CS5733 Program Synthesis

#5. Propositional Logic
Normal Forms

Ashish Mishra, August 13, 2024

Partly based on slides by Roopsha Samata at Purdue

The Paper

EUSolver

- Q1: What does EUSolver use as behavioral constraints? Structural
  - constraint? Search strategy?
  - First-order formula
  - Conditional expression grammar
  - Bottom-up enumerative with OE + pruning
- Why do they need the specification to be pointwise?
  - How would it break the enumerative solver?

EUSolver

- Q2: What are pruning/decomposition techniques EUSolver used to speed up the search?
  - Condition abduction + (special form of) equivalence reduction
- Why does EUSolver keep generating additional terms when all inputs are covered?
- How is the EUSolver equivalence reduction differ from observational equivalence we saw in class?
  - Only takes input coverage as the judgement, rather than similar behavior.
- Can we discard a term that covers a subset of the points covered by another term?

EUSolver

- Q3: What would be a naive alternative to decision tree learning for synthesizing branch conditions?
  - Learn atomic predicates that precisely classify points
    - why is this worse?
    - is it as bad as ESolver?
- Next best thing is decision tree learning w/o heuristics
  - why is this worse?

EUSolver: strengths

Divide-and-conquer (aka condition abduction)

- scales better on conditional expressions
- but: they didn't invent it

Neat application of decision tree learning

- leverages the structure of Boolean expressions

Empirically does well, especially on PBE

EUSover: weaknesses

Only applies to conditional expressions

Does not always generate the smallest expression

- in the limit, can find the smallest solution
- but unclear when to stop

Only works for pointwise specifications

- but so do ALL CEGIS-based approaches

No solution size evaluation beyond those solved by ESolver

No ablation of DT repair / branch-wise verification

Reading: point-wise,

Counterexample-Guided Quantifier Instantiation for Synthesis in SMT, CAV '15

Top-down enumeration pruning, continue...

Types and Type based Top-down pruning

Example

Drop the smallest element from each list

$$\begin{bmatrix} [71, & 75, & 83] \\ [90, & 87, & 95] \\ [68, & 77, & 80] \end{bmatrix} \Rightarrow \begin{bmatrix} [75, & 83] \\ [90 & 95] \\ [77 & 80] \end{bmatrix}$$

Example

$$\begin{bmatrix} [71, & 75, & 83] \\ [90, & 87, & 95] \\ [68, & 77, & 80] \end{bmatrix} \Rightarrow \begin{bmatrix} [75, & 83] \\ [90 & 95] \\ [77 & 80] \end{bmatrix}$$

How can we discover this program?

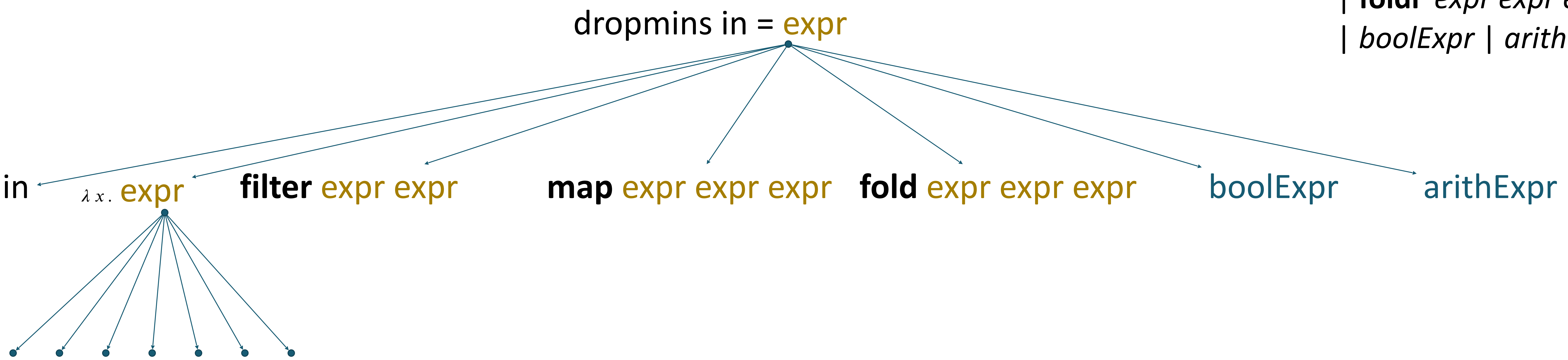
```
dropmins x = map dropmin x
    where dropmin y = filter isNotMin y
        where isNotMin z = foldl h False y
            where h t w = t || (w < z)
```

