Abstractions and small languages in synthesis

CS294: Program Synthesis for Everyone

Ras Bodik
Emina Torlak

Division of Computer Science
University of California, Berkeley

Today

Today: we describe why high-level or domain-specific programming abstractions, provided as language constructs, make synthesis more efficient and easier to use.

Next lecture: Student presentations (problem stmt).

Subsequent lecture: Language implementation Part I. Racket macros. Language embedding.

Instructions for classroom presentation

Topic: problem statement (refinement of HW1)

Elaborate on synthesis artifacts:

- what will be synthesized
- what are the specs (this item is important!)

3-minutes per student/team ==> practice!

Email Ras .ppt(x) slides by 9am before lecture.

Outline

Review of HW2

- description of staff solution

Lessons from HW2

- motivate synthesis at high level of abstraction

Reducing the candidate space (tree rotation)

- prune with domain constrains

Reducing the formula size (graph classifiers)

- synthesis followed by code generation

Synthesis at functional level (time permitting)

- followed by data structure generation

Advanced challenge

Is this lecture familiar material? Entertain yourself by designing a small language *L* that can

- express distributed protocols and
- can model message loss and reordering

How to translate programs in *L* to formulas, or otherwise find *L* programs that meet a spec.

Oh yes, when you are done, what is a good spec language for distributed protocols?

HW2 feedback

Description of solutions

We sped up the encoding by

- using smallest bit vectors possible for each variable
- not relying on the extensional theory of arrays
- eliminating redundant constraints on 2-bit variables represented as bit vectors of length 2
- eliminating constant arrays
- replacing macros with explicit let statements; and
- telling the solver which logic the encoding is in.

Lessons (encoding)

why using bitvectors helps

- bounded by the type ==> can save some explicit constraints on values of bitvector variables
- different decision procedure (eg blasting to SAT)

why must also drop Ints?

- absence of Ints allows bitblasting because no need to reason about (infinite) ints
- essentially, a different algorithm is used

why not relying on extensional theory helps

- (= a b) insists that entire arrays a,b are equal, which could be infinitely many if indexes are Ints
- a[0]=b[0] … insists only on bounded number of equalities ==> enumerate what needs to hold

Lessons (the input constraint for ind. synth.)

one perfect input vs. identify sufficient inputs

- Def: perfect ==> correct on a perfect input implies correct on all inputs
- a good input accelerates solving

careful about selecting the perfect input

- we were wrong in Part 2
- Q: how to overcome the danger of weak input?

Results (z3)

	description	Emina’s laptop (sec)	Ras’s laptop (sec)
xpose3-QF_AUFLIA.smt2	xpose3 encoding using the extensional theory of arrays and theory of integers	168	95
xpose3-QF_AUFBV.smt2	xpose3 encoding using the non-extensional theory of arrays and theory of bitvectors; this is a straightforward modification of xpose3-QF_AUFLIA.smt2	148	90
xpose3-QF_AUFBV.smt1	xpose3 encoding using the extensional theory of arrays and theory of integers; this is an optimization of xpose3-QF_AUFBV.smt2, with no array constants, with no function macros, and with an explicit specification of the logic being used	27	15
xpose2-QF_AUFBV.smt2	xpose2 encoding that is a straightforward extension of xpose3-QF_AUFBV.smt2; the key difference is the introduction of additional variables and the use of larger bitvectors to account for the new input matrix	>3600	>3600
xpose2-QF_AUFBV.smt1	xpose2 encoding that is a straightforward extension of xpose3-QF_AUFBV.smt1; the key difference is the introduction of additional variables and the use of larger bitvectors to account for the new input matrix	108	58

Results (Kodkod)

	description	SAT solver	Emina’s laptop (sec)
xpose3-unary	xpose3 hand-crafted encoding, using a unary representation of numbers	MiniSat	6
xpose3-binary	xpose3 encoding generated by Rosette, using a binary representation of numbers	MiniSat	23
		MIniSat with a carefully chosen random seed	1
xpose2-unary	xpose2 hand-crafted unary encoding, which is a straightforward extension of xpose3-unary	MiniSat	89
		Lingeling	9

