Performance optimization of correct-by-design software in Lean

Wojciech Nawrocki[1,2] with Randal E. Bryant[1], Jeremy Avigad[1,2], and Marijn J. H. Heule[1,2]

[1] Carnegie Mellon University [2] Hoskinson Center for Formal Mathematics

February 3rd, 2025

IOG R&D Seminar

Slides at voidma.in/iog.pdf

Agenda

1. Knowledge compilation
2. Checking certificates
3. Optimizing the checker
4. Future outlook

Knowledge compilation

Model counting

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Model counting

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

$$\mathcal{M}(\varphi) \triangleq \{\nu : \mathrm{Vars}(\varphi) \to 2 \mid \varphi(\nu) = 1\}$$

Task: determine $|\mathcal{M}(\varphi)|$

Model counting

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$	
0	0	0	0	0	
0	1	0	1	1	
1	0	0	1	1	
1	1	1	1	0	
		1	3	2	$\lvert\mathcal{M}(-)\rvert$

$$\mathcal{M}(\varphi) \triangleq \{\nu : \mathrm{Vars}(\varphi) \to 2 \mid \varphi(\nu) = 1\}$$

Task: determine $\lvert\mathcal{M}(\varphi)\rvert$

Model counting

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$	
0	0	0	0	0	
0	1	0	1	1	
1	0	0	1	1	
1	1	1	1	0	
		1	3	2	$\lvert\mathcal{M}(-)\rvert$

$\mathcal{M}(\varphi) \triangleq \{\nu : \mathrm{Vars}(\varphi) \to 2 \mid \varphi(\nu) = 1\}$

Task: determine $\lvert\mathcal{M}(\varphi)\rvert$

$$\lvert\mathcal{M}((\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_3 \vee x_4))\rvert = ?$$

Complexity of model counting

- Model counting (♯SAT) generalizes SAT: φ is satisfiable iff $0 < |\mathcal{M}(\varphi)|$
-
-
-

Complexity of model counting

- Model counting (♯SAT) generalizes SAT: φ is satisfiable iff $0 < |\mathcal{M}(\varphi)|$
- ♯P-complete, and ♯P is maybe harder* than NP [2:ch. 25]
-
-

Complexity of model counting

- Model counting (♯SAT) generalizes SAT: φ is satisfiable iff $0 < |\mathcal{M}(\varphi)|$
- ♯P-complete, and ♯P is maybe harder* than NP [2:ch. 25]
- Applications in combinatorics, probabilistic inference, Boolean function synthesis, program synthesis [4]
-

Complexity of model counting

- Model counting (♯SAT) generalizes SAT: φ is satisfiable iff $0 < |\mathcal{M}(\varphi)|$
- ♯P-complete, and ♯P is maybe harder* than NP [2:ch. 25]
- Applications in combinatorics, probabilistic inference, Boolean function synthesis, program synthesis [4]
- Want to solve ♯SAT efficiently in practice

Knowledge compilation: CNF to POG

$$(\neg x_1 \vee x_3 \vee \neg x_4) \wedge$$
$$(\neg x_1 \vee \neg x_3 \vee x_4) \wedge$$
$$(\neg x_1 \vee \neg x_2) \wedge$$
$$(x_3 \vee \neg x_4) \wedge$$
$$(x_1 \vee \neg x_3 \vee x_4)$$

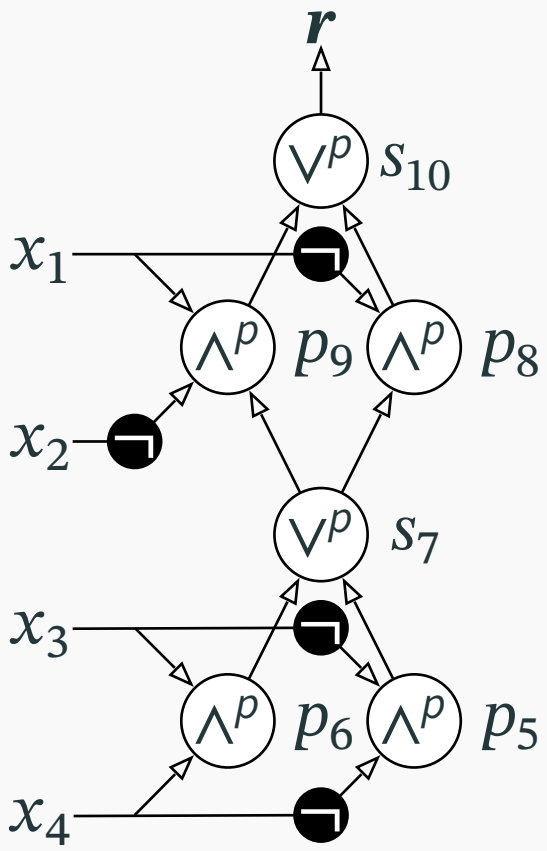
φ in conjunctive normal form

Knowledge compilation: CNF to POG

$$(\neg x_1 \vee x_3 \vee \neg x_4) \wedge$$
$$(\neg x_1 \vee \neg x_3 \vee x_4) \wedge$$
$$(\neg x_1 \vee \neg x_2) \wedge$$
$$(x_3 \vee \neg x_4) \wedge$$
$$(x_1 \vee \neg x_3 \vee x_4)$$

φ in conjunctive normal form

$\Longrightarrow$



φ as a partitioned-operation graph

Partitioned-operation graphs (POGs)

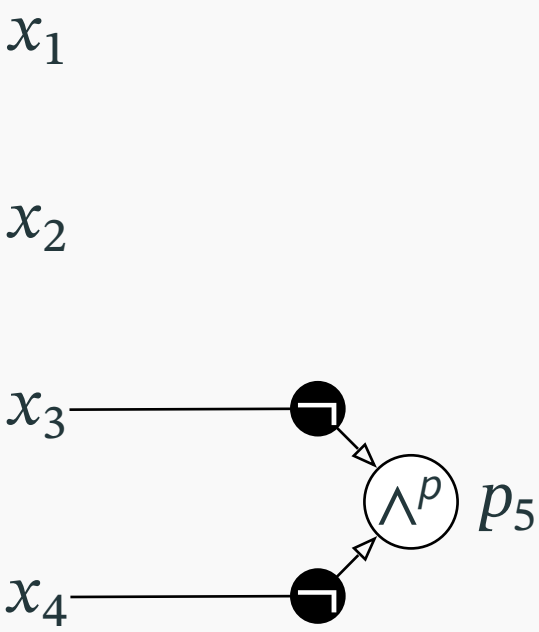
x_1

x_2

x_3

x_4

Partitioned-operation graphs (POGs)

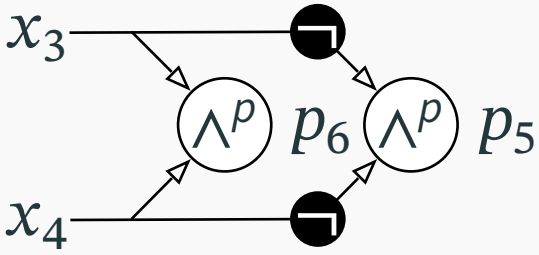


$$p_5 \triangleq \neg x_3 \wedge \neg x_4$$

Partitioned-operation graphs (POGs)

x_1

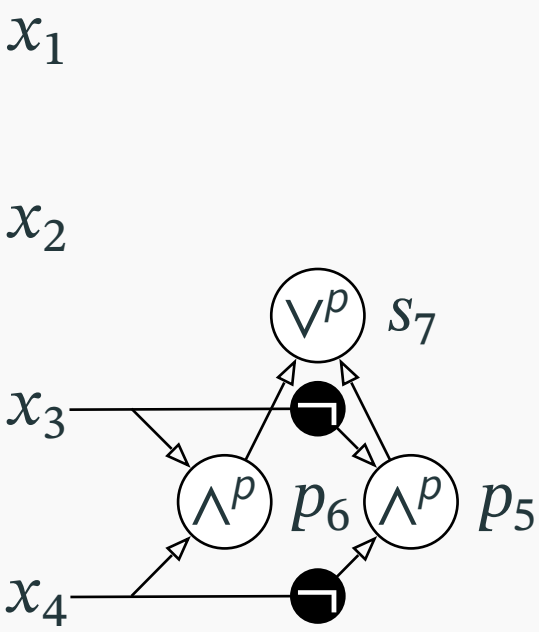
x_2



$$p_6 \triangleq x_3 \wedge x_4$$

$$p_5 \triangleq \neg x_3 \wedge \neg x_4$$

Partitioned-operation graphs (POGs)

