Synthesis Modulo Recursive Functions

based on a paper of E. Kneuss, V. Kuncak, I. Kuraj, P. Suter

Lukas Convent

January 14, 2016

What we want

Given a specification

```
def split (xs: List): (List, List) =
  choose {(r: (List, List)) =>
    content(xs) == content(r._1) ++ content(r._2)
}
```

... we want a synthesized program as a solution.

What we rely on

- *Leon* **Verifier** which reasons over user-defined recursive functions. Given a formula, the procedure is
  - Complete for counterexamples:
    `size(xs) < 3` $\rightsquigarrow$ *Counterexample* `1::2::3::nil`
  - Complete for proofs for certain classes of formulas:
    `size(1::2::xs) > 1`

What we rely on

- *Leon* **Verifier** which reasons over user-defined recursive functions. Given a formula, the procedure is
  - Complete for counterexamples:
    `size(xs) < 3` $\rightsquigarrow$ *Counterexample* `1::2::3::nil`
  - Complete for proofs for certain classes of formulas:
    `size(1::2::xs) > 1`
- **Deductive Synthesis Framework**

Structure

Deductive Synthesis Framework

We have seen this before:

Synthesis Problem

$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$

- $\bar{a}$ Set of input variables
- ϕ Synthesis predicate
- $\bar{x}$ Set of output variables

Deductive Synthesis Framework

We have seen this before:

Synthesis Problem

$[\![\bar{a} \ \langle\phi\rangle \ \bar{x}]\!]$

- $\bar{a}$ Set of input variables
- ϕ Synthesis predicate
- $\bar{x}$ Set of output variables

Synthesis Problem Solution

$[\![\bar{a} \ \langle\phi\rangle \ \bar{x}]\!] \quad \vdash \quad \langle P \mid \overline{T}\rangle$

- P Precondition
- $\overline{T}$ Program term

Deductive Synthesis Framework

... now we add a **Path condition**:

Synthesis Problem

$[\![\bar{a}\ \langle \Pi \triangleright \phi \rangle\ \bar{x}]\!]$

- $\bar{a}$ Set of input variables
- ϕ Synthesis predicate
- $\bar{x}$ Set of output variables
- Π **Path condition**

Synthesis Problem Solution

$[\![\bar{a}\ \langle \Pi \triangleright \phi \rangle\ \bar{x}]\!] \quad \vdash \quad \langle P \mid \overline{T} \rangle$

- P Precondition
- $\overline{T}$ Program term

Inference rules

We have already seen lots of inference rules:

$$\textsc{One-point}\ \frac{[\![\bar{a}\ \langle \Pi \triangleright \phi[x_0 \mapsto t] \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \text{vars}(t)}{[\![\bar{a}\ \langle \Pi \triangleright x_0 = t \land \phi \rangle\ x_0\,, \bar{x}]\!] \vdash \langle P \mid \mathsf{val}\ \bar{x} := \bar{T};\, (t\,, \bar{x}) \rangle}$$

$$\textsc{Ground}\ \frac{\mathcal{M} \models \phi \qquad \text{vars}(\phi) \cap \bar{a} = \emptyset}{[\![\bar{a}\ \langle \Pi \triangleright \phi \rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \mathcal{M} \rangle}$$

$$\textsc{Case-split}\ \frac{[\![\bar{a}\ \langle \Pi \triangleright \phi_1 \rangle\ \bar{x}]\!] \vdash \langle P_1 \mid \bar{T}_1 \rangle \qquad [\![\bar{a}\ \langle \Pi \land \neg P_1 \triangleright \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2 \rangle}{[\![\bar{a}\ \langle \Pi \triangleright \phi_1 \lor \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_1 \lor P_2 \mid \mathsf{if}(P_1)\ \{\bar{T}_1\}\ \mathsf{else}\ \{\bar{T}_2\} \rangle}$$

...

We will extend our system by **three new** types of **rules**.

1. $\frac{\cdots}{\cdots}$ Recursion Rule
2. $\frac{}{\cdots}$ Symb. Term Expl. Rule
3. $\frac{}{\cdots}$ Condition Abduction Rule

Recursion Rule (1)

1. Typically, a recursive function involves decreasing the argument of an inductive data type (e. g. **List**)

Recursion Rule (1)

1. Typically, a recursive function involves decreasing the argument of an inductive data type (e. g. **List**)
2. We use a generic schema to solve the problem for each case:

```
def rec(a₀, ā) =
  require(Π₂)
  a₀ match {
    case Nil ⇒ T̄₁
    case Cons(h, t) ⇒
      lazy val r̄ = rec(t, ā)
      T̄₂
  }
ensuring (r̄ ⇒ Φ[x̄ → r̄])
```

Recursion Rule (2)