Wish list from HW2

Wish list:

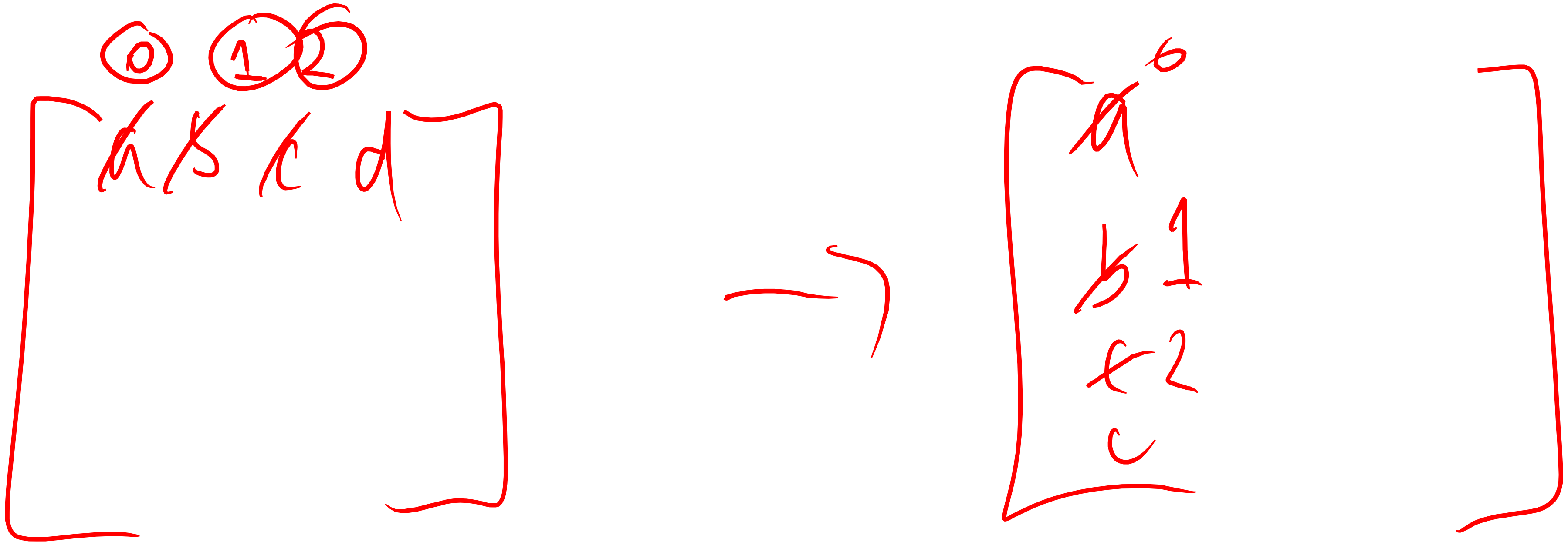
- start the solver earlier
- start the homework earlier

- specify the logic

- use faster solvers
- get feedback on where the solver is wasting time

- debug encoding on 2x2 matrix, then scale up
- facilitate easier tweaking of constraints