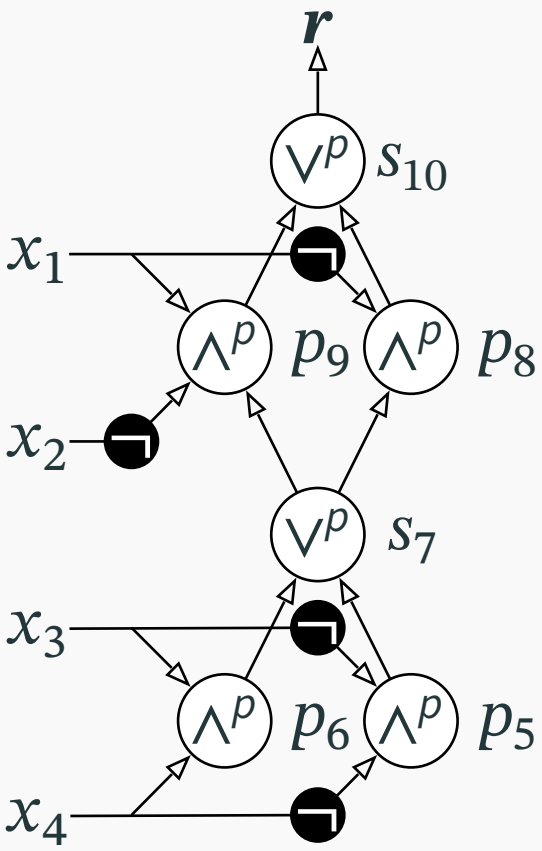


$$s_7 \triangleq p_6 \vee p_5$$

$$p_6 \triangleq x_3 \wedge x_4$$

$$p_5 \triangleq \neg x_3 \wedge \neg x_4$$

Partitioned-operation graphs (POGs)



$$\boldsymbol{r} \triangleq s_{10} \triangleq p_9 \vee p_8$$

$$p_9 \triangleq x_1 \wedge \neg x_2 \wedge s_7$$

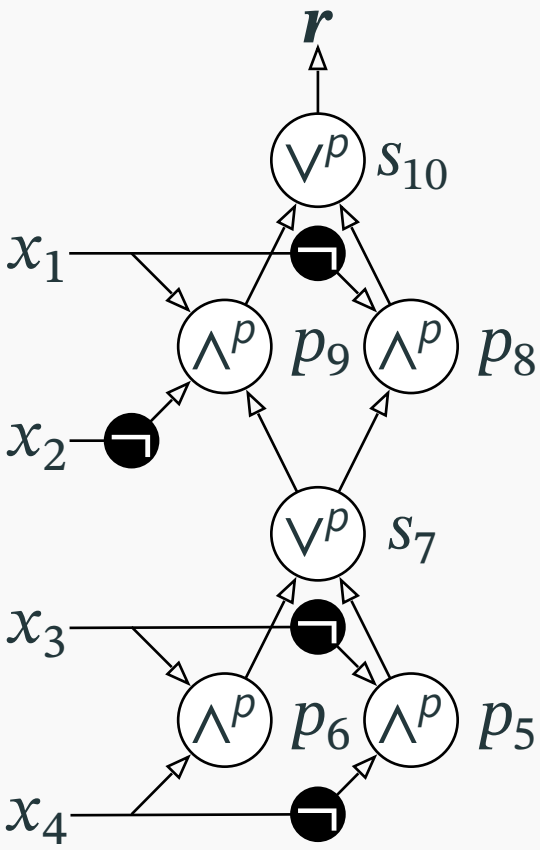
$$p_8 \triangleq \neg x_1 \wedge s_7$$

$$s_7 \triangleq p_6 \vee p_5$$

$$p_6 \triangleq x_3 \wedge x_4$$

$$p_5 \triangleq \neg x_3 \wedge \neg x_4$$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \emptyset$

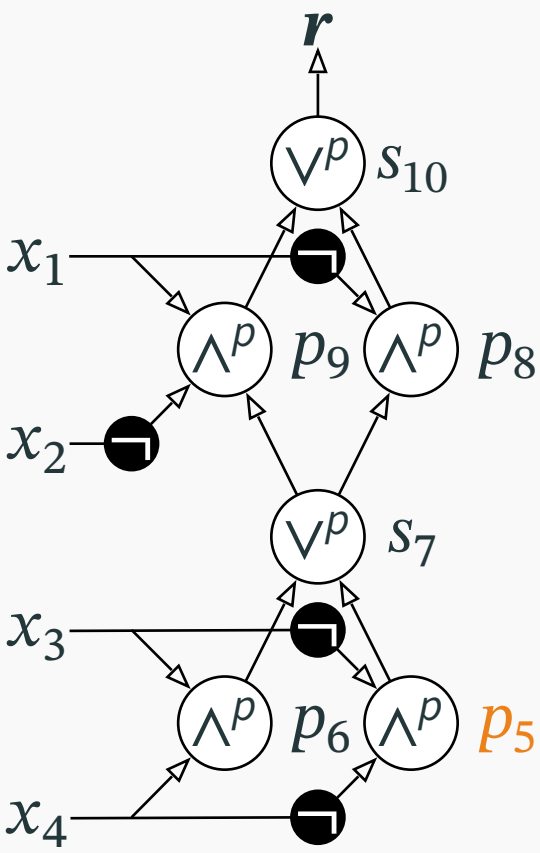
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \emptyset$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \emptyset$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\frac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \varnothing$

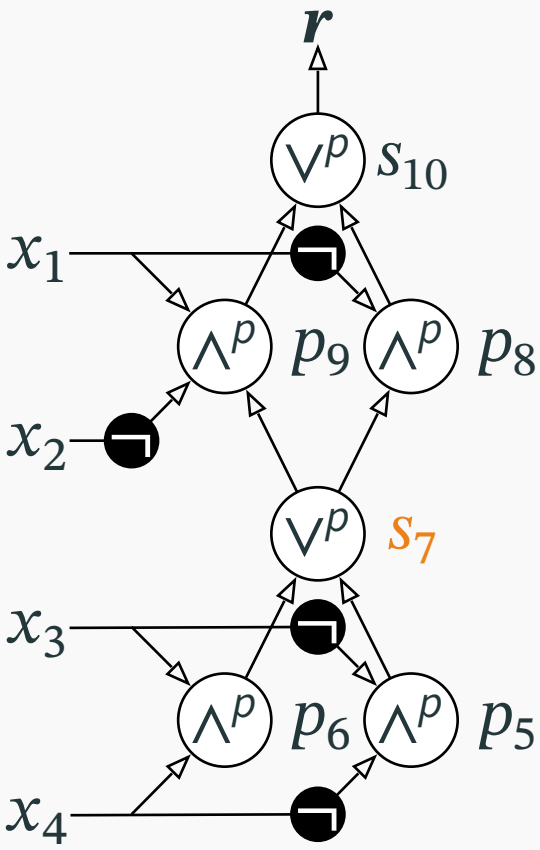
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \varnothing$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \varnothing$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\frac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\text{Vars}(\varphi) \cap \text{Vars}(\psi) = \varnothing$

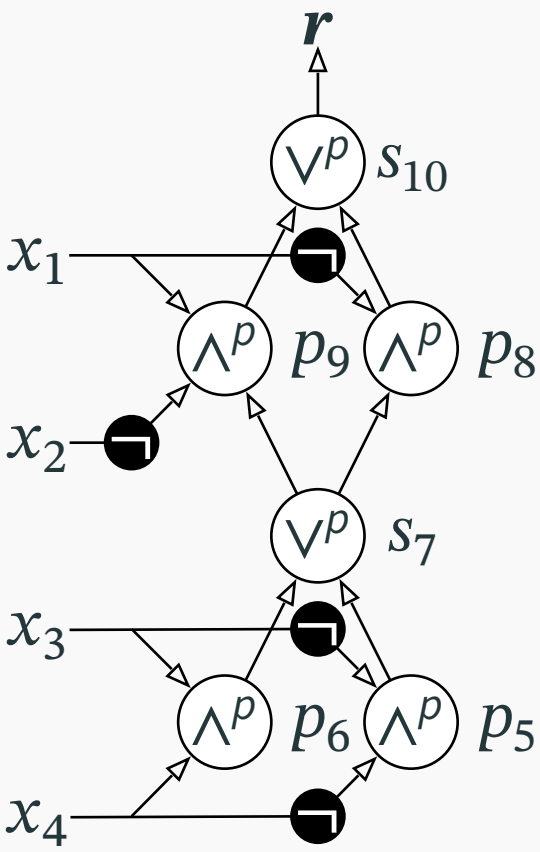
p_5 has $\text{Vars}(x_3) \cap \text{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \varnothing$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \varnothing$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\frac{|\mathcal{M}(\varphi)|}{2^{|\text{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \emptyset$