Defining the language

expr = var
| $\lambda x\,.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*

Top-down search

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



Many of these programs can be eliminated before having to complete them!

How?

Top-down search

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*

dropmins in = expr

in | $\lambda x.$ expr | **filter** expr expr | **map** expr expr | **fold** expr expr expr | boolExpr | arithExpr

This is a fully concrete program, and it clearly doesn't match the examples

Top-down search

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*

dropmins in = expr

~~in~~ $\lambda x.$ expr **filter** expr expr **map** expr expr **fold** expr expr expr boolExpr arithExpr

This program has a missing expression, but we can already tell it will not work. Why not?

Types

Our simple language supports an infinite set of types of 3 basic kinds

$$\tau \;:=\; Int \;\mid\; Bool \;\mid\; [\tau] \;\mid\; \tau \rightarrow \tau$$

Integer

Boolean

List of some type

Function from some type to some other type

Types

$$\begin{bmatrix} [71, & 75, & 83] \\ [90, & 87, & 95] \\ [68, & 77, & 80] \end{bmatrix} \Rightarrow \begin{bmatrix} [75, & 83] \\ [90 & 95] \\ [77 & 80] \end{bmatrix}$$

$$[\ [Int]\ ] \qquad\qquad [\ [Int]\ ]$$

Input and output types are lists of lists of integers

Types

Each element in our language has a type given by a *typing rule*

$$\frac{premises}{C \vdash expr : \tau}$$

A typing rule like the one above states that $expr$ has type τ in a context C as long as all the premises are satisfied

- A context simply tracks information about the type of any variables

Types

Each element in our language has a type given by a *typing rule*

$$\frac{C \text{ says var} \quad \textit{has type } \tau}{C \vdash \text{var} : \tau}$$

$$\frac{f : \tau_1 \rightarrow \tau_2 \qquad epxr : \tau_1}{C \vdash f\ expr : \tau_2}$$

$$\frac{C, x : \tau_1 \vdash expr : \tau_2}{C \vdash \lambda\, \text{x}\,.\ \text{expr} : \tau_1 \rightarrow \tau_2}$$

$$\frac{}{map : (\tau_1 \rightarrow \tau_2) \rightarrow [\tau_1] \rightarrow [\tau_2]}$$

$$\frac{}{foldl : (\tau_{start} \rightarrow \tau_{lst} \rightarrow \tau_{start}) \rightarrow \tau_{start} \rightarrow [\tau_{lst}] \rightarrow \tau_{start}}$$

$$\frac{}{boolExpr : Bool}$$

$$\frac{}{filter : (\tau \rightarrow Bool) \rightarrow [\tau] \rightarrow [\tau]}$$

$$\frac{}{intExpr : Int}$$

Type-based pruning

expr = var
| $\lambda x .\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*

dropmins in = expr

~~in~~ | $\lambda x .$ expr | **filter** expr expr | **map** expr expr | **fold** expr expr expr | boolExpr | arithExpr

$$\frac{expr : \tau_2 \ assuming \ x : \tau_1}{\lambda\, x .\ \mathrm{expr} : \tau_1 \rightarrow \tau_2}$$

