Inductive Synthesis

Neil Immerman

www.cs.umass.edu

joint work with
Shachar Itzhaky, Sumit Gulwani, and Mooly Sagiv

Inductive Synthesis: Goal

Write specification, φ, in high level logical language, e.g., SO.

Inductive Synthesis: Goal

Write specification, φ, in high level logical language, e.g., SO.

Synthesize efficient implementation, $\alpha \equiv \varphi$, in target language, T.

Inductive Synthesis: Goal

Write specification, φ, in high level logical language, e.g., SO.

Synthesize efficient implementation, $\alpha \equiv \varphi$, in target language, T.

Analogy: given query, $\varphi \in$ SQL, derive equivalent query, $\alpha \in$ SQL, with better runtime.

Inductive Synthesis: Goal

Write specification, φ, in high level logical language, e.g., SO.

Synthesize efficient implementation, $\alpha \equiv \varphi$, in target language, T.

Analogy: given query, $\varphi \in$ SQL, derive equivalent query, $\alpha \in$ SQL, with better runtime.

Example 1: Given FO equations $C = f(x, y)$ to be maintained given small changes to variables, x, y. Derive finite differencing code in T_0, performing updates to C in constant time.

Inductive Synthesis: Goal

Write specification, φ, in high level logical language, e.g., SO.

Synthesize efficient implementation, $\alpha \equiv \varphi$, in target language, T.

Analogy: given query, $\varphi \in$ SQL, derive equivalent query, $\alpha \in$ SQL, with better runtime.

Example 1: Given FO equations $C = f(x, y)$ to be maintained given small changes to variables, x, y. Derive finite differencing code in T_0, performing updates to C in constant time.

Example 2: Given graph properties, e.g., "is connected", "is a tree", in $\mathrm{FO(TC)}$, derive implementations in T_1 with guaranteed linear runtime.

Descriptive Complexity

$$\begin{array}{ccccc} \textbf{Query} & & & & \textbf{Answer} \\ q_1\ q_2\ \cdots\ q_n & \mapsto & \boxed{\textbf{Computation}} & \mapsto & a_1\ a_2\ \cdots\ a_i\ \cdots\ a_{n^k} \end{array}$$

Descriptive Complexity

$$\begin{array}{c} \textbf{Query} \\ q_1\ q_2\ \cdots\ q_n \end{array} \quad \mapsto \quad \boxed{\textbf{Computation}} \quad \mapsto \quad \begin{array}{c} \textbf{Answer} \\ a_1\ a_2\ \cdots\ a_i\ \cdots\ a_{n^k} \end{array}$$

Restrict attention to the complexity of computing individual bits of the answer, i.e., **decision problems**.

Descriptive Complexity

$$\begin{array}{ccccc} \textbf{Query} & & & & \textbf{Answer} \\ q_1\ q_2\ \cdots\ q_n & \mapsto & \boxed{\textbf{Computation}} & \mapsto & \begin{array}{c} a_1\ a_2\ \cdots\ a_i\ \cdots\ a_{n^k} \\ \cdots\ a\ \cdots \end{array} \end{array}$$

Restrict attention to the complexity of computing individual bits of the answer, i.e., **decision problems**.

How hard is it to **check** if query has property a ?

$$\underset{q_1\ q_2\ \cdots\ q_n}{\textbf{Query}} \quad \mapsto \quad \boxed{\textbf{Computation}} \quad \mapsto \quad \underset{\substack{a_1\ a_2\ \cdots\ a_i\ \cdots\ a_{n^k} \\ \cdots\ a\ \cdots}}{\textbf{Answer}}$$

Restrict attention to the complexity of computing individual bits of the answer, i.e., **decision problems**.

How hard is it to **check** if query has property a ?

How rich a language do we need to **express** property a?

Descriptive Complexity

Query $q_1\ q_2\ \cdots\ q_n$ $\mapsto$ $\boxed{\textbf{Computation}}$ $\mapsto$ **Answer** $a_1\ a_2\ \cdots\ a_i\ \cdots\ a_{n^k}$

$\cdots\ a\ \cdots$

Restrict attention to the complexity of computing individual bits of the answer, i.e., **decision problems**.

How hard is it to **check** if query has property a ?

