{in})$  

- Many ways to represent $Q$  
  - Can model as a boolean predicate at the abstract level
Dealing with quantifiers

• Eliminate symbolically
  – You can use an abstract domain
  – You can use plain-vanilla elimination
    (not recommended)

• Sample the space of inputs intelligently
Solving the synthesis problem

- Deductive synthesis
  - Write rules to describe all possible derivations from spec to actual program
  - Provably correct since only semantic-preserving programs are explored
  - Requires axiomatization of domain and complete spec from user
  - Example: Denali
Solving the synthesis problem

• Inductive synthesis
  – User gives examples of input / output of P
    • Essentially a *partial* specification
  – Requires no axioms
  – Search can take significant amount of time
Inductive synthesis: example

Define parameterized programs explicitly
   – Think of the parameterized programs as “programs with holes”

Example: Hello World of Sketching

spec:

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

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

Sketch: Integer Hole
Solving inductive synthesis

This is known as **CEGIS**
(Counter-Example Guided Inductive Synthesis)
CEGIS in Detail

**Synthesize**

- $Q(c, \text{in}_0)$
- $Q(c, \text{in}_1)$
- $Q(c, \text{in}_2)$
- $Q(c, \text{in}_3)$

**Check**

- $\neg Q(c, \text{in}_0)$
- $\neg Q(c, \text{in}_1)$
- $\neg Q(c, \text{in}_2)$
- $\neg Q(c, \text{in}_3)$

Add to existing $\text{in}_i$
Synthesizing function bodies

• Model each possible function using minterms
• Choose among candidates using multiplexers
• Example:

```c
int c = ??;
if (c == 0) return foo();
else if (c == 1) return bar();
else if (c == 2) return baz();
else error;
```

• Can now use CEGIS as before to find value of ??
What does any of this have to do with compilers?
Recall from last lecture

• Source and target languages have well-specified semantics
  – Otherwise we don’t know what we are doing

• We need to do two things:
  – *Find* code written in target language to convert source into
  – *Verify* that the found fragment is correct, i.e., semantic-preserving
Recall from last lecture

- Traditional compilers solve this using semantic-preserving transformation passes
  - Or so you hope

- Superoptimizers solve this using targeted search
  - Treat source code as specification
  - Still need to axiomatize possible transforms
Recall from last lecture

- Insight 1: given a target code fragment, we can check whether it satisfies spec or not
  - At least semi-automatically, cf. HW2
- Insight 2: we can generate candidate source-target code fragments and use verifier to check its validity
  - This is now an inductive synthesis problem!
  - We search for both target code and proof
Recall from last lecture

• Issue 1: searching for target code fragments given concrete syntax is very expensive
  – Translate from x86 assembly to SPARC

• Issue 2: Hoare-style verification requires finding loop invariants
  – Problem is undecidable in general
Recall from last lecture

- Issue 1: searching for target code fragments given concrete syntax is very expensive
  - We first “lift” source code to a higher level representation before searching

- Issue 2: Hoare-style verification requires finding loop invariants
  - We only need to find invariants that is “strong enough” to validate the postconditions
Synthesis-Enabled Translation

Original source

Code Analyzer
  Alias
  Dataflow

Code Fragment Identifier

Rewrite Searcher
  VC Computation
  Postcondition Synthesizer
  Formal Verifier

Kernel translator

Rewrite Searcher
  VC Computation
  Postcondition Synthesizer
  Formal Verifier

Target code
Kernel Translator #1: Java to SQL

Proof of Equivalence

SET

Java  Proof of Equivalence  SQL
Kernel Translator #1: Java to SQL
Java to SQL

List getUsersWithRoles () {
    List users = User.getAllUsers();
    List roles = Role.getAllRoles();
    List results = new ArrayList();
    for (User u : users) {
        for (Role r : roles) {
            if (u.roleId == r.id)
                results.add(u);
        }
    }
    return results;
}

convert to

List getUsersWithRoles () {
    return executeQuery(
        "SELECT * FROM user u, role r
         WHERE u.roleId == r.id
         ORDER BY u.roleId, r.id";
    }
}
Java to SQL

List getUsersWithRoles () {
  List users = User.getAllUsers();
  List roles = Role.getAllRoles();
  List results = new ArrayList();
  for (User u : users) {
    for (Role r : roles) {
      if (u.roleId == r.id)
        results.add(u); }}
  return results; }

precondition →
outerInvariant(users/query(...), results/[], ...)
outerInvariant(...) ∧ outer loop terminates →
results = outputExpr(users, roles) ...