$[\![a_0\,,\bar{a}\ \langle\Pi_1 \rhd \phi\rangle\ \bar{x}]\!]$

Recursion Rule (2)

$$[\![a_0\,, \bar{a}\ \langle \Pi_1 \rhd \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \mathsf{rec}(a_0,\bar{a}) \rangle$$

Recursion Rule (2)

$$\frac{[\![\bar{a}\ \langle \Pi_2 \triangleright \phi[a_0 \mapsto \mathsf{Nil}]\rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}_1 \rangle}{[\![a_0\,,\bar{a}\ \langle \Pi_1 \triangleright \phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \mathsf{rec}(a_0,\bar{a})\rangle}$$

Recursion Rule (2)

$$(\Pi_1 \wedge P) \implies \Pi_2$$

$$[\![\bar{a}\ \langle \Pi_2 \rhd \phi[a_0 \mapsto \mathsf{Nil}] \rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}_1 \rangle$$

$$[\![a_0\,, \bar{a}\ \langle \Pi_1 \rhd \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \mathsf{rec}(a_0, \bar{a}) \rangle$$

Recursion Rule (2)

$$(\Pi_1 \wedge P) \implies \Pi_2$$

$$\Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \implies \Pi_2[a_0 \mapsto t]$$

$$[\![\bar{a} \; \langle \Pi_2 \rhd \phi[a_0 \mapsto \mathsf{Nil}] \rangle \; \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}_1 \rangle$$

$$[\![a_0 \,, \bar{a} \; \langle \Pi_1 \rhd \phi \rangle \; \bar{x}]\!] \vdash \langle P \mid \mathsf{rec}(a_0,\bar{a}) \rangle$$

Recursion Rule (2)

$$\frac{\begin{array}{l}(\Pi_1 \wedge P) \implies \Pi_2 \\ \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \implies \Pi_2[a_0 \mapsto t] \\ [\![\bar{a}\ \langle \Pi_2 \rhd \phi[a_0 \mapsto \mathsf{Nil}]\rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}_1 \rangle \\ [\![\bar{r}\,, h\,, t\,, \bar{a}\ \langle \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \wedge \phi[a_0 \mapsto t, \bar{x} \mapsto \bar{r}] \rhd \phi[a_0 \mapsto \mathsf{Cons}(h,t)]\rangle\ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \bar{T}_2 \rangle\end{array}}{[\![a_0\,, \bar{a}\ \langle \Pi_1 \rhd \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \mathsf{rec}(a_0,\bar{a}) \rangle}$$

Symbolic Term Exploration Rule

- **Goal**: Find terms consisting of constructors and function calls

Symbolic Term Exploration Rule

- **Goal**: Find terms consisting of constructors and function calls
- We use generators to look out for correct programs:

```
def genNat(): Nat =
  if (⋆) 0 else 1 + genNat()

def genList(): List =
  if (⋆) Nil else (Cons (genNat(), genList()))
```

Symbolic Term Exploration Rule

- **Goal**: Find terms consisting of constructors and function calls
- We use generators to look out for correct programs:

```
def genNat(): Nat =
  if (⋆) 0 else 1 + genNat()

def genList(): List =
  if (⋆) Nil else (Cons (genNat(), genList()))
```

- **Assumption**: Our generators will generate a solution eventually.

Discovering programs (1)

Given a synthesis problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ and a generator *genX* matching the type of $\bar{x}$...

```
def genX(): X = if (⋆) ... else
                  if (⋆) ... else ...
```

Discovering programs (1)

Given a synthesis problem $[\![\bar{a} \ \langle\phi\rangle \ \bar{x}]\!]$ and a generator *genX* matching the type of $\bar{x}$...

```
def genX(): X = if (⋆) ... else
                  if (⋆) ... else ...
```

1. Unroll the generator function up to an instantiation depth

Discovering programs (1)

Given a synthesis problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ and a generator *genX* matching the type of $\bar{x}$...

```
def genX(): X = if (⋆) ... else
                  if (⋆) ... else ...
```

1. Unroll the generator function up to an instantiation depth
2. Find a model such that ϕ is fulfilled, consisting of:
   (a) $\bar{a}_0$ Some input for which the problem is solved
   (b) $\bar{b}_0$ Determines the choices $(\star)$ of `genX`

Discovering programs (1)

Given a synthesis problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ and a generator *genX* matching the type of $\bar{x}$...

```
def genX(): X = if (⋆) ... else
                  if (⋆) ... else ...
```

1. Unroll the generator function up to an instantiation depth
2. Find a model such that ϕ is fulfilled, consisting of:
   (a) $\bar{a}_0$ Some input for which the problem is solved
   (b) $\bar{b}_0$ Determines the choices $(\star)$ of `genX`
3. If a model is found, try to falsify the program (fixing $\bar{b} := \bar{b}_0$)
   (a) If it can be falsified: Discard
   (b) If it cannot be falsified: Success

Discovering programs (2)

To speed up the process, we can first prune many terms from the generator.