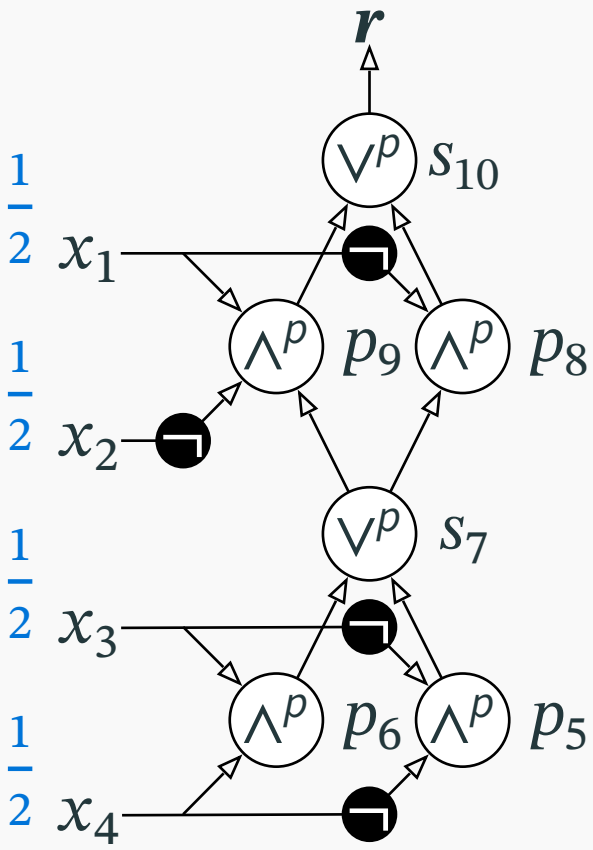
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \emptyset$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \emptyset$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\dfrac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \varnothing$

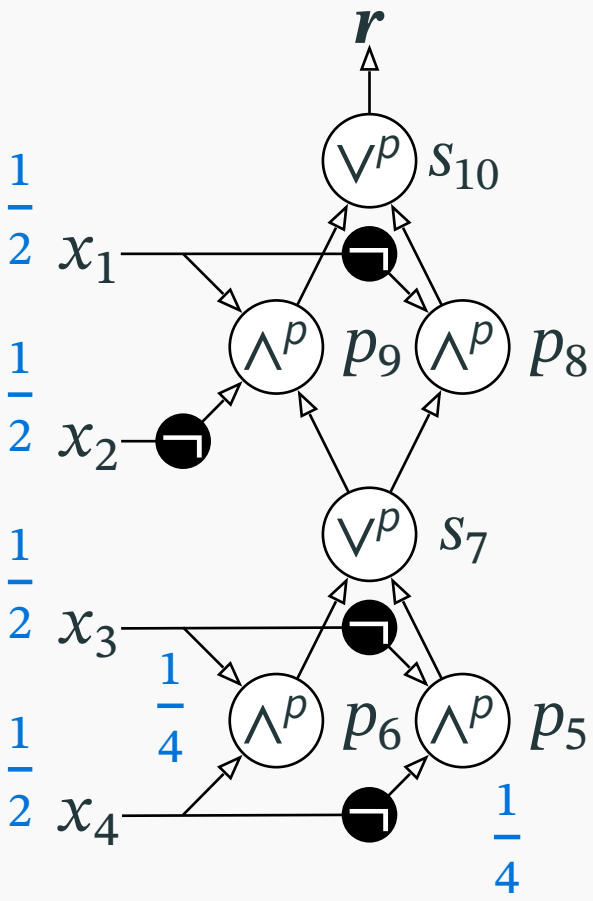
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \varnothing$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \varnothing$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\dfrac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \varnothing$

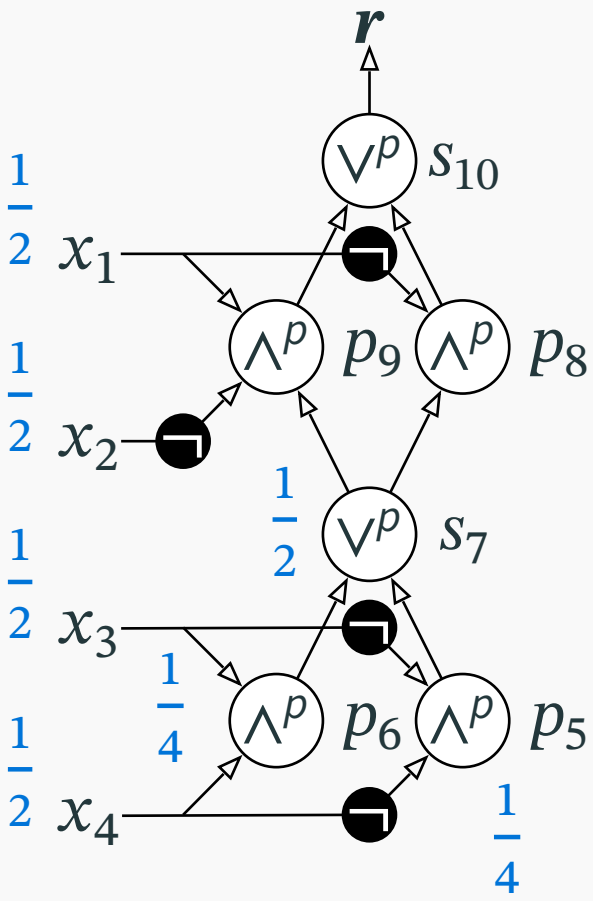
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \varnothing$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \varnothing$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\dfrac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \varnothing$

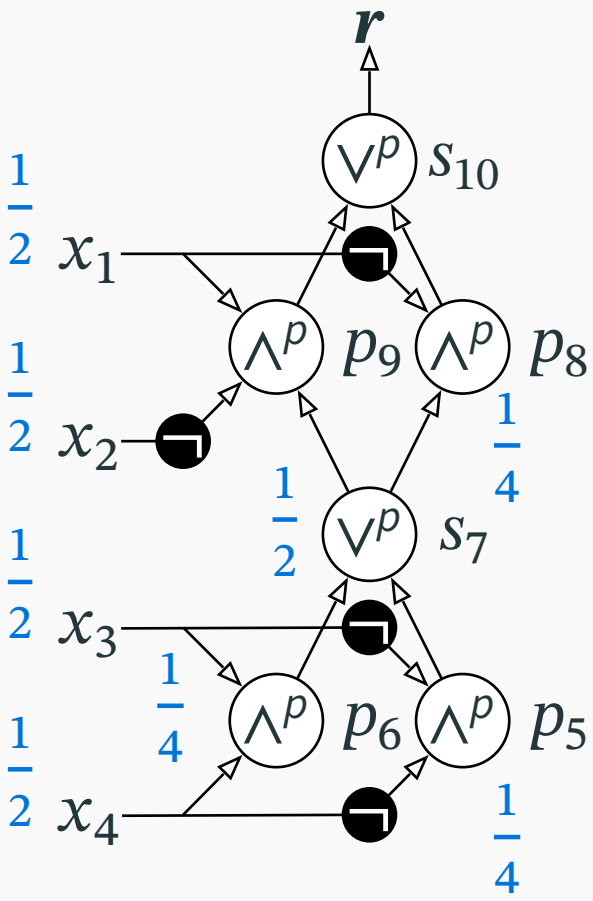
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \varnothing$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \varnothing$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\dfrac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \varnothing$

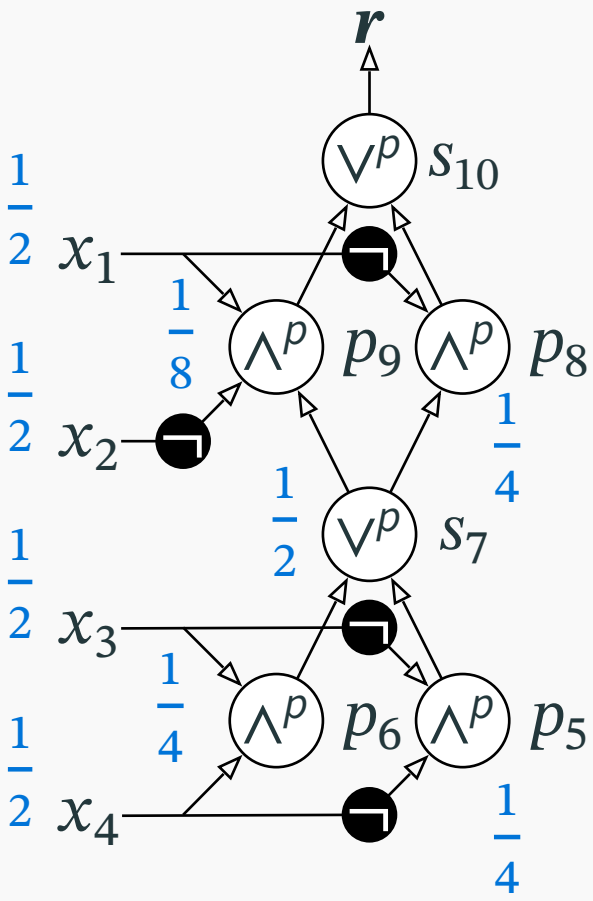
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \varnothing$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \varnothing$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\dfrac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\text{Vars}(\varphi) \cap \text{Vars}(\psi) = \emptyset$

