

Equality Saturation: Term Extraction and an Application to Network Synthesis

**General Exam: Deyuan (Mike) He
April 17, 2024**

Examination Committee

Prof.Aarti Gupta (Advisor)

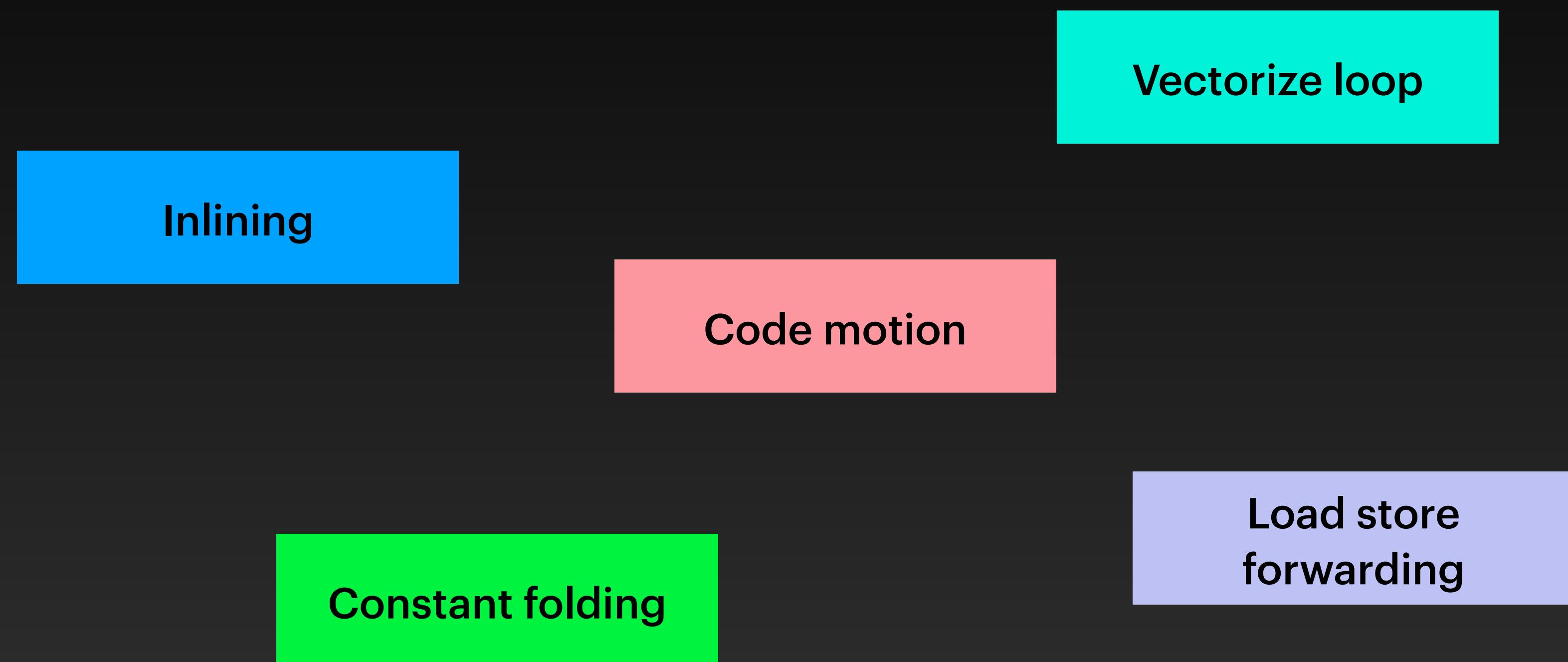
Prof.Andrew Appel

Prof.Mae Milano

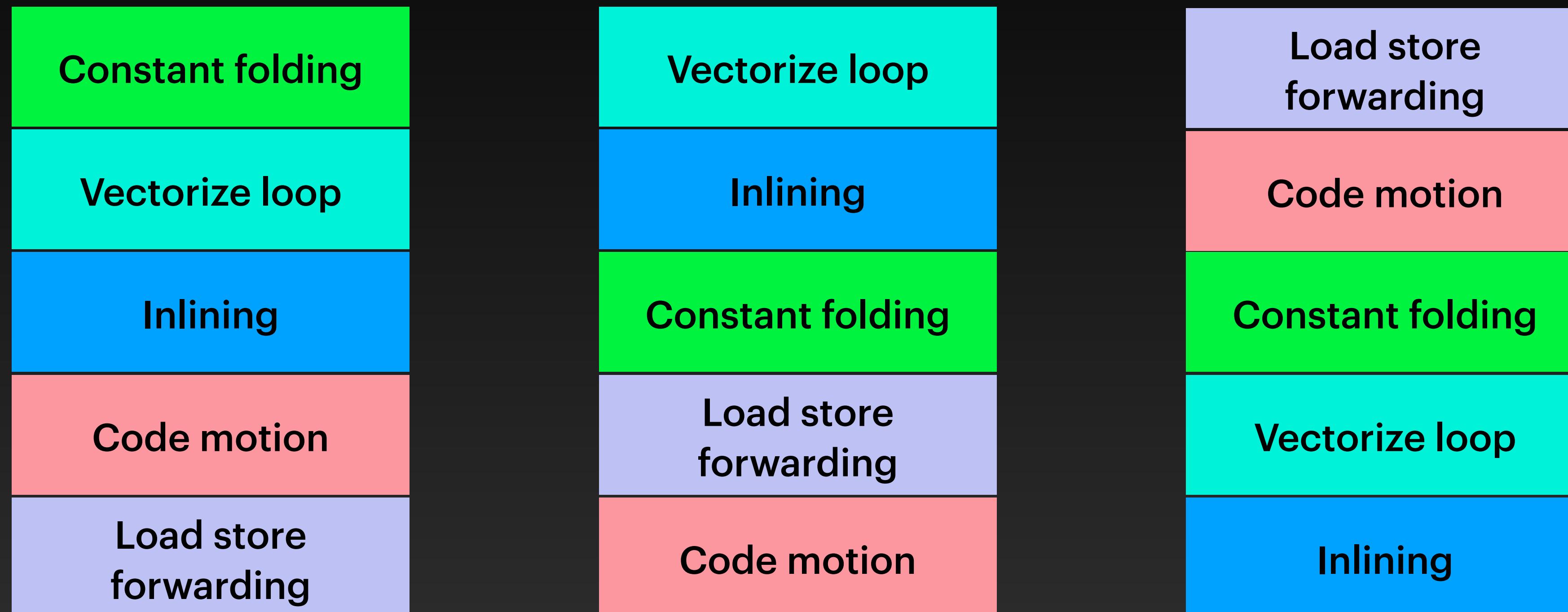
Outline

1. Brief introduction to equality saturation
2. Term Extraction for equality saturation **(Part A)**
3. Applying equality saturation for network resource synthesis **(Part B)**
4. (If time permits) Ongoing project of invariant synthesis for distributed systems