Based on the rule, this expression will have a type $\tau_1 \rightarrow \tau_2$

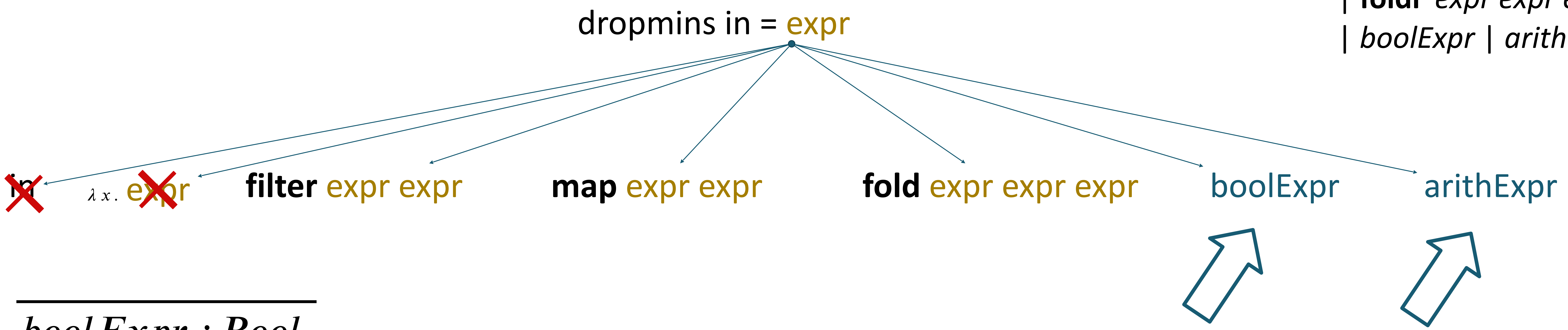
But we know the output must have type $[\ [Int]\ ]$

There is no way those types can be made equal,

so we can discard this expression!