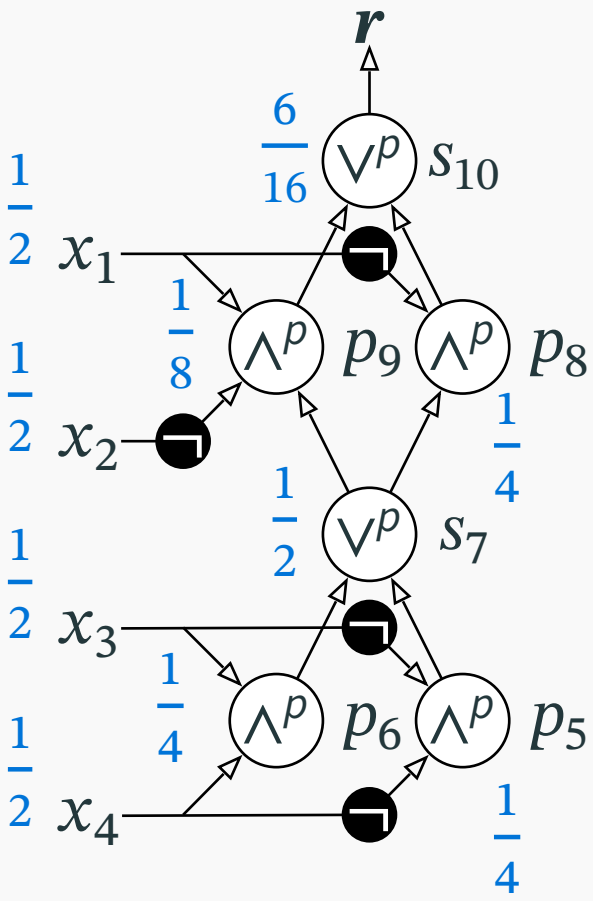
p_5 has $\text{Vars}(x_3) \cap \text{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \emptyset$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \emptyset$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\frac{|\mathcal{M}(\varphi)|}{2^{|\text{Vars}(\varphi)|}}$

Knowledge compilation: counting models



$\varphi \wedge^p \psi$ requires $\mathrm{Vars}(\varphi) \cap \mathrm{Vars}(\psi) = \varnothing$

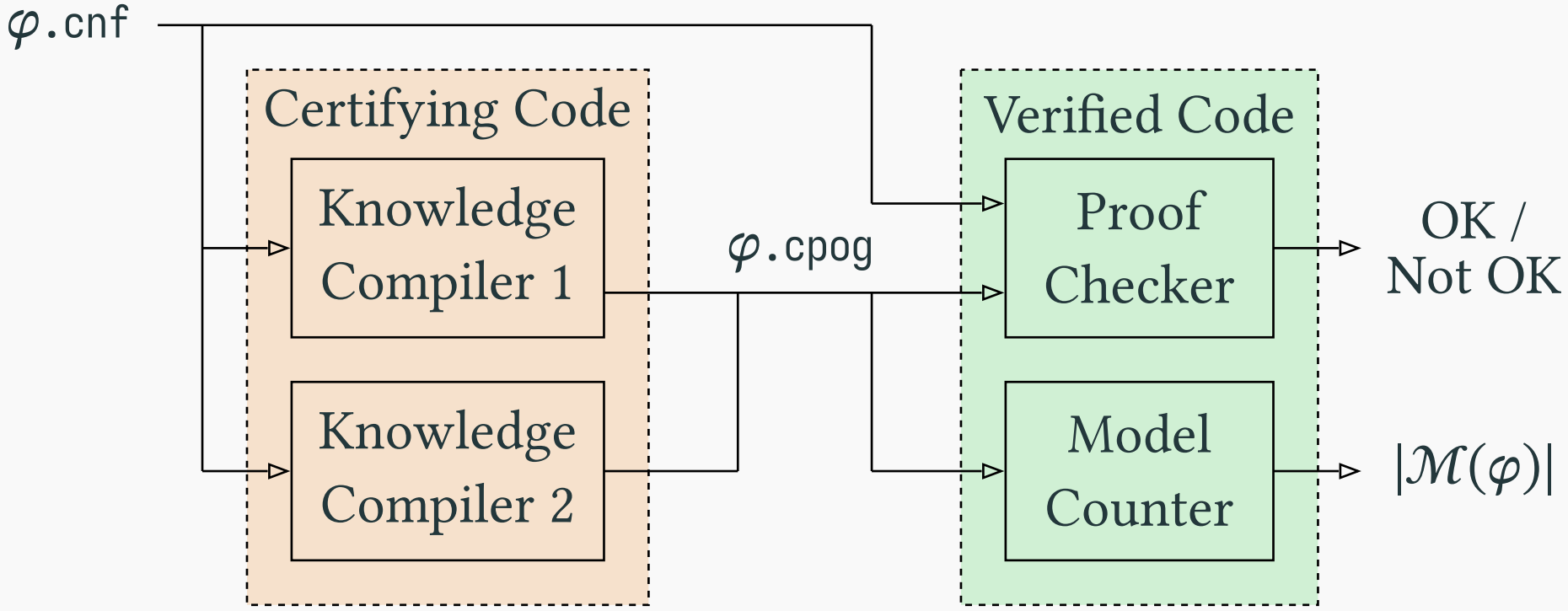
p_5 has $\mathrm{Vars}(x_3) \cap \mathrm{Vars}(x_4) = \{x_3\} \cap \{x_4\} = \varnothing$

$\varphi \vee^p \psi$ requires $\mathcal{M}(\varphi) \cap \mathcal{M}(\psi) = \varnothing$

s_7 has $\nu \in \mathcal{M}(p_5) \cap \mathcal{M}(p_6) \Rightarrow \nu(x_3) = 0 \wedge \nu(x_3) = 1$

Compute density $\dfrac{|\mathcal{M}(\varphi)|}{2^{|\mathrm{Vars}(\varphi)|}} = \dfrac{6}{16}$

Data flow in a certifying toolchain



- Standard approach to certification in automated reasoning (e.g. DRAT)

Checking certificates

How the checker operates

$(\neg x_1 \vee x_3 \vee \neg x_4)$

$(\neg x_1 \vee \neg x_3 \vee x_4)$

$(\neg x_1 \vee \neg x_2)$

$(x_3 \vee \neg x_4)$

$(x_1 \vee \neg x_3 \vee x_4)$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

x_1

x_2

x_3

x_4

Clause database $\approx \varphi$ | CPOG certificate | Partitioned-op. graph

How the checker operates

$(\neg x_1 \vee x_3 \vee \neg x_4)$

$(\neg x_1 \vee \neg x_3 \vee x_4)$

$(\neg x_1 \vee \neg x_2)$

$(x_3 \vee \neg x_4)$

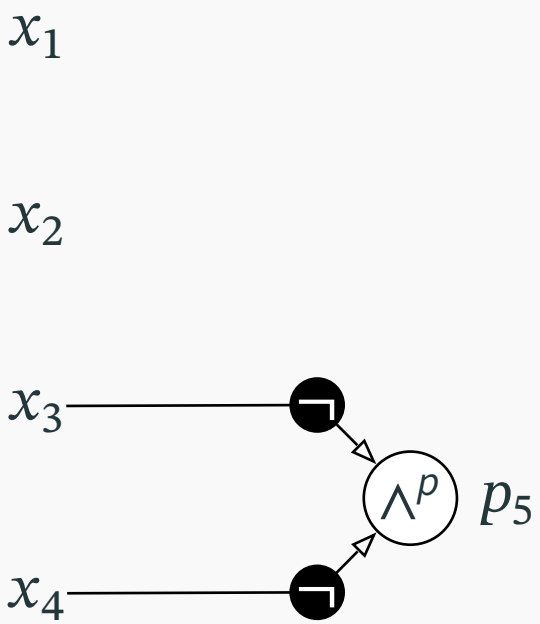
$(x_1 \vee \neg x_3 \vee x_4)$

$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



Partitioned-op. graph

How the checker operates

$(\neg x_1 \vee x_3 \vee \neg x_4)$

$(\neg x_1 \vee \neg x_3 \vee x_4)$

$(\neg x_1 \vee \neg x_2)$

$(x_3 \vee \neg x_4)$

$(x_1 \vee \neg x_3 \vee x_4)$

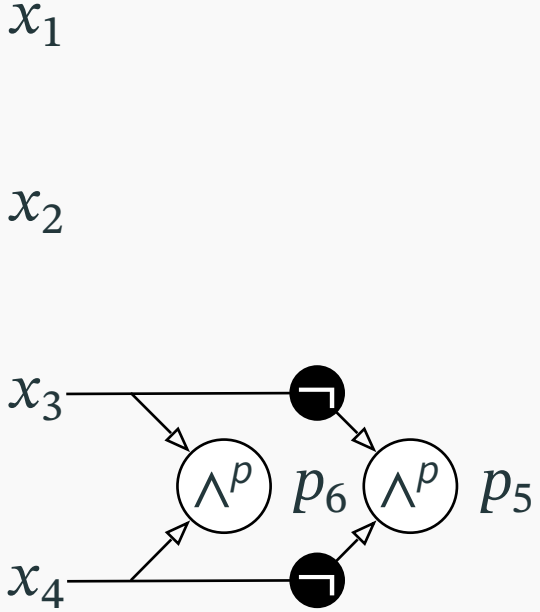
$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