Compiler optimizations are hard to design



Compiler optimizations are hard to design



*Which order to choose?
Phase Ordering Problem*

Compiler optimizations are hard to design

```

/* Description of pass structure
Copyright (C) 1987-2024 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free
Software Foundation; either version 3, or (at your option) any later
version.

GCC is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3. If not see
<http://www.gnu.org/licenses/>. */

/*
Macros that should be defined when using this file:
INSERT_PASSES_AFTER (PASS)
PUSH_INSERT_PASSES_WITHIN (PASS)
POP_INSERT_PASSES ()
NEXT_PASS (PASS)
TERMINATE_PASS_LIST (PASS)
*/
/* All passes needed to lower the function into shape optimizers can
operate on. These passes are always run first on the function, but
backend might produce already lowered functions that are not processed
by these passes. */
INSERT_PASSES_AFTER (all_lowering_passes)
NEXT_PASS (pass_warn_unused_result);
NEXT_PASS (pass_diagnose_omp_blocks);
NEXT_PASS (pass_diagnose_tm_blocks);
NEXT_PASS (pass_omp_oacc_kernels_decompose);
NEXT_PASS (pass_lower_omp);
NEXT_PASS (pass_lower_cf);
NEXT_PASS (pass_lower_tm);
NEXT_PASS (pass_refactor_eh);
NEXT_PASS (pass_lower_eh);
NEXT_PASS (pass_coroutine_lower_builtin);
NEXT_PASS (pass_build_cfg);
NEXT_PASS (pass_warn_function_return);
NEXT_PASS (pass_coroutine_early_expand_ifns);
NEXT_PASS (pass_expand_omp);
NEXT_PASS (pass_build_cgraph_edges);
TERMINATE_PASS_LIST (all_lowering_passes)

/* Interprocedural optimization passes. */
INSERT_PASSES_AFTER (all_small_ipa_passes)
NEXT_PASS (pass_ipa_free_lang_data);
NEXT_PASS (pass_ipa_function_and_variable_visibility);
NEXT_PASS (pass_ipa_strub_mode);
NEXT_PASS (pass_build_ssa_passes);
PUSH_INSERT_PASSES_WITHIN (pass_build_ssa_passes)
NEXT_PASS (pass_fixup_cfg);
NEXT_PASS (pass_build_ssa);
NEXT_PASS (pass_walloc, /*strict_mode_p=*/true);
NEXT_PASS (pass_warn_printf);
NEXT_PASS (pass_warn_nonnull_compare);
NEXT_PASS (pass_early_warn_uninitialized);
NEXT_PASS (pass_warn_access, /*early=*/true);
NEXT_PASS (pass_ubsan);
NEXT_PASS (pass_nothrow);
NEXT_PASS (pass_rebuild_cgraph_edges);
POP_INSERT_PASSES ()

NEXT_PASS (pass_local_optimization_passes);
PUSH_INSERT_PASSES_WITHIN (pass_local_optimization_passes)
NEXT_PASS (pass_fixup_cfg);
NEXT_PASS (pass_rebuild_cgraph_edges);
NEXT_PASS (pass_local_fn_summary);
NEXT_PASS (pass_early_inline);
NEXT_PASS (pass_warn_recursion);
NEXT_PASS (pass_all_early_optimizations);
PUSH_INSERT_PASSES_WITHIN (pass_all_early_optimizations)
NEXT_PASS (pass_remove_cgraph callees);
NEXT_PASS (pass_early_object_sizes);
/* Don't record nonzero bits before IPA to avoid
using too much memory. */
NEXT_PASS (pass_ccp, false /* nonzero_p */);
/* After CCP we rewrite no longer addressed locals into SSA
form if possible. */
NEXT_PASS (pass_forprop);
NEXT_PASS (pass_early_thread_jumps, /*first=*/true);
NEXT_PASS (pass_sra_early);
/* pass_build_elias is a dummy pass that ensures that we
execute TODO_rebuild_alias at this point. */
NEXT_PASS (pass_build_elias);
/* Do phiprop before FRE so we optimize std::min and std::max well.
NEXT_PASS (pass_phiprop);
NEXT_PASS (pass_fre, true /* may_iterate */);
NEXT_PASS (pass_early_vrp);
NEXT_PASS (pass_merge_phi);
NEXT_PASS (pass_dse);
NEXT_PASS (pass_cd_dce, false /* update_address_taken_p */);
NEXT_PASS (pass_phiprop, true /* early_p */);