How rich a language do we need to **express** property a?

There is a constructive isomorphism between these two approaches.

Encode Input via Relations

Graph $G = (\{v_1, \ldots, v_n\}, E, s, t)$

Binary String

$$\mathcal{A}_w = (\{p_1, \ldots, p_8\}, S)$$
$$S = \{p_2, p_5, p_7, p_8\}$$
$$w = 01001011$$

Vocabularies: $\tau_g = (E^2, s, t)$, $\tau_s = (S^1)$

First-Order Logic

input symbols:	from τ
variables:	$x, y, z, \ldots$
boolean connectives:	$\wedge, \vee, \neg$
quantifiers:	$\forall, \exists$
numeric symbols:	$=, \leq, +, \times, \min, max$

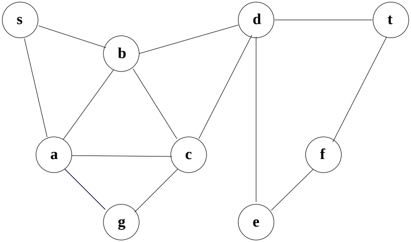
$$\alpha \equiv \forall x \exists y (E(x,y)) \in \mathcal{L}(\tau_g)$$

$$\beta \equiv \exists x \forall y (x \leq y \wedge S(x)) \in \mathcal{L}(\tau_s)$$

$$\beta \equiv S(\min) \in \mathcal{L}(\tau_s)$$

Second-Order Logic

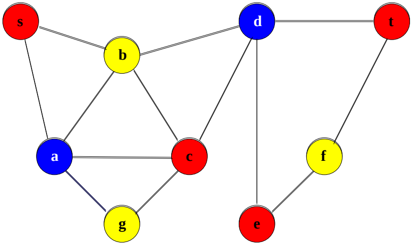
$$
\begin{aligned}
\Phi_{3-\text{color}} \quad \equiv \quad & \exists\, R^1\, Y^1\, B^1\, \forall x\, y\, ((R(x) \vee Y(x) \vee B(x)) \quad \wedge \\
& (E(x,y) \rightarrow (\neg(R(x) \wedge R(y)) \wedge \neg(Y(x) \wedge Y(y)) \\
& \qquad\qquad \wedge \quad \neg(B(x) \wedge B(y)))))
\end{aligned}
$$



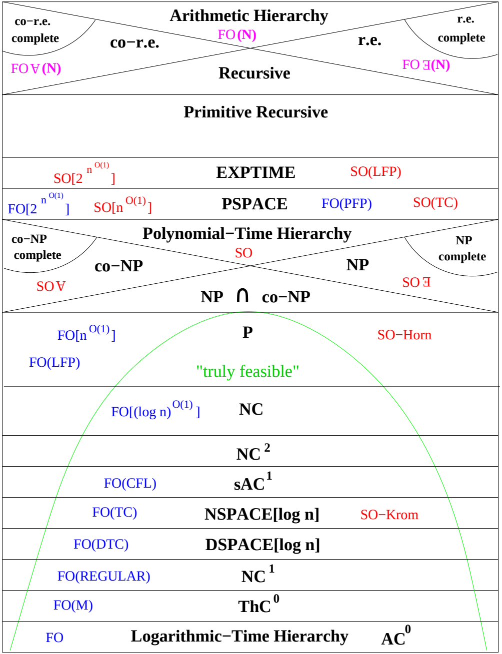
Second-Order Logic

Fagin's Theorem: NP = SO∃

$$\Phi_{3-\text{color}} \equiv \exists R^1\, Y^1\, B^1\, \forall x\, y\, ((R(x) \vee Y(x) \vee B(x)) \wedge$$
$$(E(x,y) \rightarrow (\neg(R(x) \wedge R(y)) \wedge \neg(Y(x) \wedge Y(y))$$
$$\wedge \quad \neg(B(x) \wedge B(y)))))$$



Logic as Specification Language



Model Checking rather than Satisfiabliity

Inductive Synthesis: Goal

Write specification, φ, in high level logical language, e.g., SO.

Synthesize efficient implementation, $\alpha \equiv \varphi$, in target language, T.

Could have a range of target languages: T_0, guaranteed constant runtime, T_1, guaranteed linear runtime, T_2, guaranteed $O(n^2)$ runtime, etc.