$(p_6 \leftrightarrow (x_3 \wedge x_4))$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



Partitioned-op. graph

How the checker operates

$(\neg x_1 \vee x_3 \vee \neg x_4)$

$(\neg x_1 \vee \neg x_3 \vee x_4)$

$(\neg x_1 \vee \neg x_2)$

$(x_3 \vee \neg x_4)$

$(x_1 \vee \neg x_3 \vee x_4)$

$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

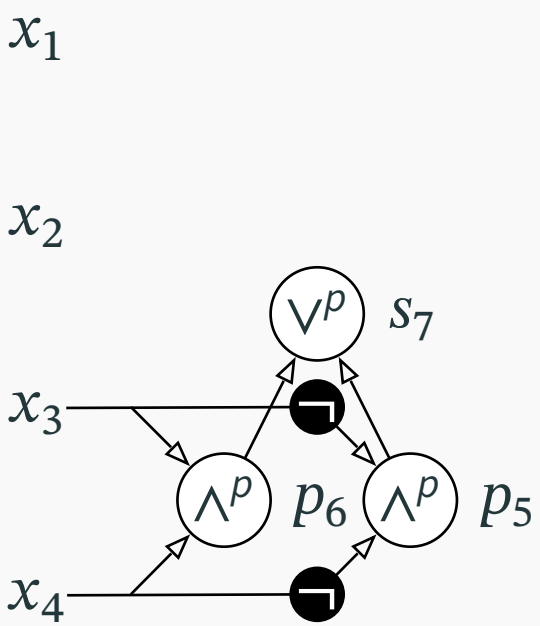
$(p_6 \leftrightarrow (x_3 \wedge x_4))$

$(s_7 \leftrightarrow (p_5 \vee p_6))$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