Type-based pruning

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



$$\overline{boolExpr : Bool}$$

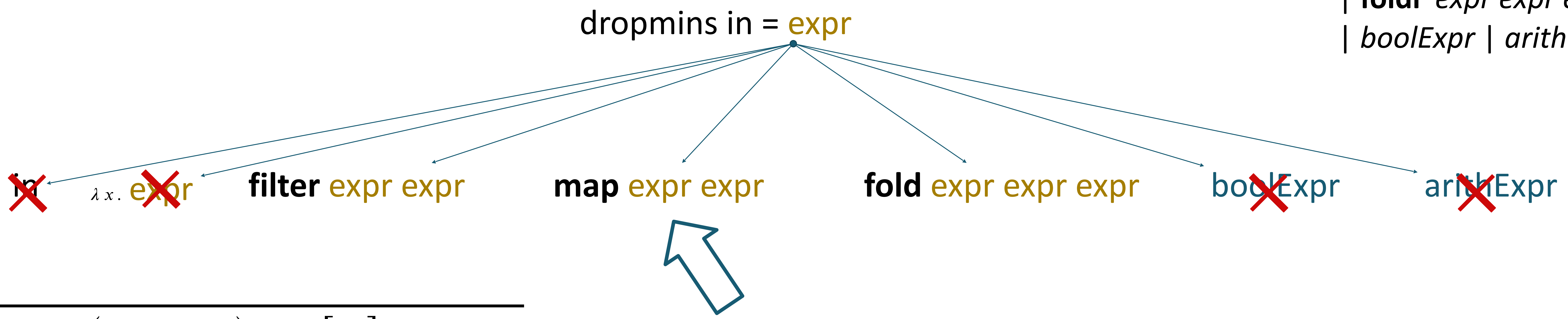
$$\overline{intExpr : Int}$$

With the same reasoning we can discard both of these expressions

They cannot possibly have the correct type

Type-based pruning

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



$$map: (\tau_1 \rightarrow \tau_2) \rightarrow [\tau_1] \rightarrow [\tau_2]$$

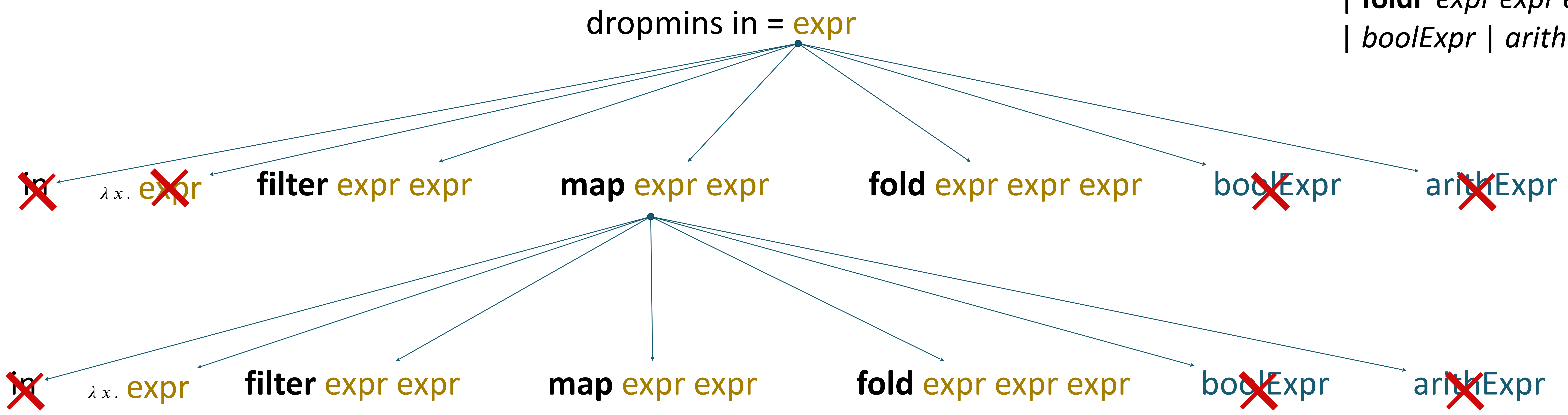
We know the output should be $[[Int]]$

This means the first expr must be $\tau_1 \rightarrow [Int]$

otherwise the types won't match

Type-based pruning

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



We can quickly dismiss many possible expressions because they cannot produce the type $\tau_1 \rightarrow [Int]$

Program Synthesis

Program Verification

Propositional Logic Normal Forms

Calculus of Computation?

> *It is reasonable to hope that the relationship between* **computation** *and* **mathematical logic** *will be as fruitful in the next century as that between* **analysis** *and* **physics** *in the last. The development of this relationship demands a concern for both applications and mathematical elegance.*

John McCarthy
A Basis for a Mathematical Theory of Computation, 1963

Propositional logic (PL) syntax

Atom	truth symbols	⊤ (“true”) and ⊥ (“false”)
	propositional variables	p, q, r, p_1, q_1
Literal	atom α or its negation $\neg\alpha$	
Formula	literal or application of a logical connective to F, F_1, F_2	

$\neg F$	“not”	(negation)
$F_1 \vee F_2$	“or”	(disjunction)
$F_1 \wedge F_2$	“and”	(conjunction)
$F_1 \rightarrow F_2$	“implies”	(implication)
$F_1 \leftrightarrow F_2$	“if and only if”	(iff)

Example

formula $F : (P \wedge Q) \ \rightarrow \ (\top \vee \neg Q)$

atoms: $P,\ Q,\ \top$

literals: $P,\ Q,\ \top,\ \neg Q$

subformulae: $P,\ Q,\ \top,\ \neg Q,\ P \wedge Q,\ \top \vee \neg Q,\ F$

abbreviation

$$F : P \wedge Q \ \rightarrow \ \top \vee \neg Q$$

PL Semantics (Meaning)

$$\text{Sentence } F + \text{Interpretation } I = \text{Truth value (true, false)}$$

Interpretation

$$I : \{P \mapsto \text{true}, Q \mapsto \text{false}, \cdots\}$$

Evaluation of F under I:

F	$\neg F$
0	1
1	0

where 0 corresponds to value false
1 corresponds to value true

$I \models F$ if F evaluates to true under I
$I \not\models F$ if F evaluates to false under I

