Representation-based Search
Version Space Algebras

CS560: Reasoning About Programs

Roopsha Samanta

Partly based on slides by Armando Solar-Lezama and Xiaokang Qiu

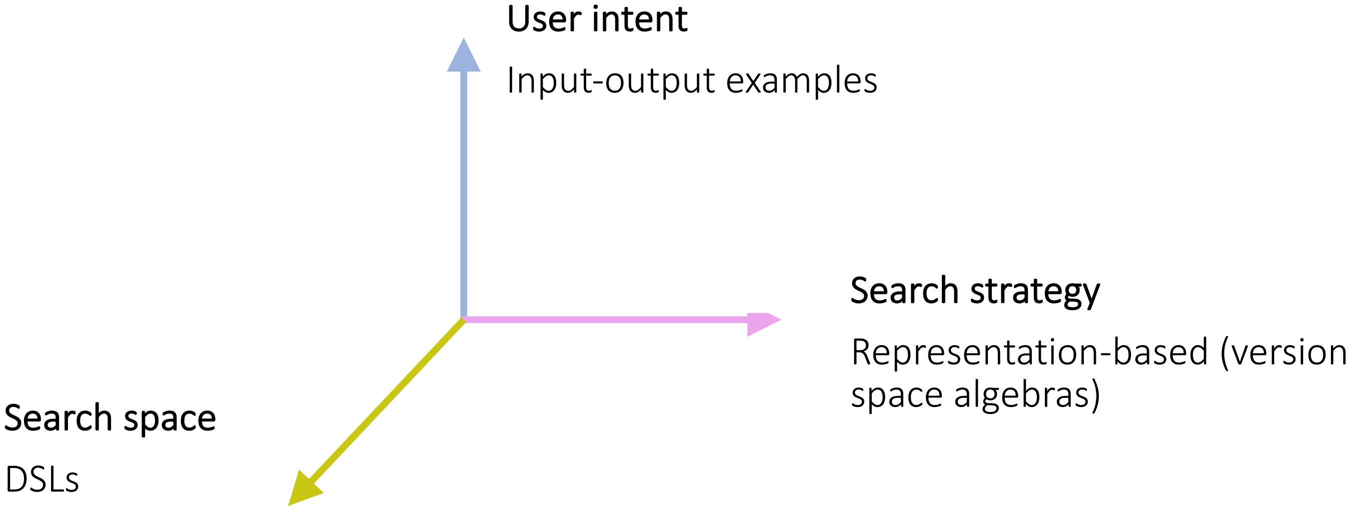
Roadmap

Previously

- Inductive synthesis
- SyGuS
- Enumerative search

Today

- Representation-based search using version space algebras
  (for inductive synthesis)


User intent
Input-output examples
Search strategy
Representation-based (version space algebras)
Search space
DSLs

Quick review: Mathematical Lattice

$(P, \sqsubseteq)$: partially-ordered set (poset) S

$\sqcup$: join, least upper bound (lub)

$\sqcap$: meet, greatest lower bound (glb)

$x \sqcup y$: join of two elements in P

$\sqcup$ S: join over subset S of P

$x \sqcap y$: meet of two elements in P

$\sqcap$ S: meet over subset S of P

(P, $\sqsubseteq$) is a lattice if $\forall x, y.\ (x \sqcup y$ and $x \sqcap y$ exist).

(P, $\sqsubseteq$) is a complete lattice if $\forall S \subseteq P.$ ($\sqcup$ S and $\sqcap$ S exist)

All finite lattices are complete

Example of a lattice that is not complete?

Version spaces

Tom Mitchell

Version space is the set of all hypotheses/functions/programs in a given hypothesis/program space consistent with a set of input-output examples

Tessa Lau

Version space algebra allows us to compose together simple version spaces, using operators such as union ($\cup$), join ($\bowtie$) and transform, in order to construct more complex version spaces

Concept learning using version spaces

Learning Boolean-valued functions

Partial order over hypothesis space H

$h_1 \sqsubseteq \mathrm{h}_2 \colon \forall x.\ h_1(x) \Rightarrow h_2(x)$

Captures generality of hypothesis.

h_2 "better" than h_1

If H is finite, $VS_{H,E}$ is a complete lattice

$\Rightarrow$ least upper bound G and greatest lower bound S exist

$\Rightarrow$ can represent $VS_{H,E}$ as (G, S)

Boundary set representation

~~G~~ is the most general hypothesis consistent with E

S is the most specific hypothesis consistent with E

(most general is the correct phrase)

Concept learning using version spaces

```
G, S := ⊤,⊥;
foreach (x,y) in E
  if y = true
    remove from G any h: h ⊭ (x,y);
    foreach h ∈ S: h ⊭ (x,y)
      remove h from S;
      add to S all minimal generalizations h₁ of h:
      h₁ ⊨ (x,y) and ∃h₂ ∈ G: h₁ ⊏ h₂;
      remove from S any h₁: ∃h₂ ∈ S: h₂ ⊏ h₁;
  if y = false
    ...
```

Specializing G

Generalizing S

If H is partially-ordered and the VS is boundary set-representable, one can represent and search very efficiently!