Partitioned-op. graph

How the checker operates

$(\neg x_1 \vee x_3 \vee \neg x_4)$

$(\neg x_1 \vee \neg x_3 \vee x_4)$

$(\neg x_1 \vee \neg x_2)$

$(x_3 \vee \neg x_4)$

$(x_1 \vee \neg x_3 \vee x_4)$

$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

$(p_6 \leftrightarrow (x_3 \wedge x_4))$

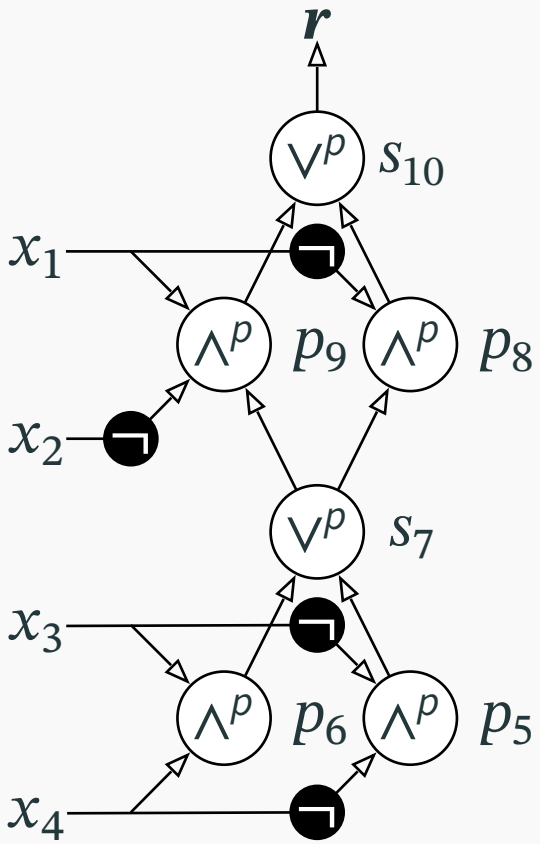
$(s_7 \leftrightarrow (p_5 \vee p_6))$

$\ldots$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



Partitioned-op. graph

How the checker operates

$(\neg x_1 \vee \neg x_3 \vee x_4)$

$(\neg x_1 \vee \neg x_2)$

$(x_3 \vee \neg x_4)$

$(x_1 \vee \neg x_3 \vee x_4)$

$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

$(p_6 \leftrightarrow (x_3 \wedge x_4))$

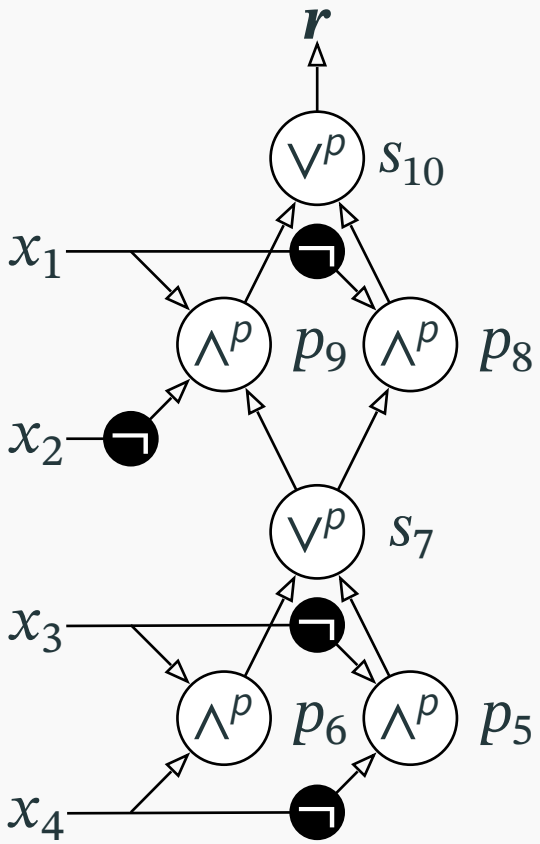
$(s_7 \leftrightarrow (p_5 \vee p_6))$

$\ldots$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



Partitioned-op. graph

How the checker operates

$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

$(p_6 \leftrightarrow (x_3 \wedge x_4))$

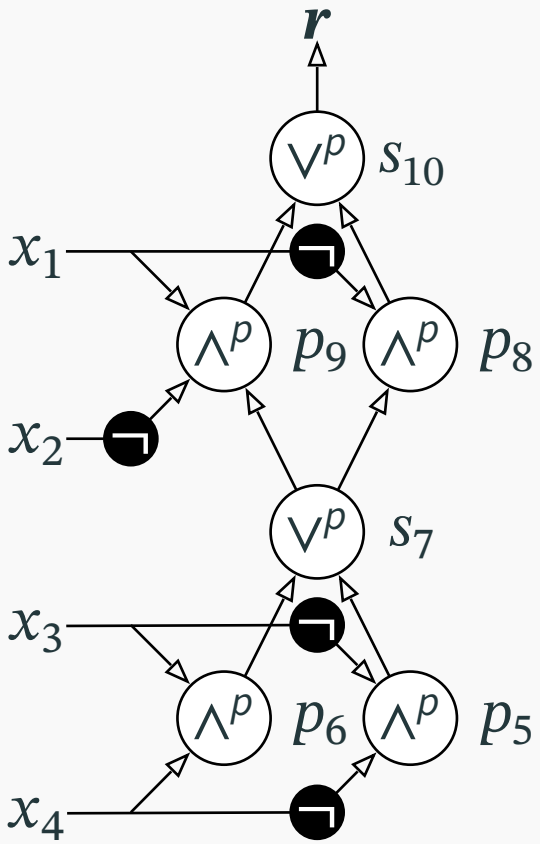
$(s_7 \leftrightarrow (p_5 \vee p_6))$

$\ldots$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



Partitioned-op. graph

How the checker operates

$\boldsymbol{r}$

$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

$(p_6 \leftrightarrow (x_3 \wedge x_4))$

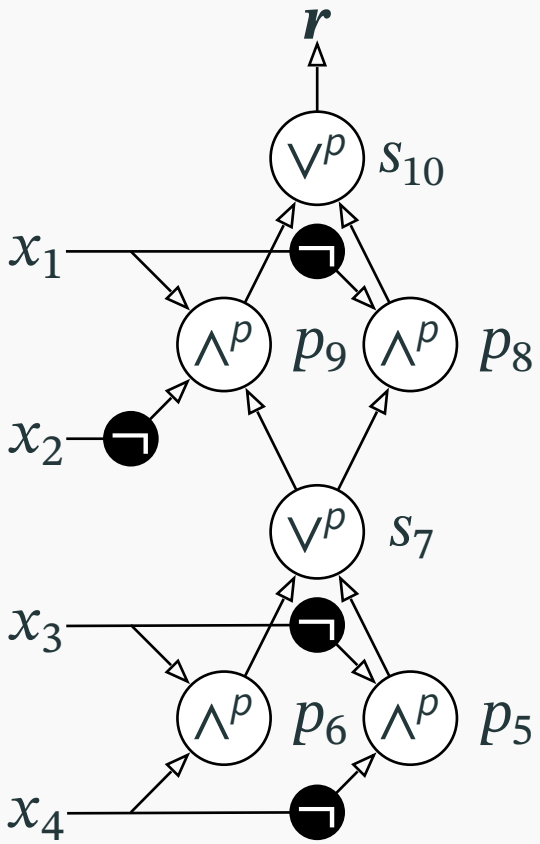
$(s_7 \leftrightarrow (p_5 \vee p_6))$

$\ldots$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



$\varphi \approx$ Partitioned-op. graph

How the checker operates

$\boldsymbol{r}$

$(p_5 \leftrightarrow (\neg x_3 \wedge \neg x_4))$

$(p_6 \leftrightarrow (x_3 \wedge x_4))$

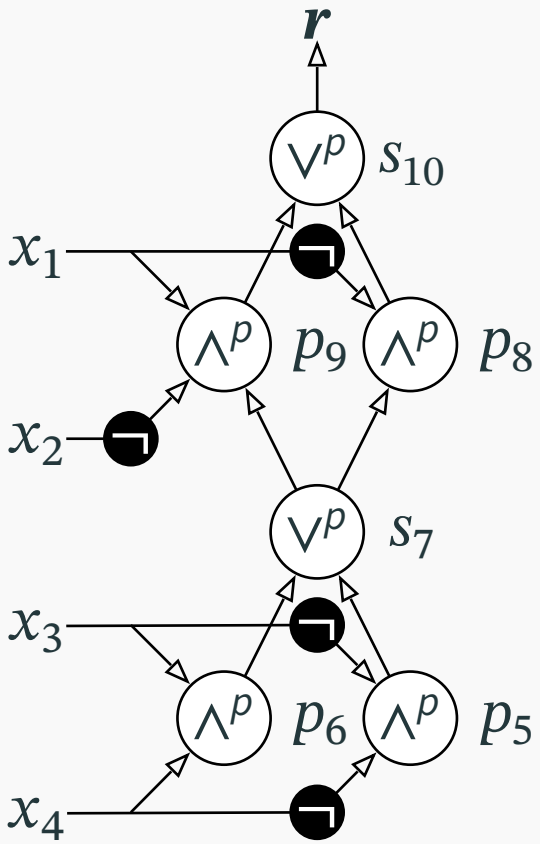
$(s_7 \leftrightarrow (p_5 \vee p_6))$

$\ldots$

Clause database $\approx \varphi$

```
p 5 -3 -4
p 6 3 4
s 7 5 6
...
d 1
...
a 10
```

CPOG certificate



$\varphi \approx$ Partitioned-op. graph

Correctness

Theorem 1. If the proof checker has assembled POG P starting from input formula φ, then φ is logically equivalent to P.

Theorem 2. If the above conditions are met, the value returned by the verified model counter is $|\mathcal{M}(\varphi)|$.

Invariants as 👻 ghost state 👻

```
structure State where
  inputCnf : ICnf
  clauseDb : ClauseDb ClauseIdx
  pog : Pog
```

Invariants as 👻 ghost state 👻

```
structure State where
  inputCnf : ICnf
  clauseDb : ClauseDb ClauseIdx
  pog : Pog

  equivalent_clauseDb : inputCnf ≈ clauseDb
  partitioned : ∀ x : Var, partitioned (pog.get x)
  ...
```

Invariants as 👻 ghost state 👻

```
structure State where
  inputCnf : ICnf
  clauseDb : ClauseDb ClauseIdx
  pog : Pog

  equivalent_clauseDb : inputCnf ≈ clauseDb
  partitioned : ∀ x : Var, partitioned (pog.get x)
  ...

def CheckerM := StateT State (Except CheckerError)
```

Invariants as 👻 ghost state 👻

```
structure State where
  inputCnf : ICnf
  clauseDb : ClauseDb ClauseIdx
  pog : Pog

  equivalent_clauseDb : inputCnf ≈ clauseDb
  partitioned : ∀ x : Var, partitioned (pog.get x)
  ...

def CheckerM := StateT State (Except CheckerError)
```

Intrinsic verification with do notation

```
def CheckerM := StateT State (Except CheckerError)

def addProd (idx : ClauseIdx) (x : Var) (ls : Array ILit) : CheckerM Unit := do
```

Intrinsic verification with `do` notation

```
def CheckerM := StateT State (Except CheckerError)

def addProd (idx : ClauseIdx) (x : Var) (ls : Array ILit) : CheckerM Unit := do
  let st : State ← get
```

Intrinsic verification with do notation

```
def CheckerM := StateT State (Except CheckerError)

def addProd (idx : ClauseIdx) (x : Var) (ls : Array ILit) : CheckerM Unit := do
  let st : State ← get

  -- Compute variable dependencies, ensuring pairwise disjointness.
  let ⟨Ds, _⟩ ← getDepsArray ls
  let ⟨_, _, hDisjoint⟩ ← disjointUnion ls Ds
  -- hDisjoint : ∀ i ≠ j, Vars[i] ∩ Vars[j] = ∅
```

Intrinsic verification with `do` notation

```
def CheckerM := StateT State (Except CheckerError)

def addProd (idx : ClauseIdx) (x : Var) (ls : Array ILit) : CheckerM Unit := do
  let st : State ← get

  -- Compute variable dependencies, ensuring pairwise disjointness.
  let ⟨Ds, _⟩ ← getDepsArray ls
  let ⟨_, _, hDisjoint⟩ ← disjointUnion ls Ds
  -- hDisjoint : ∀ i ≠ j, Vars[i] ∩ Vars[j] = ∅

  let pog' ← addConjToPog st.pog x ls
  set { st with pog := pog', ... }
```

Intrinsic verification with do notation

```
def CheckerM := StateT State (Except CheckerError)

def addProd (idx : ClauseIdx) (x : Var) (ls : Array ILit) : CheckerM Unit := do
  let st : State ← get

  -- Compute variable dependencies, ensuring pairwise disjointness.
  let ⟨Ds, _⟩ ← getDepsArray ls
  let ⟨_, _, hDisjoint⟩ ← disjointUnion ls Ds
  -- hDisjoint : ∀ i ≠ j, Vars[i] ∩ Vars[j] = ∅

  let pog' ← addConjToPog st.pog x ls
  set { st with pog := pog', ... }
```

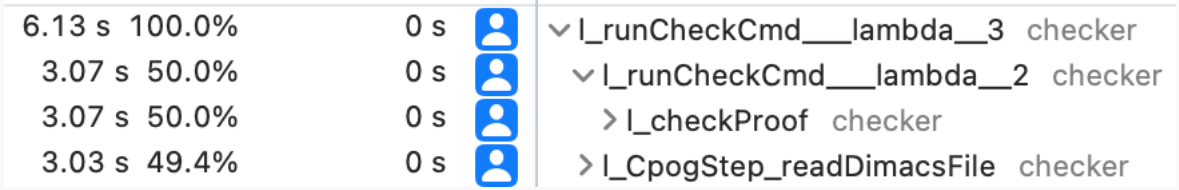
Optimizing the checker

Profiling

- Relatively simple Lean ↔ C correspondence: just look at C symbols
-
-

Profiling

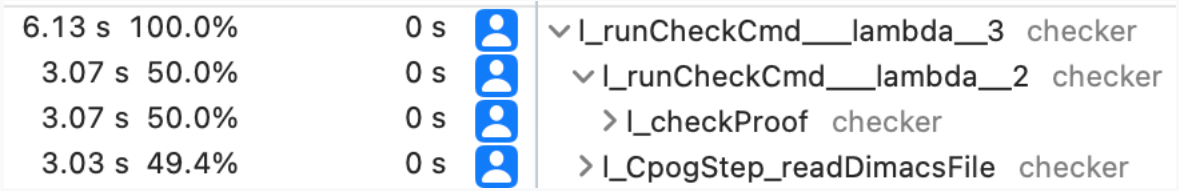
- Relatively simple Lean ↔ C correspondence: just look at C symbols
- Half time overall spent parsing



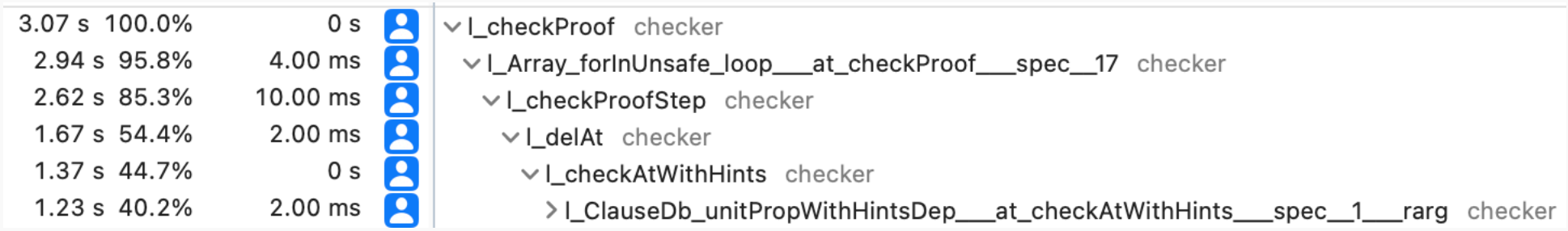
-

Profiling

- Relatively simple Lean ↔ C correspondence: just look at C symbols
- Half time overall spent parsing