Example 1: Given FO equations $C = f(x, y)$ to be maintained given small changes to variables, x, y. Derive finite differencing code in T_0, performing updates to C in constant time.

Example 2: Given graph properties, e.g., "is connected", "is a tree", in $\mathrm{FO(TC)}$, derive implementations in T_1 with guaranteed linear runtime.

Sketch of Method

1. Input: $\varphi \in \mathrm{SO}$; vocabularies: $\sigma \subseteq \sigma'$; target language: L
2. Generate instances $\mathcal{M} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\} \models \varphi$
3. Find minimum size formula $\alpha \in L$ s.t. α covers* $\mathcal{M}$
4. **If** (exists small instance $\mathcal{A} \models \varphi$ not covered by α)

   Then $\mathcal{M} + = \{\mathcal{A}\}$; **Goto** 3.
5. **Return** α

* α covers $\mathcal{M}$ iff $\mathcal{M} \models \alpha$ and
α determines the correct output bits on each $\mathcal{A} \in \mathcal{M}$,

$$\alpha \wedge \Delta_\sigma(\mathcal{A}) \vdash \Delta_{\sigma'}(\mathcal{A})$$

Finite Differencing [Paige]

Maintain $C == f(x_1, \ldots, x_k)$ where $x_i \mathrel{+}= \delta$

Example:

Expression: $C = T + S$

Change: $T \mathrel{+}= \{a\}$

Derived Code: $C \mathrel{+}= \{a\}$

Finite Differencing [Paige]

Maintain $C == f(x_1, \ldots, x_k)$ where $x_i \mathrel{+}= \delta$

Example:

Expression: $C = T + S$

Change: $T \mathrel{+}= \{a\}$

Derived Code: $C \mathrel{+}= \{a\}$

Synthesized Formula:
$$\begin{array}{l} v = a \;\rightarrow\; c'(v) = 1 \\ v \neq a \;\rightarrow\; c'(v) = c(v) \end{array}$$

Expression	Change	Synthesized Derivative	Code
$C = T + S$	$T \mathrel{+}= \{a\}$	$v = a \rightarrow c'(v) = 1$ $v \neq a \rightarrow c'(v) = c(v)$	$C \mathrel{+}= \{a\}$

$$\begin{aligned} \sigma &= (s, t, c, a; \mathrm{suc}, 0, 1, =) \\ \sigma' &= \sigma \cup (t', c') \end{aligned}$$

$$\begin{aligned} B_0(\sigma) &= \left\{\forall v(\ell_1), \forall v(\ell_1 \rightarrow \ell_2) \;\middle|\; \ell_1, \ell_2 \text{ literals}\right\} \\ B_0(\sigma) &= s(a) = 0,\; s(a) = 1 \rightarrow c'(a) = 1, \ldots \end{aligned}$$

$$T_0(\sigma) = \text{conjunctions of base formulas from } B_0(\sigma)$$

Expression	Change	Synthesized Derivative	Code
$C = T + S$	$T \mathrel{+}= \{a\}$	$v = a \rightarrow c'(v) = 1$ $v \neq a \rightarrow c'(v) = c(v)$	$C \mathrel{+}= \{a\}$

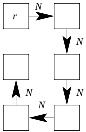
T	S	a	T'	C	C'	$c'(1)$	$c'(2)$	$c'(3)$
$\{1\}$	$\{2\}$	1	$\{1\}$	$\{1,2\}$	$\{1,2\}$	1	1	0
$\{1\}$	$\{2\}$	2	$\{1,2\}$	$\{1,2\}$	$\{1,2\}$	1	1	0
$\{1\}$	$\{2\}$	3	$\{1,3\}$	$\{1,2\}$	$\{1,2,3\}$	1	1	1

$$
\begin{aligned}
v = a &\quad\rightarrow\quad c'(v) = 1 \\
v \neq a &\quad\rightarrow\quad c'(v) = c(v)
\end{aligned}
$$