NEXT_PASS (pass_tайл_recursion);
NEXT_PASS (pass_if_to_switch);
NEXT_PASS (pass_convert_switch);
NEXT_PASS (pass_cleanup_eh);
NEXT_PASS (pass_scoppy);
NEXT_PASS (pass_profile);
NEXT_PASS (pass_local_pure_const);
NEXT_PASS (pass_modref);
/* Split functions creates parts that are not run through
early optimizations again. It is thus good idea to do this
late. */
NEXT_PASS (pass_split_functions);
NEXT_PASS (pass_strip_predict_hints, true /* early_p */);
POP_INSERT_PASSES ()
NEXT_PASS (pass_release_ssa_names);
NEXT_PASS (pass_rebuild_cgraph_edges);
NEXT_PASS (pass_local_fn_summary);
POP_INSERT_PASSES ()

NEXT_PASS (pass_ipa_remove_symbols);
NEXT_PASS (pass_ipa_strub);

INSERT_PASSES_AFTER (all_regular_ipa_passes)
NEXT_PASS (pass_analyzer);
NEXT_PASS (pass_ipa_ode);
NEXT_PASS (pass_ipa_whole_program_visibility);
NEXT_PASS (pass_ipa_profile);
NEXT_PASS (pass_ipa_icf);
NEXT_PASS (pass_ipa_devirt);
NEXT_PASS (pass_ipa_cp);
NEXT_PASS (pass_ipa_sra);
NEXT_PASS (pass_ipa_cdtor_merge);
NEXT_PASS (pass_ipa_fn_summary);
NEXT_PASS (pass_ipa_inline);
NEXT_PASS (pass_ipa_pure_const);
NEXT_PASS (pass_ipa_modref);
NEXT_PASS (pass_ipa_free_fn_summary, false /* small_p */);
NEXT_PASS (pass_ipa_reference);
/* This pass needs to be scheduled after any IP code duplication. */
NEXT_PASS (pass_ipa_single_use);
/* Comdat privatization come last, as direct references to comdat local
symbols are not allowed outside of the comdat group. Privatizing early
would result in missed optimizations due to this restriction. */
NEXT_PASS (pass_ipa_comdat);
TERMINATE_PASS_LIST (all_regular_ipa_passes)

/* Simple IPA passes executed after the regular passes. In WHOPR mode the
passes are executed after partitioning and thus see just parts of the
compiled unit. */
INSERT_PASSES_AFTER (all_late_ipa_passes)
NEXT_PASS (pass_ipa_pta);
NEXT_PASS (pass_omp_simd_clone);
TERMINATE_PASS_LIST (all_late_ipa_passes)

/* These passes are run after IPA passes on every function that is being
output to the assembler file. */
INSERT_PASSES_AFTER (all_passes)
NEXT_PASS (pass_fixup_cfg);
NEXT_PASS (pass_lower_eh_dispatch);
NEXT_PASS (pass_oacc_loop_designation);
NEXT_PASS (pass_omp_oacc_neuter_broadcast);
NEXT_PASS (pass_oacc_device_lower);
NEXT_PASS (pass_omp_device_lower);
NEXT_PASS (pass_omp_target_link);
NEXT_PASS (pass_adjust_alignment);
NEXT_PASS (pass_harden_control_flow_redundancy);
NEXT_PASS (pass_all_optimizations);
PUSH_INSERT_PASSES_WITHIN (pass_all_optimizations)
NEXT_PASS (pass_remove_cgraph callees);
/* Initial scalar cleanups before alias computation.
They ensure memory accesses are not indirect wherever possible. */
NEXT_PASS (pass_strip_predict_hints, false /* early_p */);
NEXT_PASS (pass_ccp, true /* nonzero_p */);
/* After CCP we rewrite no longer addressed locals into SSA
form if possible. */
NEXT_PASS (pass_object_sizes);
NEXT_PASS (pass_post_ipa_warn);
/* Must run before loop unrolling. */
NEXT_PASS (pass_warn_access, /*early=*/true);
/* Profile count may overflow as a result of inlining very large
loop nests. This pass should run before any late pass that makes
use of profile. */
NEXT_PASS (pass_rebuild_frequencies);
NEXT_PASS (pass_complete_unroll);
NEXT_PASS (pass_backprop);
NEXT_PASS (pass_phiprop);
NEXT_PASS (pass_forprop);
/* pass_build_alias is a dummy pass that ensures that we
execute TODO_rebuild_alias at this point. */
NEXT_PASS (pass_build_alias);
NEXT_PASS (pass_return_slot);