- Unit propagation heaviest in proof checking



Functional but in-place (FBIP)

- Purely functional + efficient ⇒ persistent data structures (Chris Okasaki. [9])?
  - ▸
-
-

Functional but in-place (FBIP)

- Purely functional + efficient ⇒ persistent data structures (Chris Okasaki. [9])?
  - No, want in-place mutation
-
-

Functional but in-place (FBIP)

- Purely functional + efficient ⇒ persistent data structures (Chris Okasaki. [9])?
  - No, want in-place mutation
- Functional but in-place cell reuse with reference counting [10, 11]

```
let st : State ← get

-- If refcount(st.pog) = 1, pog' becomes st.pog mutated in-place
let pog' ← addConjToPog st.pog x ls

set { st with pog := pog', ... }
```

-

Functional but in-place (FBIP)

- Purely functional + efficient ⇒ persistent data structures (Chris Okasaki. [9])?
  - No, want in-place mutation
- Functional but in-place cell reuse with reference counting [10, 11]

```
let st : State ← get

-- If refcount(st.pog) = 1, pog' becomes st.pog mutated in-place
let pog' ← addConjToPog st.pog x ls

set { st with pog := pog', ... }
```

- Allows for non-persistent data structures (e.g. flat arrays in place of lists)

Choosing data representation

- Clause database backed by hashmap
-
-
-

Choosing data representation

- Clause database backed by hashmap
- Partitioned-operation graph backed by flat array
-
-

Choosing data representation

- Clause database backed by hashmap
- Partitioned-operation graph backed by flat array
- Fast unit propagation with persistent partial assignments (Cayden R. Codel, Marijn J. H. Heule, and Jeremy Avigad. [3])
-

Choosing data representation

- Clause database backed by hashmap
- Partitioned-operation graph backed by flat array
- Fast unit propagation with persistent partial assignments (Cayden R. Codel, Marijn J. H. Heule, and Jeremy Avigad. [3])
- Ensuring `refcount(..) = 1` unfortunately challenging: no static linearity checks, so look at profile; `lean_copy_expand_array` should not be called a lot

Other optimizations

- Parsing: go lower-level than `String.splitOn`, minimize allocations

```
def bytesToNat (buf : ByteArray) (s e : Nat) : Nat := Id.run do
  let mut ret : Nat := 0
  for i in [s:e] do
    ret := ret * 10 + (buf[i]!.val - 48)
  return ret
```

-

Other optimizations

- Parsing: go lower-level than `String.splitOn`, minimize allocations

```
def bytesToNat (buf : ByteArray) (s e : Nat) : Nat := Id.run do
  let mut ret : Nat := 0
  for i in [s:e] do
    ret := ret * 10 + (buf[i]!.val - 48)
  return ret
```

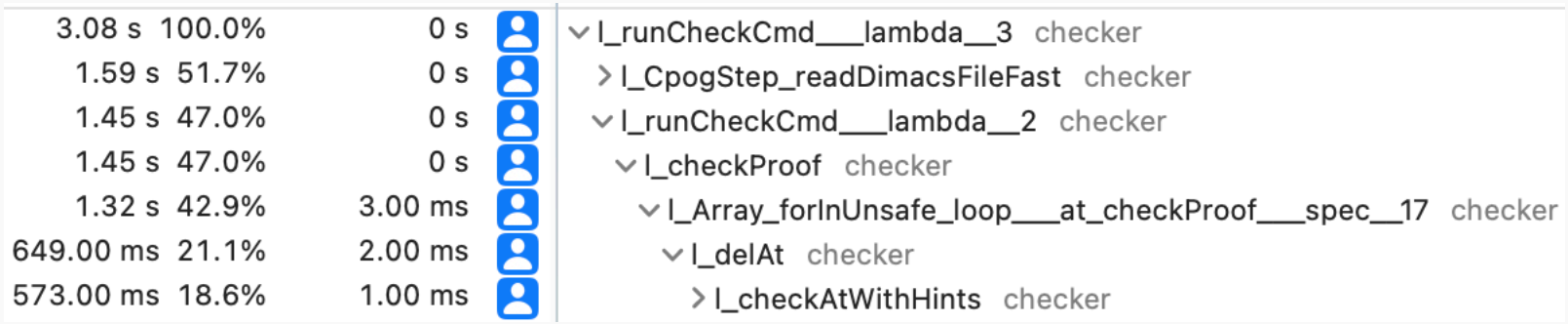
- Tail recursion: not crucial for speed, but ensures constant stack usage

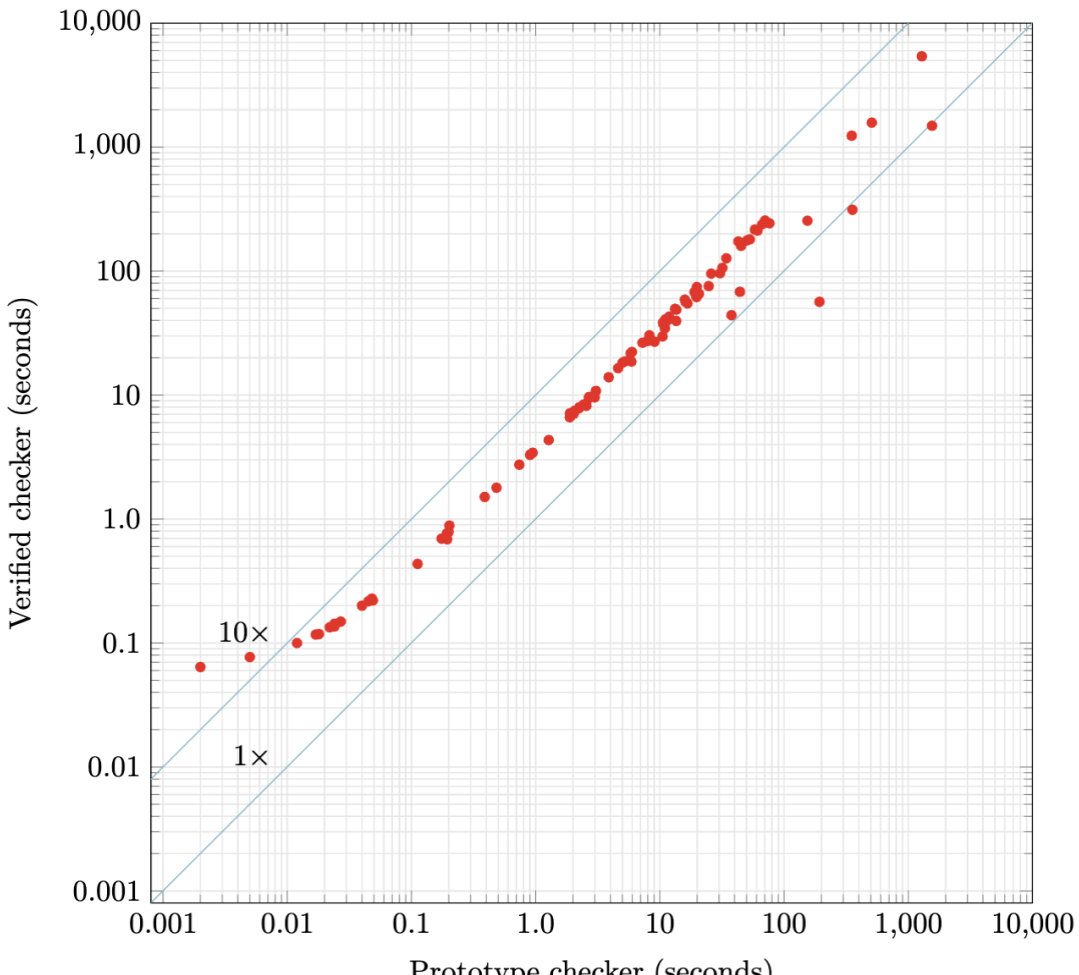
Other optimizations

- Parsing: go lower-level than `String.splitOn`, minimize allocations

```
def bytesToNat (buf : ByteArray) (s e : Nat) : Nat := Id.run do
  let mut ret : Nat := 0
  for i in [s:e] do
    ret := ret * 10 + (buf[i]!.val - 48)
  return ret
```

- Tail recursion: not crucial for speed, but ensures constant stack usage




10,000
1,000
100
10
1.0
0.1
0.01
0.001
Verified checker (seconds)
10×
1×
0.001
0.01
0.1
1.0
10
100
1,000
10,000
Prototype checker (seconds)