Discovering programs (2)

To speed up the process, we can first prune many terms from the generator.

1. Compile the expression $\phi[\overline{x} \mapsto genX(\overline{b})]$

Discovering programs (2)

To speed up the process, we can first prune many terms from the generator.

1. Compile the expression $\phi[\overline{x} \mapsto genX(\overline{b})]$
2. Rapidly discard many candidate programs

Discovering programs (2)

To speed up the process, we can first prune many terms from the generator.

1. Compile the expression $\phi[\overline{x} \mapsto genX(\overline{b})]$
2. Rapidly discard many candidate programs
3. Build a new pruned generator, then run the procedure from before

Example

$$[\![a_0, a_1 \;\; \langle true \triangleright x = a_0 * a_1 \rangle \;\; x]\!]$$

Example

$$\frac{[\![a_1 \ \langle true \triangleright x = 0 * a_1 \rangle \ x]\!] \quad [\![x' a'_0 a_1 \ \langle true \wedge x' = a'_0 * a_1 \triangleright x = (1 + a'_0) * a_1 \rangle \ x]\!]}{[\![a_0, a_1 \ \langle true \triangleright x = a_0 * a_1 \rangle \ x]\!]}$$

Example

$$\dfrac{\dfrac{}{[\![a_1 \ \langle true \triangleright x = 0\rangle \ \ x]\!]}}{[\![a_1 \ \langle true \triangleright x = 0 * a_1\rangle \ \ x]\!]} \quad [\![x' a_0' a_1 \ \langle true \wedge x' = a_0' * a_1 \triangleright x = (1 + a_0') * a_1\rangle \ \ x]\!]$$

$$\overline{[\![a_0, a_1 \ \langle true \triangleright x = a_0 * a_1\rangle \ \ x]\!]}$$

Example

$$\dfrac{\dfrac{\dfrac{\checkmark}{[\![a_1 \ \langle true \rhd x = 0 \rangle \ x]\!]}}{[\![a_1 \ \langle true \rhd x = 0 * a_1 \rangle \ x]\!]} \quad [\![x' a_0' a_1 \ \langle true \wedge x' = a_0' * a_1 \rhd x = (1 + a_0') * a_1 \rangle \ x]\!]}{[\![a_0, a_1 \ \langle true \rhd x = a_0 * a_1 \rangle \ x]\!]}$$

Example

```
def genX() =
        if (⋆) a₁
  else if (⋆) x'
  else if (⋆) genX() + x'
  else ...
```

✓

$$\llbracket a_1 \ \langle true \triangleright x = 0 \rangle \ \ x \rrbracket$$

$$\llbracket a_1 \ \langle true \triangleright x = 0 * a_1 \rangle \ \ x \rrbracket \qquad \llbracket x' a_0' a_1 \ \langle true \wedge x' = a_0' * a_1 \triangleright x = (1 + a_0') * a_1 \rangle \ \ x \rrbracket$$

$$\llbracket a_0, a_1 \ \langle true \triangleright x = a_0 * a_1 \rangle \ \ x \rrbracket$$

Example

```
def genX() =
      if (⋆) a_1
 else if (⋆) x'
 else if (⋆) genX() + x'        P:= a_1 + x'
 else ...
```

✓

$$\llbracket a_1 \ \langle true \triangleright x = 0 \rangle \ \ x \rrbracket$$

$$\llbracket a_1 \ \langle true \triangleright x = 0 * a_1 \rangle \ \ x \rrbracket \qquad \llbracket x' a_0' a_1 \ \langle true \wedge x' = a_0' * a_1 \triangleright x = (1 + a_0') * a_1 \rangle \ \ x \rrbracket$$

$$\llbracket a_0, a_1 \ \langle true \triangleright x = a_0 * a_1 \rangle \ \ x \rrbracket$$

Example

✓

```
def genX() =
      if (⋆) a_1
  else if (⋆) x'
  else if (⋆) genX() + x'
  else ...
```

P:= a_1 + x'

```
a_1 + x' = (1+a_0') * a_1
a_1 + a_0'*a_1 = (1+a_0') * a_1
```

✓

$$\llbracket a_1 \ \langle true \triangleright x = 0 \rangle \ \ x \rrbracket$$

$$\llbracket a_1 \ \langle true \triangleright x = 0 * a_1 \rangle \ \ x \rrbracket \qquad \llbracket x' a_0' a_1 \ \langle true \wedge x' = a_0' * a_1 \triangleright x = (1 + a_0') * a_1 \rangle \ \ x \rrbracket$$

$$\llbracket a_0, a_1 \ \langle true \triangleright x = a_0 * a_1 \rangle \ \ x \rrbracket$$