NEXT_PASS (pass_fre, true /* may_iterate */);
NEXT_PASS (pass_merge_phi);
NEXT_PASS (pass_thread_jumps_full, /*first=*/true);
NEXT_PASS (pass_vrp, false /* final_p */);
NEXT_PASS (pass_dse);
NEXT_PASS (pass_dce);
/* pass_stdarg is always run and at this point we execute
TODO_remove_unused_locals to prune CLOBBERs of dead
variables which are otherwise a churn on alias walkings. */
NEXT_PASS (pass_stdarg);
NEXT_PASS (pass_call_dce);
NEXT_PASS (pass_cselim);
NEXT_PASS (pass_copy_prop);
NEXT_PASS (pass_tree_ifcombine);
NEXT_PASS (pass_merge_phi);
NEXT_PASS (pass_phiopt, false /* early_p */);
NEXT_PASS (pass_tайл_recursion);
NEXT_PASS (pass_ch);
NEXT_PASS (pass_lower_complex);
NEXT_PASS (pass_lower_bitint);
NEXT_PASS (pass_sra);
/* The dom pass will also resolve all __builtin_constant_p calls
that are still there to 0. This has to be done after some
propagations have already run, but before some more dead code
is removed, and this place fits nicely. Remember this when
trying to move or duplicate pass_dominator somewhere earlier. */
NEXT_PASS (pass_thread_jumps, /*first=*/true);
NEXT_PASS (pass_dominator, true /* may_peel_loop_headers_p */);
/* Threading can leave many const/copy propagations in the IL.
Clean them up. Failure to do so well can lead to false
positives from warnings for erroneous code. */
NEXT_PASS (pass_copy_prop);
/* Identify paths that should never be executed in a conforming
program and isolate those paths. */
NEXT_PASS (pass_isolate_erroneous_paths);
NEXT_PASS (pass_reassoc, true /* early_p */);
NEXT_PASS (pass_dce);
NEXT_PASS (pass_forprop);
NEXT_PASS (pass_phiopt, false /* early_p */);
NEXT_PASS (pass_ccp, true /* nonzero_p */);
/* After CCP we rewrite no longer addressed locals into SSA
form if possible. */
NEXT_PASS (pass_expand_powcabs);
NEXT_PASS (pass_optimize_bswap);
NEXT_PASS (pass_laddress);
NEXT_PASS (pass_lim);
NEXT_PASS (pass_walloc, false);
NEXT_PASS (pass_prel);
NEXT_PASS (pass_sink_code, false /* unsplit edges */);
NEXT_PASS (pass_sancov);
NEXT_PASS (pass_asan);
NEXT_PASS (pass_tsan);

NEXT_PASS (pass_treeLoop);
PUSH_INSERT_PASSES_WITHIN (pass_treeLoop)
/* Before loop_init we rewrite no longer addressed locals into SSA
form if possible. */
NEXT_PASS (pass_tree_loop_init);
NEXT_PASS (pass_tree_unswitch);
NEXT_PASS (pass_loop_split);
NEXT_PASS (pass_scev_cprop);
NEXT_PASS (pass_loop_versioning);
NEXT_PASS (pass_loop_jam);
/* All unswitching, final value replacement and splitting can expose
empty loops. Remove them now. */
NEXT_PASS (pass_cd_dce, false /* update_address_taken_p */);
NEXT_PASS (pass_iv_canon);
NEXT_PASS (pass_loop_distribution);
NEXT_PASS (pass_linterchange);
NEXT_PASS (pass_copy_prop);
NEXT_PASS (pass_graphite);
PUSH_INSERT_PASSES_WITHIN (pass_graphite)
NEXT_PASS (pass_graphite_transforms);
NEXT_PASS (pass_lim);
NEXT_PASS (pass_copy_prop);
NEXT_PASS (pass_dce);
POP_INSERT_PASSES ()
NEXT_PASS (pass_parallelize_loops, false /* oacc_kernels_p */);
NEXT_PASS (pass_expand_omp_ssa);
NEXT_PASS (pass_ch_vec);
NEXT_PASS (pass_if_conversion);
/* pass_vectorize must immediately follow pass_if_conversion.
Please do not add any other passes in between. */
NEXT_PASS (pass_vectorize);
PUSH_INSERT_PASSES_WITHIN (pass_vectorize)
NEXT_PASS (pass_dce);
POP_INSERT_PASSES ()
NEXT_PASS (pass_predcom);
NEXT_PASS (pass_complete_unroll);
NEXT_PASS (pass_pre_slp_scalar_cleanup);