F_1	F_2	$F_1 \wedge F_2$	$F_1 \vee F_2$	$F_1 \rightarrow F_2$	$F_1 \leftrightarrow F_2$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Satisfying and Falsifying Interpretations

Example

$F : P \land Q \rightarrow P \lor \neg Q$
$I : \{P \mapsto \text{true}, Q \mapsto \text{false}\}$

P	Q	$\neg Q$	$P \land Q$	$P \lor \neg Q$	F
1	0	1	0	1	1

$1 = \text{true}$ $\qquad$ $0 = \text{false}$

F evaluates to true under I

PL Semantics (Inductive definitions)

Base Case:

$$I \models \top$$
$$I \not\models \bot$$
$$I \models P \quad \text{iff} \quad I[P] = \text{true}$$
$$I \not\models P \quad \text{iff} \quad I[P] = \text{false}$$

Inductive Case:

$$I \models \neg F \quad \text{iff } I \not\models F$$
$$I \models F_1 \wedge F_2 \quad \text{iff } I \models F_1 \text{ and } I \models F_2$$
$$I \models F_1 \vee F_2 \quad \text{iff } I \models F_1 \text{ or } I \models F_2$$
$$I \models F_1 \rightarrow F_2 \quad \text{iff, if } I \models F_1 \text{ then } I \models F_2$$
$$I \models F_1 \leftrightarrow F_2 \quad \text{iff, } I \models F_1 \text{ and } I \models F_2, \text{ or } I \not\models F_1 \text{ and } I \not\models F_2$$

Note:

$$I \not\models F_1 \rightarrow F_2 \quad \text{iff} \quad I \models F_1 \text{ and } I \not\models F_2$$

Example

$$F : P \wedge Q \rightarrow P \vee \neg Q$$

$$I : \{P \mapsto \text{true}, Q \mapsto \text{false}\}$$

1.	$I \models P$	since $I[P] = \text{true}$	
2.	$I \not\models Q$	since $I[Q] = \text{false}$	
3.	$I \models \neg Q$	by 2 and $\neg$	
4.	$I \not\models P \wedge Q$	by 2 and $\wedge$	
5.	$I \models P \vee \neg Q$	by 1 and $\vee$	
6.	$I \models F$	by 4 and $\rightarrow$	Why?

Thus, F is true under I.

Satisfiability and Validity

F is satisfiable iff there exists $I : I \vDash F$

F is valid iff for all $I : I \vDash F$

Duality:
F is valid iff $\neg F$ is unsatisfiable

Procedure for deciding satisfiability *or* validity suffices!

Deciding satisfiability/validity

- Basic techniques
  - Truth table method: search-based
  - Semantic argument method: deductive technique
- SAT solvers
  - Combine search and deduction

Truth table method

1. Enumerate all interpretations
2. Search for satisfying interpretation

Brute-force!
Impractical (2^n interpretations)
Can't be used if domain is not finite, e.g., for first-order logic

$$F: P \land Q \rightarrow P \lor \neg Q$$

P	Q	$P \land Q$	$\neg Q$	$P \lor \neg Q$	F
$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$

Thus F is valid.

Example

$$F : P \lor Q \to P \land Q$$

P	Q	$P \lor Q$	$P \land Q$	F	
0	0	0	0	1	$\leftarrow$ satisfying I
0	1	1	0	0	$\leftarrow$ falsifying I
1	0	1	0	0	
1	1	1	1	1	

Thus F is satisfiable, but invalid.