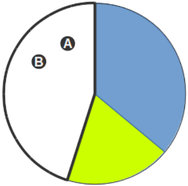
Condition Abduction Rule

- **Goal**: Synthesize recursive function bodies
- Typically, a recursive function consists of a top-level case analysis

Condition Abduction Rule

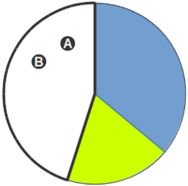
- **Goal**: Synthesize recursive function bodies
- Typically, a recursive function consists of a top-level case analysis
- **Idea**: Pick program terms, then "abduce" their preconditions until the whole input space is covered

Synthesis Procedure




```
if      cond1 then P1
else if cond2 then P2
else ?
```

Synthesis Procedure

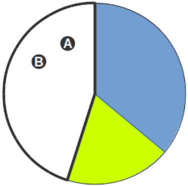



```
if        cond1 then P1
else if   cond2 then P2
else ?
```

Find $\overline{\textbf{P3}}$ which covers most sample models A B

Is $\overline{\textbf{P3}}$ valid for ?

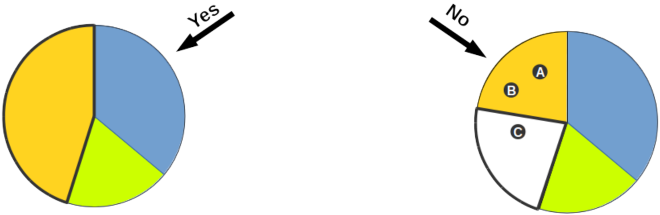
Synthesis Procedure



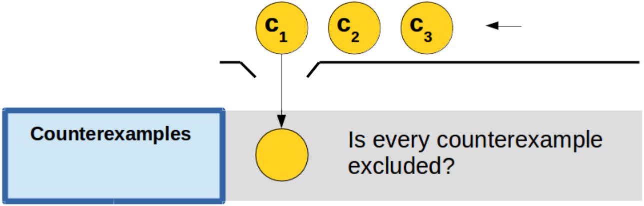

```
if        cond1 then P1
else if   cond2 then P2
else ?
```

Find $\overline{\textbf{P3}}$ which covers most sample models Ⓐ Ⓑ

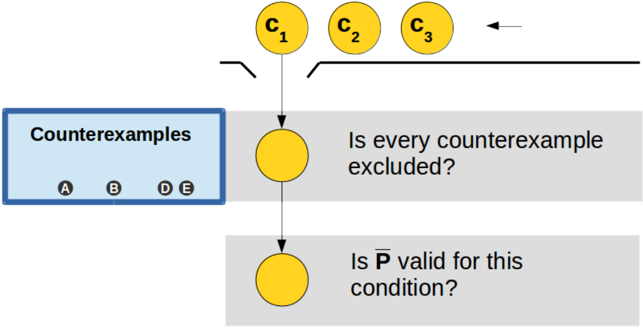
Is $\overline{\textbf{P3}}$ valid for  ?



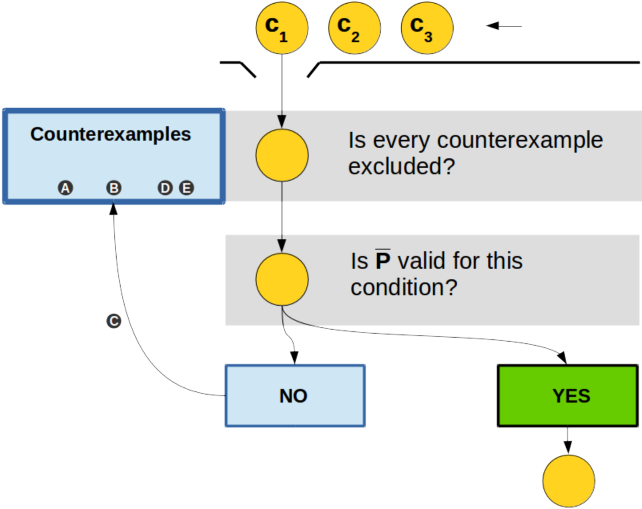
Finding a condition for $\overline{\mathbf{P}}$



Finding a condition for $\overline{\mathbf{P}}$



Finding a condition for $\overline{\mathbf{P}}$



Final assembly

- We have added three new rules which extend the deductive framework
- Synthesis now consists of finding one inference tree for a problem
- At any time, we can stop the synthesis process and use intermediate results

Discussion

- *What are use cases for these techniques, what are their limits?*
- *How can the generator functions best be designed?*
- *How can the search for a synthesis derivation best be organized?*
- *Can ad-hoc interaction during synthesis with the user be considered a strategy?*