PUSH_INSERT_PASSES_WITHIN (pass_pre_slp_scalar_cleanup)
NEXT_PASS (pass_fre, false /* may_iterate */);
NEXT_PASS (pass_dse);
POP_INSERT_PASSES ()
NEXT_PASS (pass_slp_vectorize);
NEXT_PASS (pass_loop_prefetch);
/* Run IVOPTS after the last pass that uses data-reference analysis
as that doesn't handle TARGET_MEM_REFs. */
NEXT_PASS (pass_iv_optimize);
NEXT_PASS (pass_lim);
NEXT_PASS (pass_tree_loop_done);
POP_INSERT_PASSES ()
/* Pass group that runs when pass_tree_loop is disabled or there
are no loops in the function. */
NEXT_PASS (pass_tree_no_loop);
PUSH_INSERT_PASSES_WITHIN (pass_tree_no_loop)
NEXT_PASS (pass_slp_vectorize);
POP_INSERT_PASSES ()
NEXT_PASS (pass_simuid_cleanup);
NEXT_PASS (pass_lower_vector_ssa);
NEXT_PASS (pass_lower_switch);
NEXT_PASS (pass_cse_sincos);
NEXT_PASS (pass_cse_reciprocals);
NEXT_PASS (pass_reassoc, false /* early_p */);
NEXT_PASS (pass_strength_reduction);
NEXT_PASS (pass_split_paths);
NEXT_PASS (pass_tracer);
NEXT_PASS (pass_fre, false /* may_iterate */);
/* After late FRE we rewrite no longer addressed locals into SSA
form if possible. */
NEXT_PASS (pass_thread_jumps, /*first=*/false);
NEXT_PASS (pass_dominator, false /* may_peel_loop_headers_p */);
NEXT_PASS (pass_strlen);
NEXT_PASS (pass_thread_jumps_full, /*first=*/false);
NEXT_PASS (pass_vrp, true /* final_p */);
/* Run CCP to compute alignment and nonzero bits. */
NEXT_PASS (pass_ccp, true /* nonzero_p */);
NEXT_PASS (pass_warn_restrict);
NEXT_PASS (pass_dse);
NEXT_PASS (pass_dce, true /* update_address_taken_p */);
/* After late DCE we rewrite no longer addressed locals into SSA
form if possible. */
NEXT_PASS (pass_forprop);
NEXT_PASS (pass_sink_code, true /* unsplit edges */);
NEXT_PASS (pass_phiopt, false /* early_p */);
NEXT_PASS (pass_fold_builtins);
NEXT_PASS (pass_optimize_widening_mul);
NEXT_PASS (pass_store_merging);
/* If DCE is not run before checking for uninitialized uses,
we may get false warnings (e.g., testsuite/gcc.dg/unitn-5.c).
However, this also causes us to misdiagnose cases that should be
real warnings (e.g., testsuite/gcc.dg/pr18501.c). */
NEXT_PASS (pass_cd_dce, false /* update_address_taken_p */);
NEXT_PASS (pass_scoppy);
NEXT_PASS (pass_tайл_calls);
/* Split critical edges before late uninitialized warning to reduce the
number of false positives from it. */
NEXT_PASS (pass_split_crit_edges);
NEXT_PASS (pass_late_warn_uninitialized);
NEXT_PASS (pass_local_pure_const);
NEXT_PASS (pass_modref);
/* unprop replaces constants by SSA names. This makes analysis harder
and thus it should be run last. */
NEXT_PASS (pass_unprop);

POP_INSERT_PASSES ()
NEXT_PASS (pass_all_optimizations_g);
PUSH_INSERT_PASSES_WITHIN (pass_all_optimizations_g)
/* The idea is that with -O0 we do not perform any IPA optimization
so post-IPA work should be restricted to semantically required
passes and all optimization work is done early. */
NEXT_PASS (pass_remove_cgraph callees);
NEXT_PASS (pass_strip_predict_hints, false /* early_p */);
/* Lower remaining pieces of GIMPLE. */
NEXT_PASS (pass_lower_complex);
NEXT_PASS (pass_lower_bitint);
NEXT_PASS (pass_lower_vector_ssa);
NEXT_PASS (pass_lower_switch);
/* Perform simple scalar cleanup which is constant/copy propagation. */
NEXT_PASS (pass_ccp, true /* nonzero_p */);
NEXT_PASS (pass_post_ipa_warn);
NEXT_PASS (pass_object_sizes);
/* Fold remaining builtins. */
NEXT_PASS (pass_fold_builtins);
NEXT_PASS (pass_strlen);