Method 2: Semantic Argument

Proof by contradiction:

1. Assume F is not valid
2. Apply proof rules
3. Contradiction (i.e, $\bot$) along every branch of proof tree $\Rightarrow F$ is valid
4. Otherwise, F is not valid

A bit of an overhead for PL

Applicable to first-order logic

Proof rules

$$\frac{I \models \neg F}{I \not\models F} \qquad \frac{I \not\models \neg F}{I \models F}$$

$$\frac{I \models F \wedge G}{\begin{array}{l} I \models F \\ I \models G \end{array}} \leftarrow \text{and} \qquad \frac{I \not\models F \wedge G}{I \not\models F \mid I \not\models G} \ \nwarrow \text{or}$$

$$\frac{I \models F \vee G}{I \models F \mid I \models G} \qquad \frac{I \not\models F \vee G}{\begin{array}{l} I \not\models F \\ I \not\models G \end{array}}$$

$$\frac{I \models F \rightarrow G}{I \not\models F \mid I \models G} \qquad \frac{I \not\models F \rightarrow G}{\begin{array}{l} I \models F \\ I \not\models G \end{array}}$$

$$\frac{I \models F \leftrightarrow G}{I \models F \wedge G \mid I \not\models F \vee G} \qquad \frac{\begin{array}{l} I \models F \\ I \not\models F \end{array}}{I \models \bot}$$

$$\frac{I \not\models F \leftrightarrow G}{I \models F \wedge \neg G \mid I \models \neg F \wedge G}$$

Example

To Prove $F: P \wedge Q \rightarrow P \vee \neg Q$ is valid.

Let's assume that F is not valid and that I is a falsifying interpretation.

1.	I	$\not\models$	$P \wedge Q \rightarrow P \vee \neg Q$	assumption
2.	I	$\models$	$P \wedge Q$	1 and $\rightarrow$
3.	I	$\not\models$	$P \vee \neg Q$	1 and $\rightarrow$
4.	I	$\models$	P	2 and $\wedge$
5.	I	$\not\models$	P	3 and $\vee$
6.	I	$\models$	$\perp$	4 and 5 are contradictory

Thus F is valid

Example 2

To Prove $F: (P \rightarrow Q) \wedge (Q \rightarrow R) \rightarrow (P \rightarrow R)$ is valid.

Example 2

Let's assume that F is not valid.

1.	I	$\not\models$	F	assumption
2.	I	$\models$	$(P \rightarrow Q) \land (Q \rightarrow R)$	1 and $\rightarrow$
3.	I	$\not\models$	$P \rightarrow R$	1 and $\rightarrow$
4.	I	$\models$	P	3 and $\rightarrow$
5.	I	$\not\models$	R	3 and $\rightarrow$
6.	I	$\models$	$P \rightarrow Q$	2 and of $\land$
7.	I	$\models$	$Q \rightarrow R$	2 and of $\land$

Example 2

Two cases from 6

$8a.$	I	$\not\models$	P	6 and $\rightarrow$
$9a.$	I	$\models$	$\bot$	4 and 8a are contradictory

and

$8b.$	I	$\models$	Q	6 and $\rightarrow$

Two cases from 7

$9ba.$	I	$\not\models$	Q	7 and $\rightarrow$
$10ba.$	I	$\models$	$\bot$	8b and 9ba are contradictory

and

$9bb.$	I	$\models$	R	7 and $\rightarrow$
$10bb.$	I	$\models$	$\bot$	5 and 9bb are contradictory

Our assumption is incorrect in all cases — F is valid.

Semantic judgements, Equivalence

F_1 and F_2 are equivalent ($F_1 \Leftrightarrow F_2$)

iff for all interpretations I, $I \models F_1 \leftrightarrow F_2$

To prove $F_1 \Leftrightarrow F_2$ show $F_1 \leftrightarrow F_2$ is valid.

F_1 implies F_2 ($F_1 \Rightarrow F_2$)

iff for all interpretations I, $I \models F_1 \rightarrow F_2$

$F_1 \Leftrightarrow F_2$ and $F_1 \Rightarrow F_2$ are not formulae!

Normal Forms

- A normal form for a logic is a syntactical restriction such that for every formula in the logic, there is an equivalent formula in the normal form.