- unit tests
- list of ideas of what can have impact

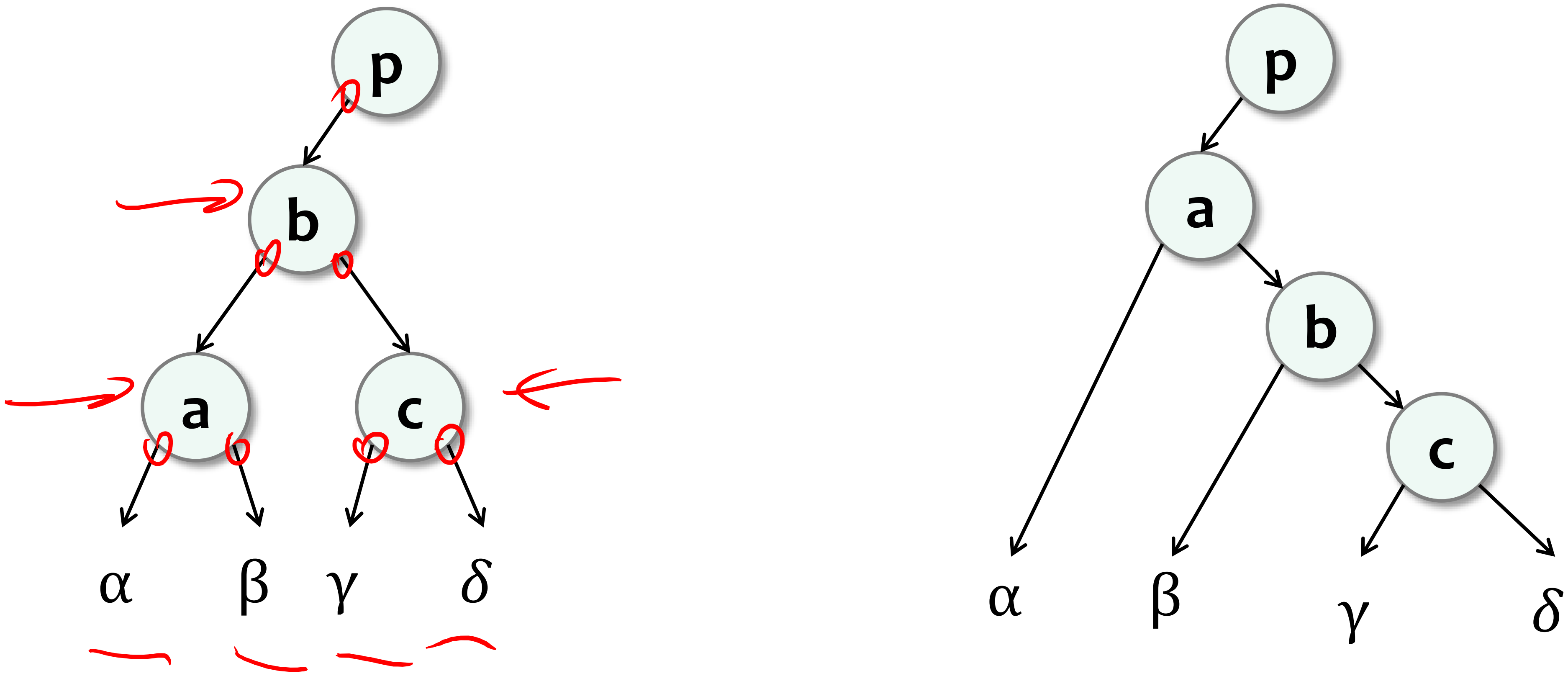


$\exists \vec{h}.\ \phi(\vec{h}, 17)$

Reducing the Size of Candidate Space

Example: Synthesis of tree rotation

We want to suitably rotate tree A into tree B.



We don’t know exactly how to rotate.

So we ask the synthesizer.

Partial program for rotation

We have to update (up to) 7 memory locations.

We have seven pointer values available.

A straightforward partial program:

```
r.left  := {| ~~p~~ | a | b | c | α | β | γ | δ |}
a.left  := {| ~~p~~ | a | b | c | α | β | γ | δ |}
…
c.right := {| ~~p~~ | a | b | c | α | β | γ | δ |}
```

Search space: 7^7, about 10^{17} if you have 3 such sketches

Reducing the search space

Encode that the pointer rotation is a permutation.

```
(p.left, a.left, …, c.right) :=
  synth_permutation(p, a, b, c, α, β, γ, δ)
```

Search space: $7! < 7^7$

Implementing the permutation construct

```
def synth_permutation(lst):
    retval = empty list
    chosen = empty set
    repeat len(lst) times
        ix = ??(0..len(lst)-1)
        append lst[ix] to retval
        assert ix not in chosen
        add ix to chosen
    return retval
```

How many choices exist for len(lst) = 7? 7^7

so does using the permutation reduce search space to 7! ?

Locally ruled out choices

In synth_permutation, selecting ix that has been chosen is immediately ruled out by the assertion

We call this **locally** ruled out choice.

- there are 7!, not 7^7 , choices that satisfy the assertion

Compare this with a **globally** ruled out choice

- such a choice fails only after the solver propagates its effects to assertions in the postcondition.