Expression	Change	Synthesized Derivative	Code
$C = T + S$	$T \mathrel{+}= \{a\}$	$v = a \rightarrow c'(v) = 1$ $v \neq a \rightarrow c'(v) = c(v)$	$C \mathrel{+}= \{a\}$
$C = T + S$	$T \mathrel{-}= \{a\}$	$v \neq a \rightarrow c'(v) = c(v)$ $\neg T(a) \rightarrow c'(a) = 0$ $T(a) \rightarrow c'(a) = c(a)$	**if** $a \notin S : C \mathrel{-}= \{a\}$
$C = T - S$	$T \mathrel{+}= \{a\}$	$v \neq a \rightarrow c'(v) = c(v)$ $\neg S(a) \rightarrow c'(a) = 1$ $S(a) \rightarrow c'(a) = 0$	**if** $a \notin S : C \mathrel{+}= \{a\}$
$C = T - S$	$T \mathrel{-}= \{a\}$	$c(v) = 0 \rightarrow c'(v) = 0$ $v \neq a \rightarrow c'(v) = c(v)$ $v = a \rightarrow c'(a) = 0$	$C \mathrel{-}= \{a\}$

Expression	Change	Synthesized Derivative	Code
$C = f(S)$	$S \mathrel{+}= \{a\}$	$v \neq f(a) \rightarrow c'(v) = c(v)$ $v = a \rightarrow c'(f(a)) = 1$	$C \mathrel{+}= \{f(a)\}$
$C = f^{-1}(S)$	$f(a) = b$	$v \neq a \rightarrow c'(v) = c(v)$ $S(b) \rightarrow c'(a) = 1$ $\neg S(b) \rightarrow c'(a) = 0$	**if** $b \notin S : C \mathrel{-}= \{a\}$ **else** $C \mathrel{+}= \{a\}$
$c_S = \#S$	$S \mathrel{+}= \{a\}$	$S(a) \rightarrow c'_S = c_S$ $\neg S(a) \rightarrow c_S + 1 = c'_S$	**if** $a \notin S : c_S \mathrel{+}= 1$
$c_S = \#S$	$S \mathrel{-}= \{a\}$	$\neg S(a) \rightarrow c'_S = c_S$ $S(a) \rightarrow c'_S + 1 = c_S$	**if** $a \in S : c_S \mathrel{-}= 1$
$c = (\#S == 0)$	$S \mathrel{+}= \{a\}$	$v = a \rightarrow c' = 0$	$c =$ **false**
$c = (\#S == 0)$	$S \mathrel{-}= \{a\}$	$c_S \neq 1 \rightarrow c' = c$ $c'_S = c_S \rightarrow c = c'$ $c'_S = 0 \rightarrow c' = 1$	**if** $a \in S : c_S \mathrel{-}= 1$ $c = (c_S == 0)$

Deriving Graph Classifiers

Name	**Example**	**Input Spec. (+Integrity Const.)**	**Synthesized Formula**
SLL		$1{:}1\ N \wedge$ $\forall u(\neg N^{+}(u,u))$ $\left(\begin{array}{c}\text{root } r \text{ via } N\\ \text{functional } N\end{array}\right)$	$\#p_N(r) = 0 \wedge$ $\forall v(\#p_N(v) \leq 1)$

Abbreviation	Meaning
self-loop-free N	$\forall u(\neg N(u,u))$
root r via N	$\forall u(N^{\star}(r,u))$
functional N	$\forall u,v,x\big(N(x,u) \wedge N(x,v) \to u = v\big)$
1:1 N	$\forall u,v,x\big(N(u,x) \wedge N(v,x) \to u = v\big)$

$$s_e(i) \;=\; \{j \mid i \xrightarrow{e} j\} \qquad\qquad p_e(i) \;=\; \{j \mid j \xrightarrow{e} i\}$$

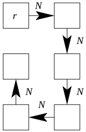
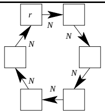
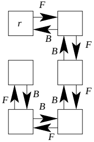
Target Language T_1

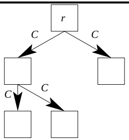
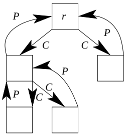
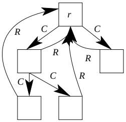
$$R(\sigma) = \{e \mid e \text{ a reg exp over } \sigma\}; \qquad S(\sigma) = \{A \mid A \in \sigma\}$$

$$s_e(i) = \{j \mid i \xrightarrow{e} j\}; \qquad p_e(i) = \{j \mid j \xrightarrow{e} i\}$$