- Three useful normal forms for propositional logic:
  - Negation Normal Form (NNF)
  - Disjunctive Normal Form (DNF)
  - Conjunctive Normal Form (CNF)

Negation Normal Form (NNF)

Atom	T , ⊥ , propositional variables
Literal	Atom | ¬Atom
Formula	Literal | Formula op Formula
op	∨ | ∧

The only logical connectives are ¬, ∧, ∨

Negations appear only in literals

Conversion to NNF:

Eliminate → and ↔

"Push negations in" using DeMorgan's Laws:

$\neg(F_1 \wedge F_2) \Leftrightarrow (\neg F_1 \vee \neg F_2)$

$\neg(F_1 \vee F_2) \Leftrightarrow (\neg F_1 \wedge \neg F_2)$

Example: Convert $F : \neg(P \rightarrow \neg(P \wedge Q))$ to NNF

$F' : \neg(\neg P \vee \neg(P \wedge Q))$ $\rightarrow$ to $\vee$

$F'' : \neg\neg P \wedge \neg\neg(P \wedge Q)$ De Morgan's Law

$F''' : P \wedge P \wedge Q$ $\neg\neg$

F ‴ is equivalent to F (F ‴ ⇔ F) and is in NNF

Disjunctive Normal Form (DNF)

Atom	$\top$, $\bot$, propositional variables
Literal	Atom | ¬Atom
Disjunct	Literal ∧ Disjunct
Formula	Disjunct ∨ Formula

Disjunction of conjunctions of literals

$$\bigvee_i \bigwedge_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}$$

Conversion to DNF:

First convert to NNF

Distribute ∧ over ∨

$$((F_1 \lor F_2) \land F_3) \Leftrightarrow ((F_1 \land F_3) \lor (F_2 \land F_3))$$

$$(F_1 \land (F_2 \lor F_3)) \Leftrightarrow ((F_1 \land F_2) \lor (F_1 \land F_3))$$

Deciding satisfiability of DNF formulas is trivial
Why not convert all PL formulas to DNF for SAT solving?
Exponential blow-up of formula size in DNF conversion!

Example

Example: Convert

$$F: \ (Q_1 \ \vee \ \neg\neg Q_2) \ \wedge \ (\neg R_1 \ \rightarrow \ R_2) \text{ into DNF}$$

$$\begin{aligned} F' &: (Q_1 \vee Q_2) \wedge (R_1 \vee R_2) && \text{in NNF} \\ F'' &: (Q_1 \wedge (R_1 \vee R_2)) \vee (Q_2 \wedge (R_1 \vee R_2)) && \text{dist} \\ F''' &: (Q_1 \wedge R_1) \vee (Q_1 \wedge R_2) \vee (Q_2 \wedge R_1) \vee (Q_2 \wedge R_2) && \text{dist} \end{aligned}$$

F ′′′ is equivalent to F (F ′′′ ⇔ F) and is in DNF

Conjunctive Normal Form (CNF)

Atom	⊤ , ⊥ , propositional variables
Literal	Atom | ¬Atom
Clause	Literal ∨ Clause
Formula	Clause ∧ Formula

Conjunction of disjunctions of literals

$$\bigwedge_i \bigvee_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}$$

Deciding satisfiability of CNF formulas is not trivial
CNF conversion must also exhibit an exponential blow-up of formula size
Yet, almost all SAT solvers convert to CNF first before solving. Why?

Conversion to CNF:

First convert to NNF

Distribute ∨ over ∧

$$((F_1 \wedge F_2) \vee F_3) \Leftrightarrow ((F_1 \vee F_3) \wedge (F_2 \vee F_3))$$

$$(F_1 \vee (F_2 \wedge F_3)) \Leftrightarrow ((F_1 \vee F_2) \wedge (F_1 \vee F_3))$$

Natural representation because in practice, many formulas arise from multiple constraints that must hold *simultaneously* (AND).