Further space reduction

In addition to a permutation, we insist that the reordered nodes form a binary search tree

```
(p.left, a.left, …, c.right) :=
  synth_permutation(p, a, b, c, α, β, γ, δ)

assert bst_to_depth_4(p)

def bst_to_depth_4(p):
    assert p.d >= p.left.d
       …
       and p.d <= p.right.right.right.d
```

How is this a small language?

What do permutation, bst_to_depth_4 have to do with abstractions or languages?

These are constructs of a tree manipulation language

We defined them inside the host language

- ie, they are embedded in the host

- and compiled to formulas

Summary

Effective size of candidate space $\neq 2^{\text{bits of holes}}$

Because local assertions prune the search space

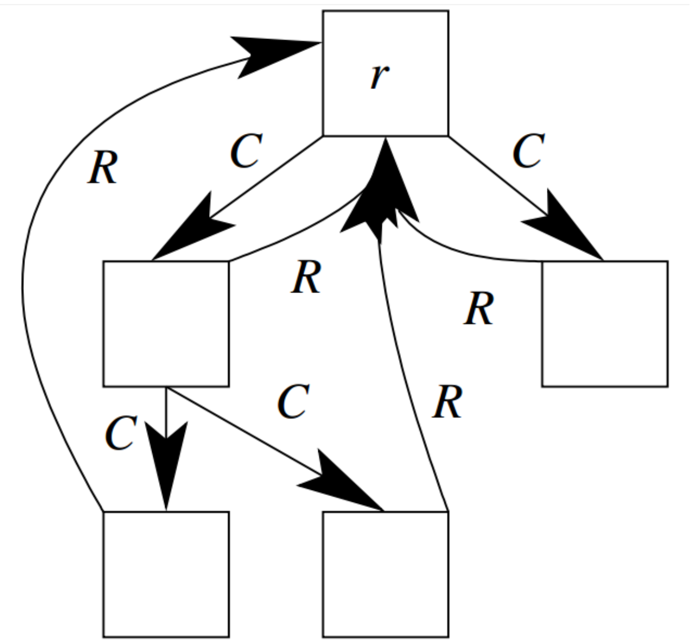
In fact, recall L4: more bits in encoding often better

Reducing the Size of Encoding

Graph classifiers

Synthesize graph classifiers (ie, *repOK* checkers), eg:

- singly linked list
- cyclic linked list
- doubly linked list
- directed tree
- tree with parent pointer ---->
- strongly connected



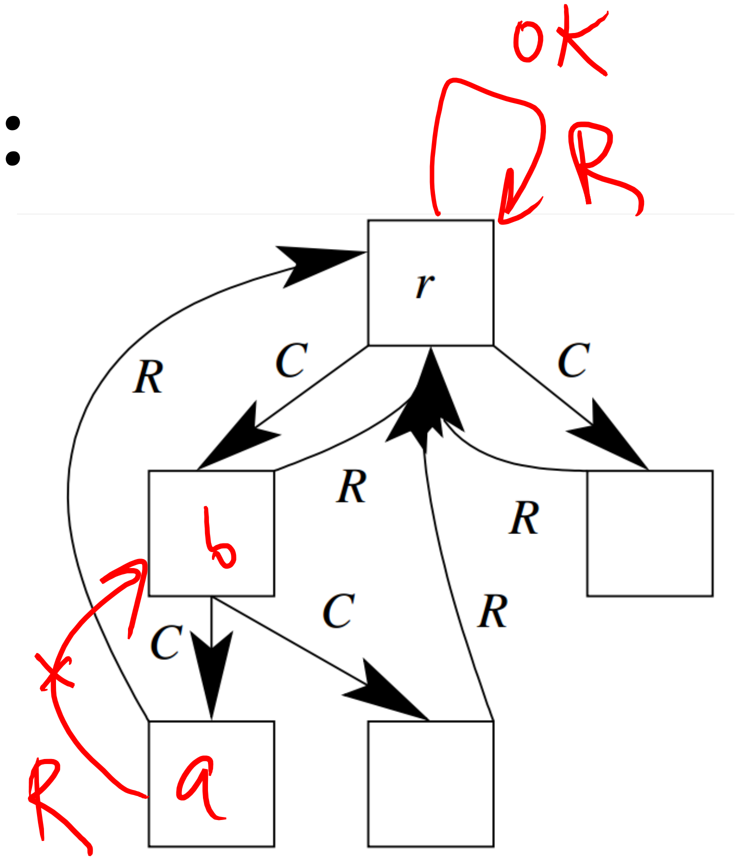
Ensure **linear** running time.