$\langle\text{base}\rangle$	$::=$	$\langle\text{clause}\rangle \to d \mid \langle\text{clause}\rangle \to \neg d \mid$ $\neg\langle\text{clause}\rangle \to d \mid \neg\langle\text{clause}\rangle \to \neg d$	
$\langle\text{clause}\rangle$	$::=$	$\langle\text{atom}\rangle \mid \forall v\ \langle\text{atom}\rangle \mid$ $\forall v\ (v \neq r \to \langle\text{atom}\rangle)$	
$\langle\text{atom}\rangle$	$::=$	$\langle\text{int}\rangle = \langle\text{const}\rangle \mid$ $\langle\text{int}\rangle \leq \langle\text{const}\rangle \mid \langle\text{set}\rangle = \langle\text{set}\rangle$	
$\langle\text{int}\rangle$	$::=$	$\langle\text{const}\rangle \mid \#\langle\text{set}\rangle$	
$\langle\text{const}\rangle$	$::=$	$0 \mid 1$	
$\langle\text{set}\rangle$	$::=$	$\{r\} \mid s_e(r) \mid p_e(r) \mid$ $s_\ell(v) \mid p_\ell(v)$	$e \in R(\sigma)$ $\ell \in S(\sigma)$

Thm: Every element of the language T_1 runs in expected linear time in the worst case.

Name	Example	Input Spec. (+Integrity Const.)	Synthesized Formula
SLL		$1{:}1\ N \wedge$ $\forall u(\neg N^{+}(u,u))$ $\left(\begin{array}{c}\text{root } r \text{ via } N\\ \text{functional } N\end{array}\right)$	$\#p_N(r) = 0 \wedge$ $\forall v(\#p_N(v) \leq 1)$
CYCLE		$\forall u, v(N^{\star}(u,v))$ $\left(\begin{array}{c}\text{root } r \text{ via } N\\ \text{functional } N\end{array}\right)$	$\#p_N(r) = 1$
DLL		$1{:}1\ F \wedge 1{:}1\ B \wedge$ $\forall u, v\big((F(u,v) \leftrightarrow B(v,u))$ $\wedge \neg F^{+}(u,u)\big)$ $\left(\begin{array}{c}\text{root } r \text{ via } F\\ \text{functional } F, B\end{array}\right)$	$\#p_F(r) = 0 \wedge$ $\forall v(s_F(v) = p_B(v))$

Name	Example	Input Spec. (+Integrity Const.)	Synthesized Formula
TREE		1:1 $C \wedge$ $\forall u(\neg C(u,r))$ (root r via C)	$\#p_C(r) = 0 \wedge$ $\forall v(\#p_C(v) \leq 1)$
TREEPP		1:1 $C \wedge$ $\forall u,v\big((C(u,v) \leftrightarrow P(v,u))$ $\wedge \neg C(u,r)\big)$ (root r via C functional P)	$\#s_P(r) = 0 \wedge$ $\forall v(s_P(v) = p_C(v))$
TREERP		1:1 $C \wedge$ $\forall u,v\big(\neg C(u,s) \wedge \neg R(r,u)$ $\wedge (u \neq s \rightarrow R(u,r))\big)$ (root r via C functional R)	$\#p_C(r) = 0 \wedge$ $p_R(r) = s_{C^+}(r) \wedge$ $\forall v(\#p_C(v) \leq 1)$

Example: Testing If a Graph is Bipartite

$$\Phi_{bp} \quad \equiv \quad \exists S^1 \forall xy (E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$$

Example: Testing If a Graph is Bipartite

$$\Phi_{bp} \quad \equiv \quad \exists S^1 \forall xy(E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$$

Maintain Invariant: $\beta \;\equiv\; \forall xy(E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$ incrementally as we add edges to an initially empty graph.

Example: Testing If a Graph is Bipartite

$$\Phi_{bp} \quad \equiv \quad \exists S^1 \forall xy (E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$$

Maintain Invariant: $\beta \; \equiv \; \forall xy (E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$ incrementally as we add edges to an initially empty graph.