What if not?

Compose simpler version spaces!

Version Space Algebra: Union and Join

$$VS_{H_1,E} \cup VS_{H_2,E} = VS_{H_1 \cup H_2,E}$$

$$VS_{H_1,E_1} \bowtie VS_{H_2,E_2} = \{\langle h_1, h_2 \rangle \mid h_1 \in VS_{H_1,E_1}, h_2 \in VS_{H_2,E_2}, C(\langle h_1, h_2 \rangle, \langle E_1, E_2 \rangle)\}$$

"Cross-product"

$\langle h_1, h_2 \rangle$ is consistent with $\langle E_1, E_2 \rangle$

Pair

Composition

Application designer assigns interpretation to $\langle h_1, h_2 \rangle$,

Version Space Algebra: Union and Join

$$VS_{H_1,E} \cup VS_{H_2,E} = VS_{H_1 \cup H_2,E}$$

$$VS_{H_1,E_1} \bowtie VS_{H_2,E_2} = \{\langle h_1, h_2\rangle \mid h_1 \in VS_{H_1,E_1}, h_2 \in VS_{H_2,E_2}, C(\langle h_1, h_2\rangle, \langle E_1, E_2\rangle)\}$$

$VS_{H_1,E_1} \bowtie VS_{H_2,E_2}$ is an independent join iff:

$$\forall E_1, E_2, h_1 \in H_1, h_2 \in H_2.\ C(h_1, E_1) \wedge C(h_2, E_2) \Rightarrow C(\langle h_1, h_2\rangle, \langle E_1, E_2\rangle)\}$$

Highly efficient representation!

Flashfill

Poosha Samanta → Samanta

- **Domain-specific language** for string manipulating programs
  - Expressive enough to cover wide range of tasks
  - Restricted enough to enable efficient search
- **Data structure** for representing consistent programs
  - Reuse ideas from version space algebra
  - Start with simple version spaces
  - Define combinators to construct complex version spaces from simple ones
- **Synthesis algorithm** for learning consistent programs in DSL
- **Ranking** consistent programs

[Gulwani 2010]

Flashfill

- **Domain-specific language** for string manipulating programs
  - Expressive enough to cover wide range of tasks
  - Restricted enough to enable efficient search
- **Data structure** for representing consistent programs
  - Reuse ideas from version space algebra
  - Start with simple version spaces
  - Define combinators to construct complex version spaces from simple ones
- **Synthesis algorithm** for learning consistent programs in DSL
- **Ranking** consistent programs

[Gulwani 2010]

Flashfill DSL (Simplified version)

Program: (String x_1, …, String x_k) → String

H := switch((B_1,E_1),...,(B_n,E_n)) | E

String Expression

E := concatenate(A,E) | A

Trace Expression

A := subStr(X,P,P) | constStr(S)

Atomic Expression

P := pos(R_1, R_2) | constPos(K)

Position Expression

R := tokenSeq(T_1,..., T_m) | T | ϵ

Regular Expression

T := C | C+ | ...

Token Expression

A Switch operator evaluates predicates over the input string tuple, and chooses the branch that then produces the output string

Each branch is a concatenation of atomic string expressions, which produce pieces of the output string

Each atomic expression can be a constant, or a substring of one of the input strings

Position in string whose left/right side matches with regular expressions R_1/R_2

Roopsha Samanta Dr.

⊔

α+⊔

α+

Flashfill

- **Domain-specific language** for string manipulating programs
  - Expressive enough to cover wide range of tasks
  - Restricted enough to enable efficient search
- **Data structure** for representing consistent programs
  - Reuse ideas from version space algebra
  - Start with simple version spaces
  - Define combinators to construct complex version spaces from simple ones
- **Synthesis algorithm** for learning consistent programs in DSL
- **Ranking** consistent programs

Version spaces

Enable succinct representation of all consistent programs in memory!