Verification conditions

outerInvariant(users, roles, u, results, ...)
innerInvariant(users, roles, u, r, results, ...)
results = outputExpr(users, roles)
Expressing invariants

- Theory of Ordered Relations (TOR)
- Similar to relational algebra
- Model relations as ordered lists

L ::= program var
  | []
  | L : L | L : e
  | top_e(L)
  | L ⊖_f L | σ_f(L)
  | π_f(L) | order_e(L)

e ::= L[i]
  | e op e
  | max(L) | min(L)
  | sum(L) | avg(L)
  | size(L)
Java to SQL

List getUsersWithRoles () {
    List users = User.getAllUsers();
    List roles = Role.getAllRoles();
    List results = new ArrayList();
    for (User u : users) {
        for (Role r : roles) {
            if (u.roleId == r.id)
                results.add(u);
        }
    }
    return results; }

precondition \rightarrow
outerInvariant(users/query(...), results/[], ...)
outerInvariant(...) \land outer loop terminates \rightarrow
results = outputExpr(users, roles) ...

results = users \bowtie_{roleId = id} roles

outerInvariant(users, roles, u, results, ...)
results_{j+1} = results_j :
users[i] \bowtie_{roleId = id} roles [0..j]
Java to SQL

Original source

Program + Unknown Postcondition + Unknown Invariants

Code Analyzer

Alias
Dataflow

Code Fragment Identifier

Sketch
C-like language with holes and assertions

VC Generator

Unroll Inline Enumerate

Rewrite Searcher

VC Computation
Postcondition Synthesizer
Formal Verifier

Target code

Kernel translator
Join Query

- Nested-loop join $\rightarrow$ Hash join!
- $O(n^2) \rightarrow O(n)$
Kernel Translator #2: Fortran to Halide

Legacy Fortran/C++ Code

Proof of Equivalence

Stencil DSL (Halide)

SET
Legacy Fortran to Halide

```plaintext
for (k=y_min-2;k<=y_max+2;k++) {
    for (j=x_min-2;j<=x_max+2;j++) {
        post_vol[((x_max+5)*(k-(y_min-2))+(j)-(x_min-2))] = volume[((x_max+4)*(k-(y_min-2))+(j)-(x_min-2))] + vol_flux_y[((x_max+4)*(k+1 -(y_min-2))+(j)-(x_min-2))] - vol_flux_y[((x_max+4)*(k-(y_min-2))+(j) - (x_min-2))];
    }
}
```

Postcondition:

\[ post\_vol[j,k] = volume[j,k] + vol\_flux[j,k+1] + vol\_flux[j,k] \]
Expressing invariants

\[ \forall (i, j) \in \text{Dom} . A[i, j] = \text{expr}\left( \{ B_n[\text{expr}(i, j), \text{expr}(i, j)] \} \right) \]

```java
out = 0;
for (int i = 0; i < n - 1; ++i){
    out[i+1] = in[i];
}
```

- Big invariants
- Complex floating point arithmetic
- Universal Quantifiers
**Example**

\[
\begin{align*}
& \text{out} = 0 \\
& \text{for(} \text{int } i=0; \ i\leq n-1; \ ++i)\{ \\
& \quad \text{out}[i+1] = \text{in}[i]; \\
& \}
\end{align*}
\]

\[
\forall i, n, \text{out, in, idx} \quad 0 \leq i \leq n-1 \\
\forall j \in [1, i] \quad \text{out}[j] = \text{in}[j-1] \land i \geq n-1 \quad \rightarrow \quad \text{out}[\text{idx}] = \begin{cases} \\
\text{in}[\text{idx}] & \text{idx} \in [1,n) \\
0 & \text{otherwise}
\end{cases}
\]

- Loop invariant
- \(\neg\text{loopCond}\)
- \(\text{out}=\text{expr}\)
Example

∀ i, n, out, in, idx

\[
\forall \ i, n, out, in, idx \quad \bigwedge_{j \in \{idx\} \cap [1,i]} out[j] = in[j - 1] \\
\bigwedge_{j \in \{idx\} \cap [1,i]} out[j] = 0
\]

\[\land \quad i \geq n - 1 \quad \rightarrow \quad out[idx] = \begin{cases} 
\text{in}[idx] & \text{idx } \in [1,n) \\
0 & \text{otherwise}
\end{cases}\]

out = 0

for(int i=0; i<n-1; ++i){
    out[i+1] = in[i];
}

\neg\text{loopCond}

out=\text{expr}

Loop invariant
Synthesis time with parallel synthesis on 24 cores
Speedups on 24 cores
Summary

• Automatic translation from source to target language is hard
• Use synthesis to bridge the gap

• Future work:
  – Cost-based translation
  – Language for developers to express invariants
Course Outline

• Static analysis
• Language design
• Program Verification
• Dynamic analysis
• New compilers
  – superoptimizers
  – synthesis-based translation
Other PL classes of interest

- 503: Software Engineering
- 505: Programming Languages
- 507: Computer-Aided Reasoning for Software
- 504: Advanced Topics in Software Systems
- 599: Verifying Software Systems

Thank you for taking this class
Have a great summer!