Base Case: $G_0 = (V, \emptyset, \emptyset) \models \beta$

Example: Testing If a Graph is Bipartite

$$\Phi_{bp} \quad \equiv \quad \exists S^1 \forall xy (E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$$

Maintain Invariant: $\beta \equiv \forall xy(E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$ incrementally as we add edges to an initially empty graph.

Base Case: $G_0 = (V, \emptyset, \emptyset) \models \beta$

Inductively Assume: $G = (V, E, S) \models \beta$ and add add an edge (a, b): $E' := E \cup \{(a, b)\}$

Example: Testing If a Graph is Bipartite

$$\Phi_{bp} \quad \equiv \quad \exists S^1 \forall xy (E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$$

Maintain Invariant: $\beta \;\equiv\; \forall xy (E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y)))$ incrementally as we add edges to an initially empty graph.

Base Case: $G_0 = (V, \emptyset, \emptyset) \models \beta$

Inductively Assume: $G = (V, E, S) \models \beta$ and add add an edge (a, b): $E' := E \cup \{(a,b)\}$

Case 1: $(V, E', S) \models \beta$ so we're fine.

Case 2: $(V, E', S) \models \neg\beta$

$$(G, a/x, b/y) \models (S(x) \leftrightarrow S(y))$$

Case 2: $(V, E', S) \models \neg\beta$

$$(G, a/x, b/y) \models (S(x) \leftrightarrow S(y))$$

WLOG to reestablish β we must change the value of $S(a)$.

Case 2: $(V, E', S) \models \neg\beta$

$$(G, a/x, b/y) \models (S(x) \leftrightarrow S(y))$$

WLOG to reestablish β we must change the value of $S(a)$.

Naive Incremental Algorithm: If b is in this connected component, report failure
Else: change the value of $S(c)$ for all c in the connected component of a.

Naive Algorithm takes time $O(nm)$.

Better Incremental Algorithm

Keep track of connected component of a in two disjoint parts:

$$S(a) \; = \; C(a) \cap S \qquad\qquad \overline{S}(a) \; = \; C(a) \cap \overline{S}$$

Better Incremental Algorithm

Keep track of connected component of a in two disjoint parts:

$$S(a) \;=\; C(a) \cap S \qquad\qquad \overline{S}(a) \;=\; C(a) \cap \overline{S}$$

1. **if** $(S(a) = S(b))$: **return**("not bipartite")
2. $S(b) := S(b) \cup \overline{S}(a); \quad \overline{S}(b) := \overline{S}(b) \cup S(a)$

Better Incremental Algorithm

Keep track of connected component of a in two disjoint parts:

$$S(a) \;=\; C(a) \cap S \qquad\qquad \overline{S}(a) \;=\; C(a) \cap \overline{S}$$

1. **if** $(S(a) = S(b))$: **return**("not bipartite")
2. $S(b) := S(b) \cup \overline{S}(a); \quad \overline{S}(b) := \overline{S}(b) \cup S(a)$

Finally, revisit case 1 and maintain the data structure:

$S(b) := S(b) \cup S(a); \quad \overline{S}(b) := \overline{S}(b) \cup \overline{S}(a)$

Better Incremental Algorithm

Keep track of connected component of a in two disjoint parts:

$$S(a) \; = \; C(a) \cap S \qquad\qquad \overline{S}(a) \; = \; C(a) \cap \overline{S}$$

1. **if** $(S(a) = S(b))$: **return**("not bipartite")
2. $S(b) := S(b) \cup \overline{S}(a)$; $\quad \overline{S}(b) := \overline{S}(b) \cup S(a)$

Finally, revisit case 1 and maintain the data structure:
$S(b) := S(b) \cup S(a)$; $\quad \overline{S}(b) := \overline{S}(b) \cup \overline{S}(a)$

With the sets $S, \overline{S}$ instantiated using Union/Find, the complexity of this incremental algorithm is then essentially linear, i.e, $O(m)$.

Future Directions $\mathcal{M} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$

- Apply this simple methodology to many more settings.
- Use to program by example.
- Use to automatically take care of the tedious part of programming and maintaining software.
- Use to translate back and forth between different formalisms and levels.
- Use to generate distributed algorithms, reactive systems, incremental algorithms.
- Build in composition to make this approach scale.
- Increase the sophistication of our learning/synthesis approach.