Lean/C ≈ 3.54x*

Future outlook

Outlook on verified programming in Lean

- With dependent types, verification feels natural
-
-

Outlook on verified programming in Lean

- With dependent types, verification feels natural
- More infrastructure and proof automation in place now than in 2022
  - Verified standard library being built: no longer have to spend half the time proving your own hashmap correct!
-

Outlook on verified programming in Lean

- With dependent types, verification feels natural
- More infrastructure and proof automation in place now than in 2022
  - Verified standard library being built: no longer have to spend half the time proving your own hashmap correct!
- Better verification for `do` notation upcoming, perhaps?
  - No way to state loop postconditions as of now

```
for i in [0:n] with invariant P i do -- Not real syntax!
  ...
-- Want a proof of `P (n - 1)` here
```

Outlook on efficient functional programming

- Linearity is crucial, enabling in-place mutation without `ST` [6, 7]
-
-
-
-

Outlook on efficient functional programming

- Linearity is crucial, enabling in-place mutation without `ST` [6, 7]
- With reference counting: static linearity checking useful, but not necessary
-
-
-

Outlook on efficient functional programming

- Linearity is crucial, enabling in-place mutation without `ST` [6, 7]
- With reference counting: static linearity checking useful, but not necessary
- With garbage collection: static checking necessary
  - Linear Haskell [1]
  - Oxidized OCaml [8]
-
-

Outlook on efficient functional programming

- Linearity is crucial, enabling in-place mutation without `ST` [6, 7]
- With reference counting: static linearity checking useful, but not necessary
- With garbage collection: static checking necessary
  - Linear Haskell [1]
  - Oxidized OCaml [8]
- Linear dependent types not a solved problem
-

Outlook on efficient functional programming

- Linearity is crucial, enabling in-place mutation without `ST` [6, 7]
- With reference counting: static linearity checking useful, but not necessary
- With garbage collection: static checking necessary
  - Linear Haskell [1]
  - Oxidized OCaml [8]
- Linear dependent types not a solved problem
- Fast reference counting with `mimalloc` (Lean, Koka) [5]

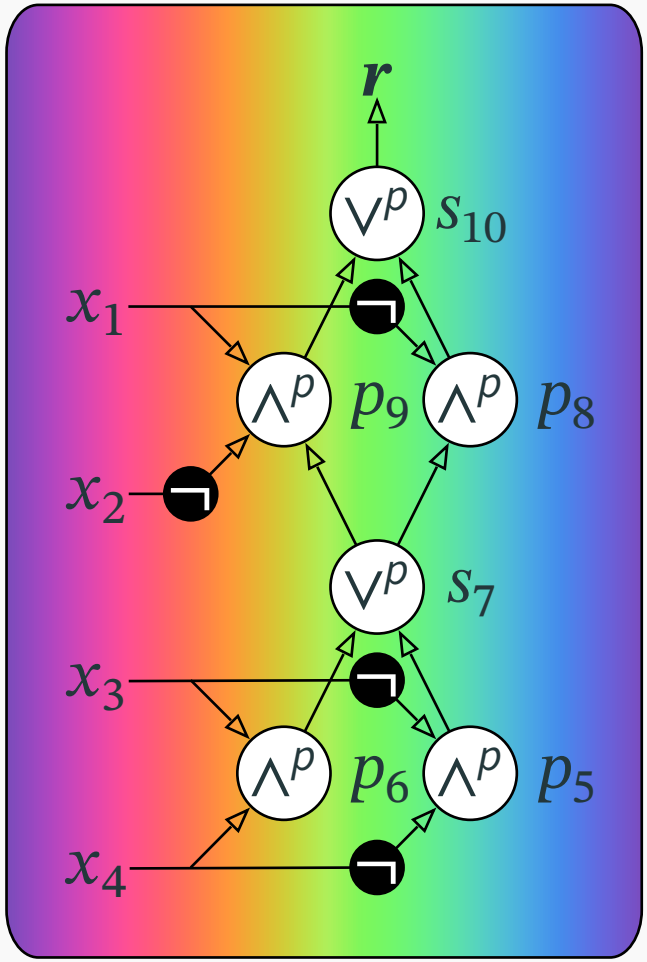
Thank you

arxiv.org/abs/2501.12906

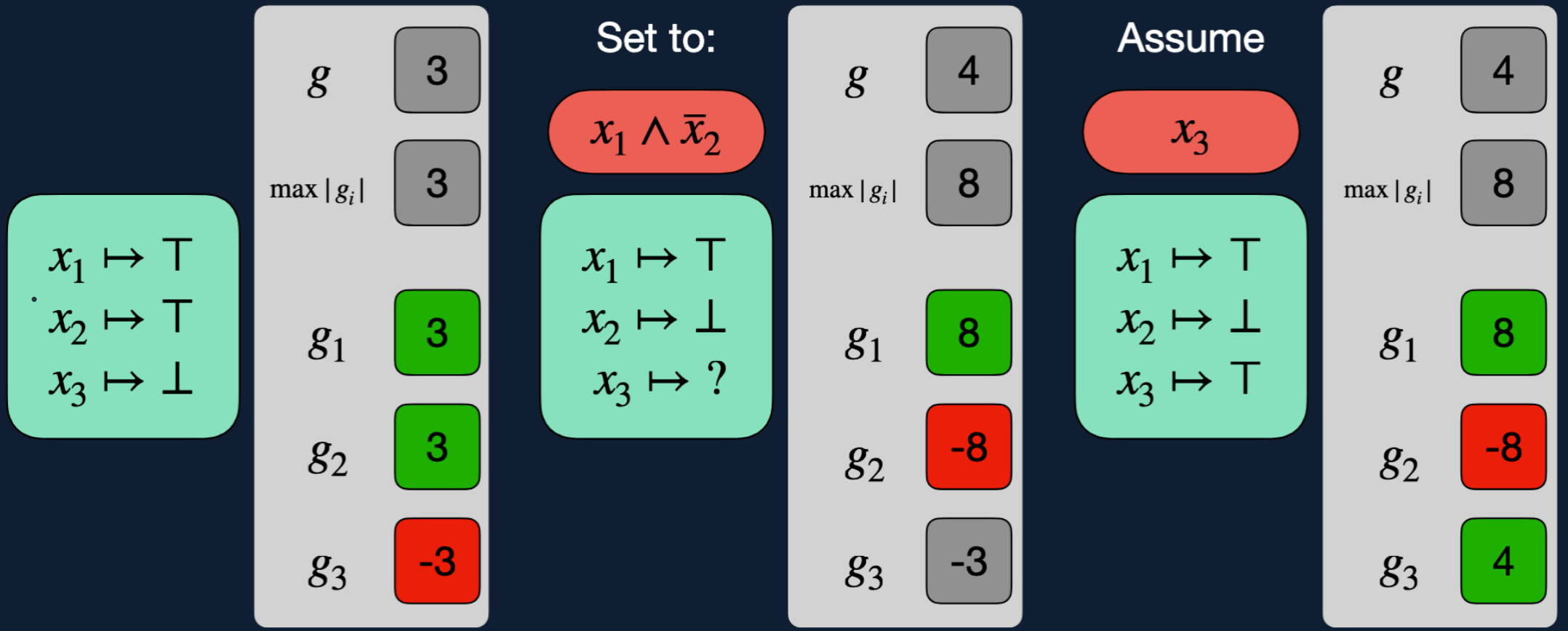
github.com/rebryant/cpog

voidma.in | mathstodon.xyz/@Vtec234

How would you write this in Haskell?



Implementing truth assignments [3]



Bibliography

[1] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL (2018), 1–29. https://doi.org/10.1145/3158093

[2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2021. *Handbook of Satisfiability - Second Edition*. IOS Press. https://doi.org/10.3233/FAIA336

[3] Cayden R. Codel, Marijn J. H. Heule, and Jeremy Avigad. 2024. Verified Substiution Redundancy Checking. In *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design - FMCAD 2024 (Formal Methods in Computer-Aided Design)*, 2024. TU Wien Academic Press, Vienna, Austria, 186–196. https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_24

[4] Johannes Klaus Fichte, Markus Hecher, and Florim Hamiti. 2021. The Model Counting Competition 2020. *ACM J. Exp. Algorithmics* 26, (2021), 1–26. https://doi.org/10.1145/3459080

[5] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings (Lecture Notes in Computer Science)*, 2019. Springer, 244–265. https://doi.org/10.1007/978-3-030-34175-6_13

[6] Anton Lorenzen and Daan Leijen. 2022. Reference counting with frame limited reuse. *Proc. ACM Program. Lang.* 6, ICFP (2022), 357–380. https://doi.org/10.1145/3547634

[7] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP\({^2}\) : Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP (2023), 275–304. https://doi.org/10.1145/3607840

[8] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP (2024), 485–514. https://doi.org/10.1145/3674642

[9] Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.

[10] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*, 2021. Association for Computing Machinery, Virtual, Canada, 96–111. https://doi.org/10.1145/3453483.3454032

[11] Sebastian Ullrich and Leonardo de Moura. 2019. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. *CoRR* (2019). Retrieved from http://arxiv.org/abs/1908.05647