Potential Problem with CNF: Size blowup

Distributivity will duplicate entire subformulas

Can happen repeatedly: $(p_1 \wedge p_2 \wedge p_3) \vee (q_1 \wedge q_2 \wedge q_3) =$
$(p_1 \vee (q_1 \wedge q_2 \wedge q_3)) \wedge (p_2 \vee (q_1 \wedge q_2 \wedge q_3)) \wedge (p_3 \vee (q_1 \wedge q_2 \wedge q_3))$
$= (p_1 \vee q_1) \wedge (p_1 \vee q_2) \wedge (p_1 \vee q_3)$
$\wedge (p_2 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_2 \vee q_3)$
$\wedge (p_3 \vee q_1) \wedge (p_3 \vee q_2) \wedge (p_3 \vee q_3)$

Worst-case blowup? : exponential!

Can't use this transformation for subsequent algorithms (e.g., satisfiability checking) if resulting formula is inefficiently large (possibly too large to represent/process).

Equisatisfiability and Tseitin's Transformation

Two formulas F_1 and F_2 are equisatisfiable iff:
F_1 is satisfiable iff F_2 is satisfiable

Tseitin's transformation converts any PL formula F_1 to equisatisfiable formula F_2 in CNF with only a linear increase in size

Note that equisatisfiability is a much weaker notion than equivalence, but is adequate for checking satisfiability.

Tseitin Transformation

Idea: rather than duplicate subformula:

introduce *new proposition* to represent it

add constraint: *equivalence* of subformula with new proposition

write this equivalence in CNF

Transformation rules for three basic operators

formula	$p \leftrightarrow$ formula	rewritten in CNF
$\neg A$	$(\neg A \to p) \wedge (p \to \neg A)$	$(A \vee p) \wedge (\neg A \vee \neg p)$
$A \wedge B$	$(A \wedge B \to p) \wedge (p \to A \wedge B)$	$(\neg A \vee \neg B \vee p) \wedge (A \vee \neg p) \wedge (B \vee \neg p)$
$A \vee B$	$(p \to A \vee B) \wedge (A \vee B \to p)$	$(A \vee B \vee \neg p) \wedge (\neg A \vee p) \wedge (\neg B \vee p)$

Tseitin's Transformation

1. Introduce an auxiliary variable rep(G) for each subformula $G = G_1 \; op \; G_2$ of formula F_1

2. Constrain auxiliary variable to be equivalent to subformula: rep(G) $\leftrightarrow$ rep(G_1) op rep(G_2)

3. Convert equivalence constraint to CNF: **CNF**(rep(G) $\leftrightarrow$ rep(G_1) op rep(G_2))

4. Let F_2 be rep(F) $\wedge$ $\bigwedge_G$ **CNF**(rep(G) $\leftrightarrow$ rep(G_1) op rep(G_2)). Check if F_2 is satisfiable.

F_1 and F_2 are equisatisfiable!

Size of each equivalence constraint is bounded by a constant

This restricts the size of F_2 to be linear in the size of F_1: $|F_2| = 30.|F_1| + 2$

Tseitin Transformation: Example

Add numbered proposition for each operator:

$(a \overset{1}{\land} \neg b) \lor \neg(c \overset{2}{\land} d)$

no need to number negations

nor top-level operator $(\ldots) \lor (\ldots)$

New propositions: $p_1 \leftrightarrow a \overset{1}{\land} \neg b, \quad p_2 \leftrightarrow c \overset{2}{\land} d$.

Rewrite equivalences for new propositions in CNF, conjunct with top-level operator of formula:

$(p_1 \lor \neg p_2)$

$\land (\neg a \lor b \lor p_1) \land (a \lor \neg p_1) \land (\neg b \lor \neg p_1)$

$\land (\neg c \lor \neg d \lor p_2) \land (c \lor \neg p_2) \land (d \lor \neg p_2)$

overall formula

$p_1 \leftrightarrow a \land \neg b$

$p_2 \leftrightarrow c \land d$

What do we get?

A new formula with more propositions than the original one
  NOT an equivalent formula

New formula is *satisfiable iff the original is satisfiable*
  we call it *equisatisfiable*)

Size of resulting formula: *linear* in original size
  good for use in satisfiability checking

Logistics

- Reviews for Week 3.
  - Due Thursday!
- Other questions?