Flashfill

- **Domain-specific language** for string manipulating programs
  - Expressive enough to cover wide range of tasks
  - Restricted enough to enable efficient search
- **Data structure** for representing consistent programs
  - Reuse ideas from version space algebra
  - Start with simple version spaces
  - Define combinators to construct complex version spaces from simple ones
- **Synthesis algorithm** for learning consistent programs in DSL
- **Ranking** consistent programs

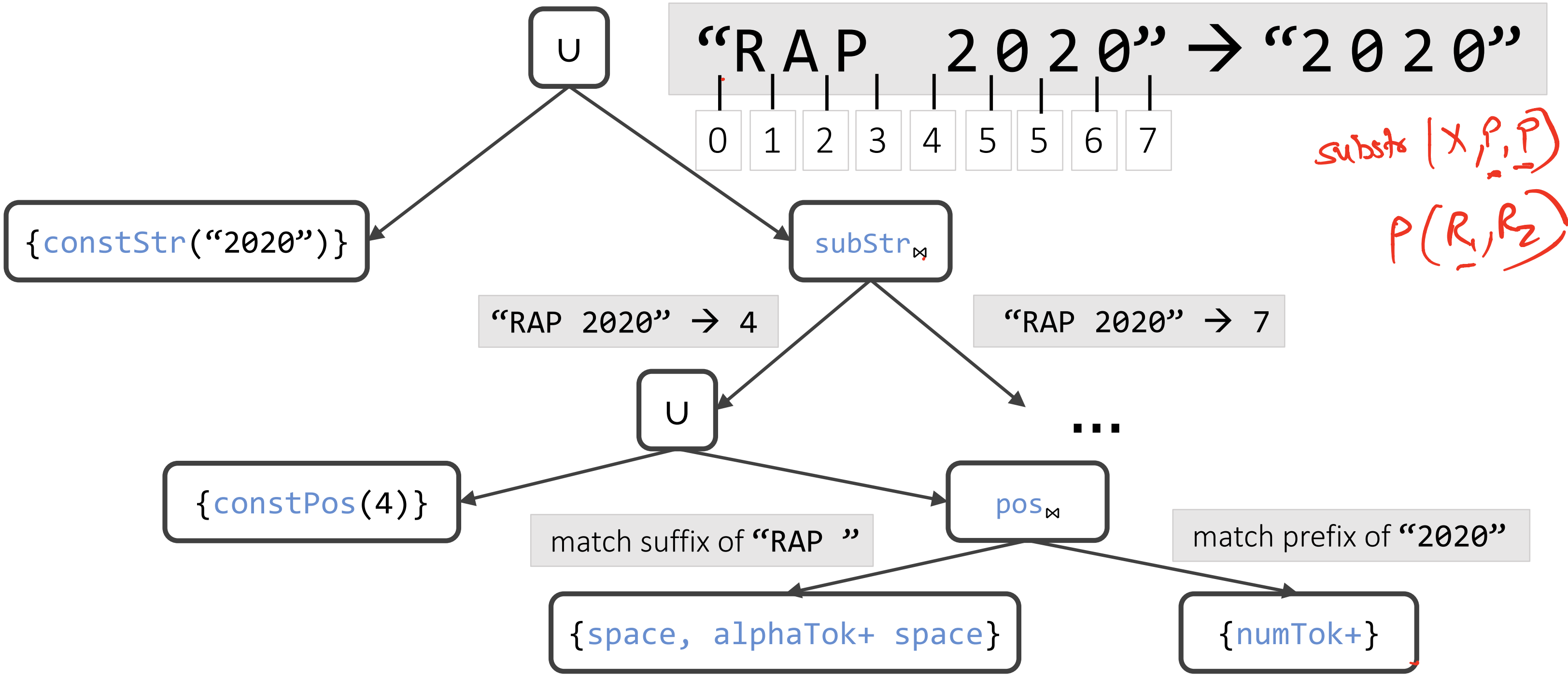
Synthesis algorithm

- For each input-output example,
  compute the set of all trace expressions consistent with example
- Partition examples
- Intersect sets of trace expressions for inputs in the same partition
- Learn conditionals to place inputs in appropriate partition

Partition must satisfy the following properties:

- Intersection of trace sets must be non-empty
- Number of partitions should be as small as possible
- Can learn predicates to classify the partitions

Learning & representing atomic expressions



Flashfill

- **Domain-specific language** for string manipulating programs
  - Expressive enough to cover wide range of tasks
  - Restricted enough to enable efficient search
- **Data structure** for representing consistent programs
  - Reuse ideas from version space algebra
  - Start with simple version spaces
  - Define combinators to construct complex version spaces from simple ones
- **Synthesis algorithm** for learning consistent programs in DSL
- **Ranking** consistent programs

Ranking

- Prefer shorter programs.
  - Fewer number of conditionals.
  - Shorter substring expression, regular expressions.
- Prefer programs with fewer constants.

Strategies

- **Baseline**: Pick any minimal sized program using minimal number of constants.
- **Manual:** Break conflicts using a weighted score of program features.
- **Machine Learning:** Weights are learned from training data.

Summary

Today

- Representation-based search using version space algebras (for inductive synthesis)

Finite tree automata

Next

- Constraint-based search