[Izthaky *et al*, OOPSLA 2010]

root parent

Specification (tree with ~~parent~~ pointer)

Precondition (integrity assumption):



root r via C ∧ functional R

all nodes are reachable from r via C

Postcondition (classification):

a R b

a R r

C is 1:1 $\wedge\ \forall u\,.\ \neg C(u,r) \wedge \neg R(r,u) \wedge (u \neq r \Rightarrow R(u,r))$

no two parents

root has no parent

∀ non root points to root

Synthesized linear-time classifier

The classifier (not a simple paraphrase of the spec!):

$$\#pC(r) = 0 \ \wedge \ pR(r) = \ s_{C+}(r) \ \wedge \ \forall v \ (\#p_C(v) \le 1)$$

Explained:

$\#p_C(r) = 0$	The cardinality of the set of C-predecessors of the root *r* is 0.
$p_R(r) = \ sC_+(r)$	The set of *R*-predecessors of the root equals the set of nodes forward reachable from the root.
$\forall v \ (\#p\underline{C}(v) \le 1)$	Each node is a child of no more than one node.

This classifier still looks declarative to me!

This classifier can be compiled to an operational pgm.

with guaranteed linear time performance

First, using DFS, compute inverse edges

so that we can compute predecessor sets p_C, p_R

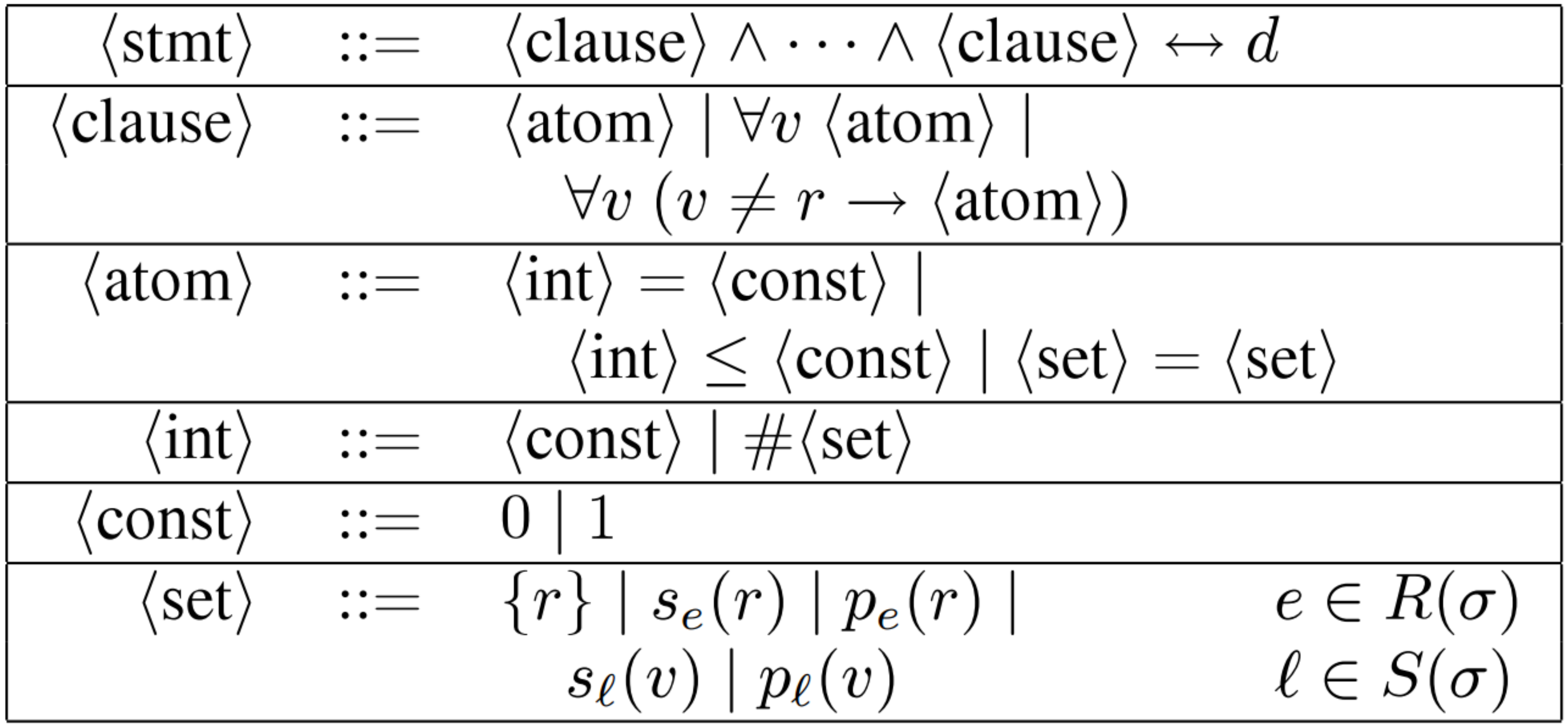
Next, compute these conditions with DFS:

$\#p_C(r) = 0$	$O(1)$
$p_R(r) = sC_+(r)$	$O(E)$
$\forall v\ (\#pC(v) \leq 1)$	$O(E)$

The partial program

Recall that a partial program (sketch) is a grammar.

each classifier is a <stmt> from this grammar



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

How is linear time guaranteed?

The partial program contains only one variable, *v*

hence we cannot form properties over, say, pairs of nodes

Reachability across label strings only from the root

$s_{C+}(r)$ is legal but $s_{C+}(v)$ is not

why? evaluating, say, $\forall v\ \#p_{\mathrm{A*(}}\mathrm{v}) = 1$ needs $O(n^2 \lg n)$ time

Regular expressions are bounded in length, of course

$s_{\underbrace{\mathrm{B+C*A+}}}(r)$ hence they can be computed during DFS

Discussion

What did we gain with this high-level program?

encoding:

solver efficiency:

engineering complexity:

Their inductive synthesis algorithm

Simple thanks to the structure of the language:

1. assume you have positive and negative instance sets P, N.
2. enumerate all clauses C
3. find clauses C_P that are true on each graph in P
4. find smallest subset $\{ci_1, ci_2, \dots, c_{ik}\}$ of C_P such that $c_{i1} \wedge c_{i2} \wedge \dots \wedge c_{ik}$ is false for all graphs from N

Summary of Izhaky *et al*

The key concept we have seen is

synthesis at high-level of abstraction

- guarantees resource constraints (here, linear time)
- a simpler synthesis algorithm

followed by deterministic compilation

- essentially, this is just pattern-driven code generation
- eg, translate #p_C(v) to some fixed code

Other uses of languages?

Summary

synthesis followed by deterministic compile

- the compiler could benefit from synthesis, though

higher-level abstraction ==> smaller programs and thus smaller formulas

- not by itself smaller search spaces

reduce search space via domain constraints

- eg, what rotations are legal

Concepts not covered

constructs for specs, including examples

- ex: angelic programming could create examples inputs

reduce ambiguity

- if your spec is incomplete (eg examples), then smaller candidate space reduces ambiguity in the spec

feedback to the user/programmer in familiar domain

- eg describing the localized bug using unsat core

support abstraction that will be used in synthesis

- ignore actual value in AG, actual multiplication in HPC codes

implicitly codify domain properties

  – so that you can automatically determine that a single

Looking ahead

Languages that will be built in cs294 projects:

- distributed protocols (asynchrony, lost messages)
- distributed protocols (bounded asynchrony)
- web scraping (how to name DOM elements)
- spatial programming in forth
- attribute grammar evaluators
- distributed memory data structures and operations
- parsers for programming contests

Next lecture (Tuesday)

Read *Fudging up Racket*

Implementing a language in Racket

Optimizations