```

500+ LoC to define the order

Compiler optimizations are hard to design

Observation: program transformations are *destructive*

$$\frac{\begin{array}{c} ?V \times 2 \\ (X \times 2) \div 2 \end{array}}{\begin{array}{c} ?V \times 2 \rightarrow ?V \ll 1 \\ \hline ?V \mapsto X \end{array}} \quad (X \ll 1) \div 2$$

$$\begin{array}{l} (?X \times ?Y) \div ?Z \rightarrow ?X \times (?Y \div ?Z) \\ ?X \div ?X \rightarrow 1 \\ ?X \times 1 \rightarrow ?X \end{array}$$

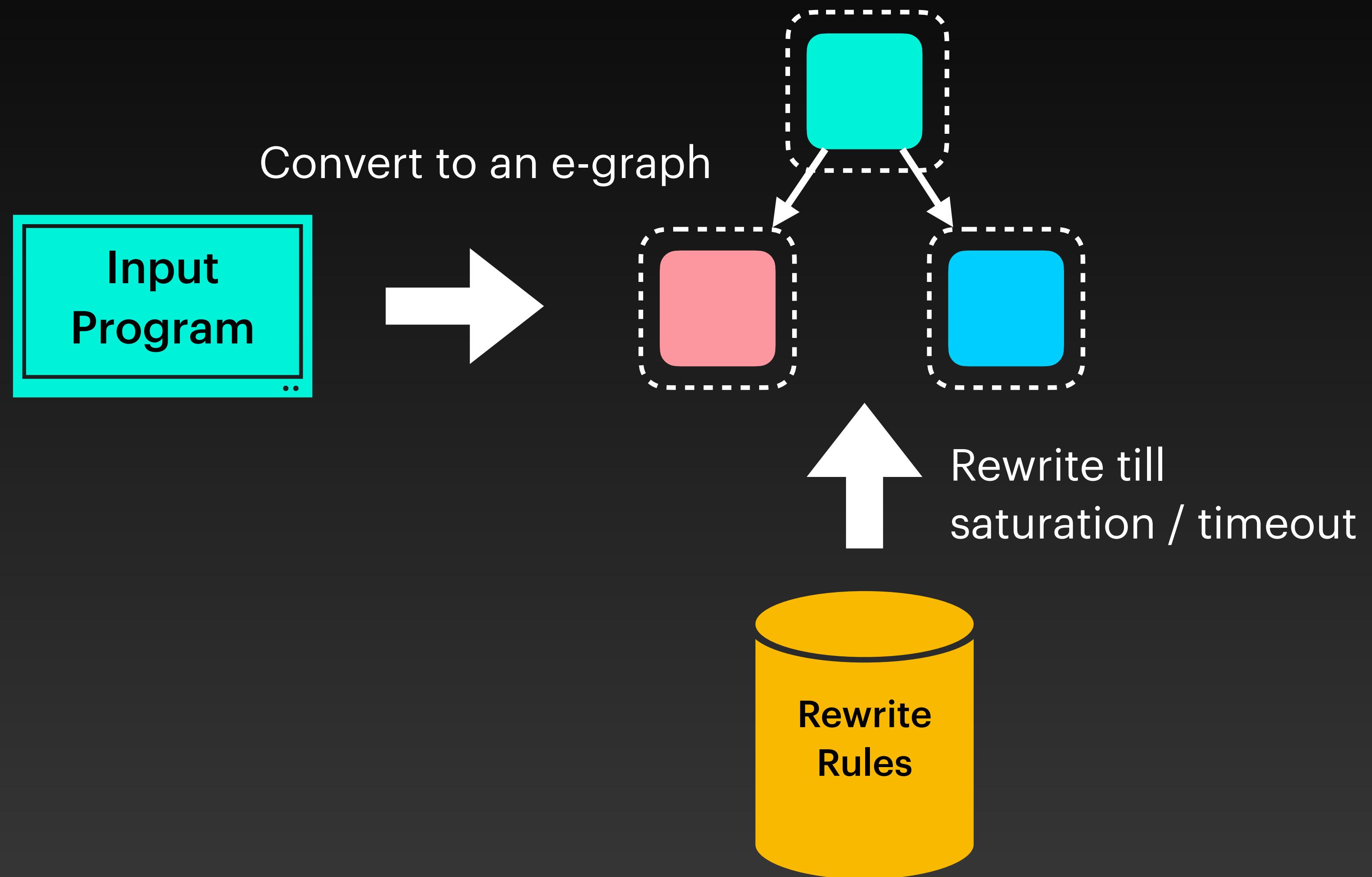
$$(X \times 2) \div 2$$



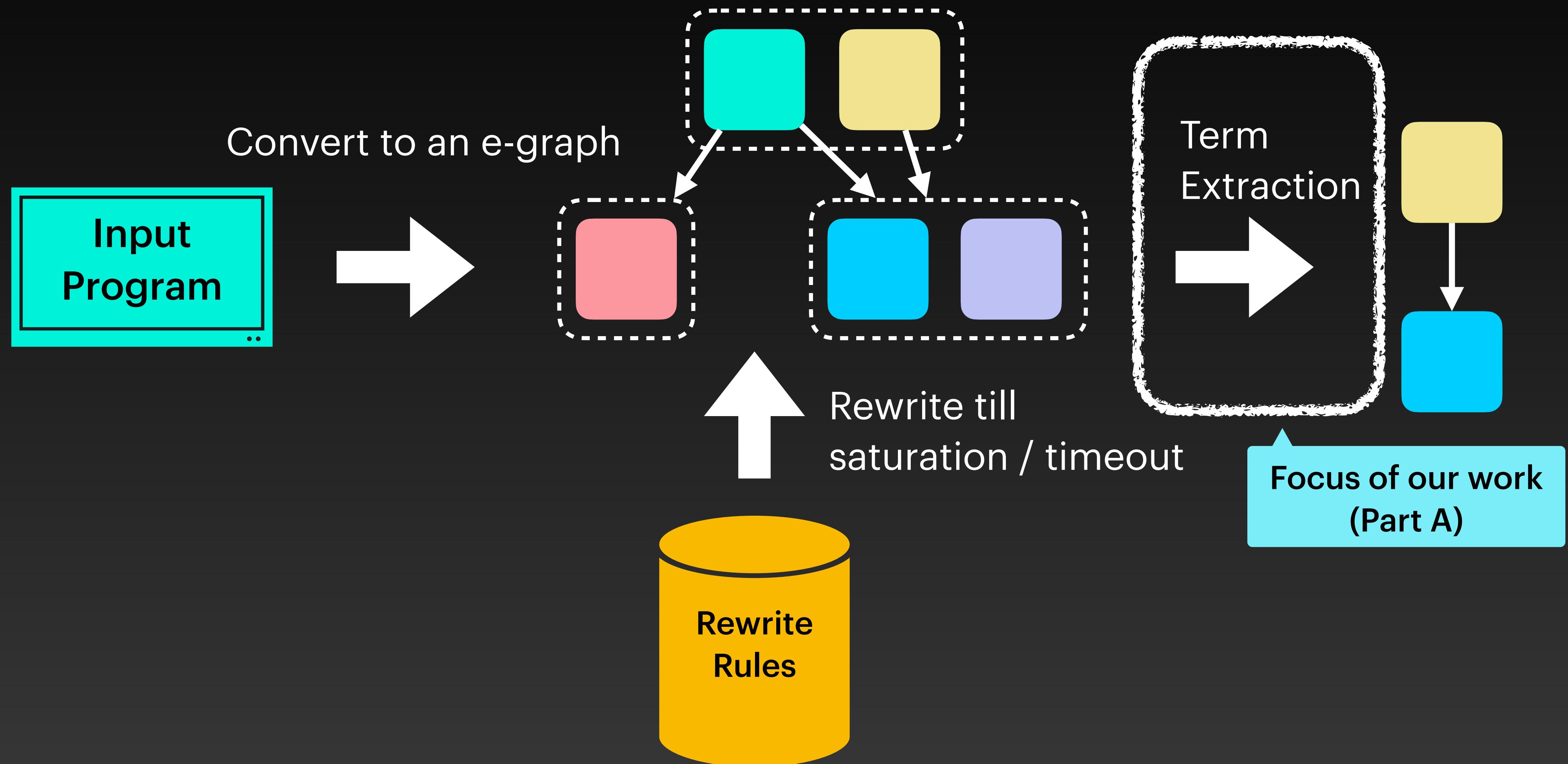
Equality Saturation

Non-destructive rewriting

Equality Saturation

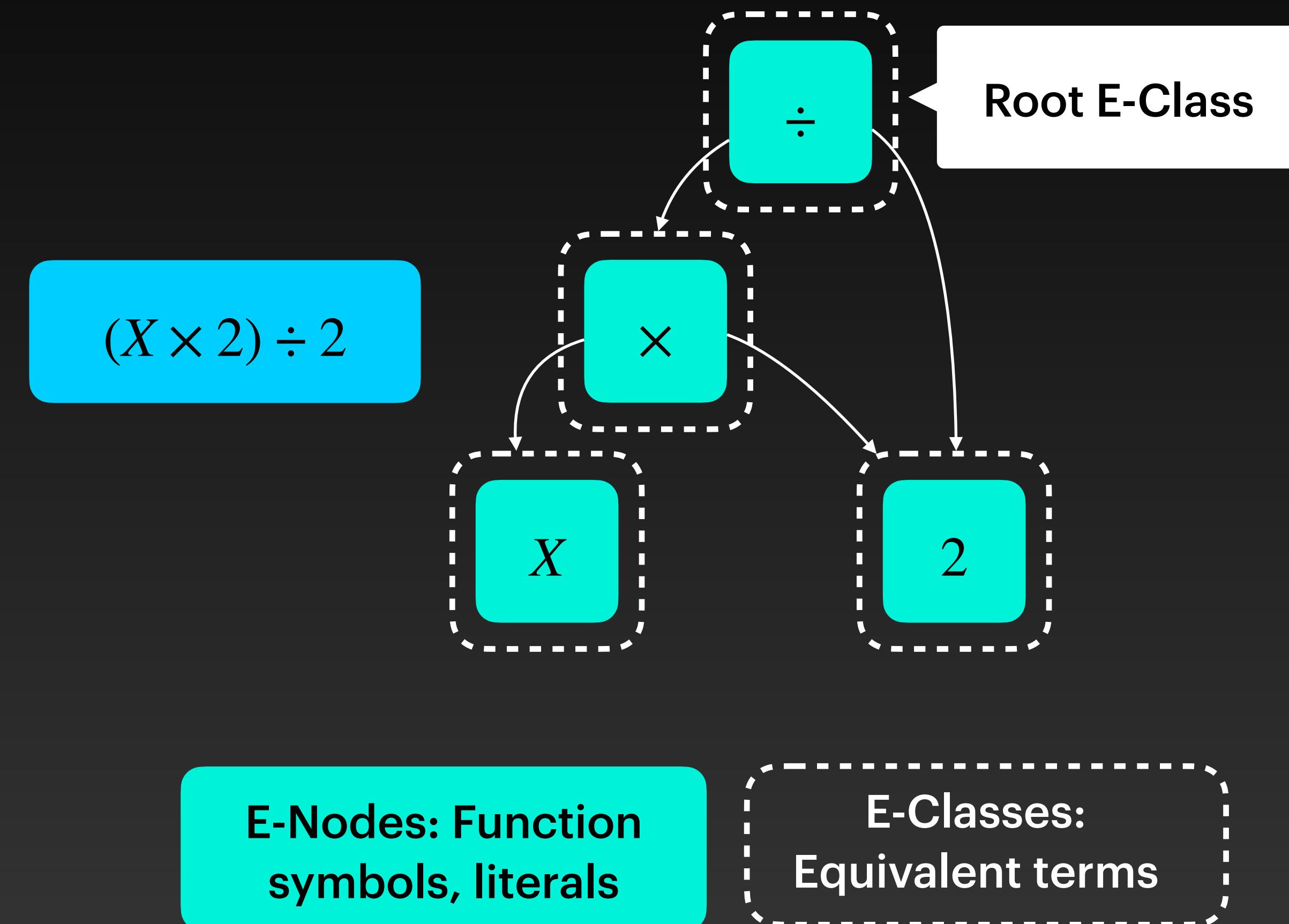


Equality Saturation



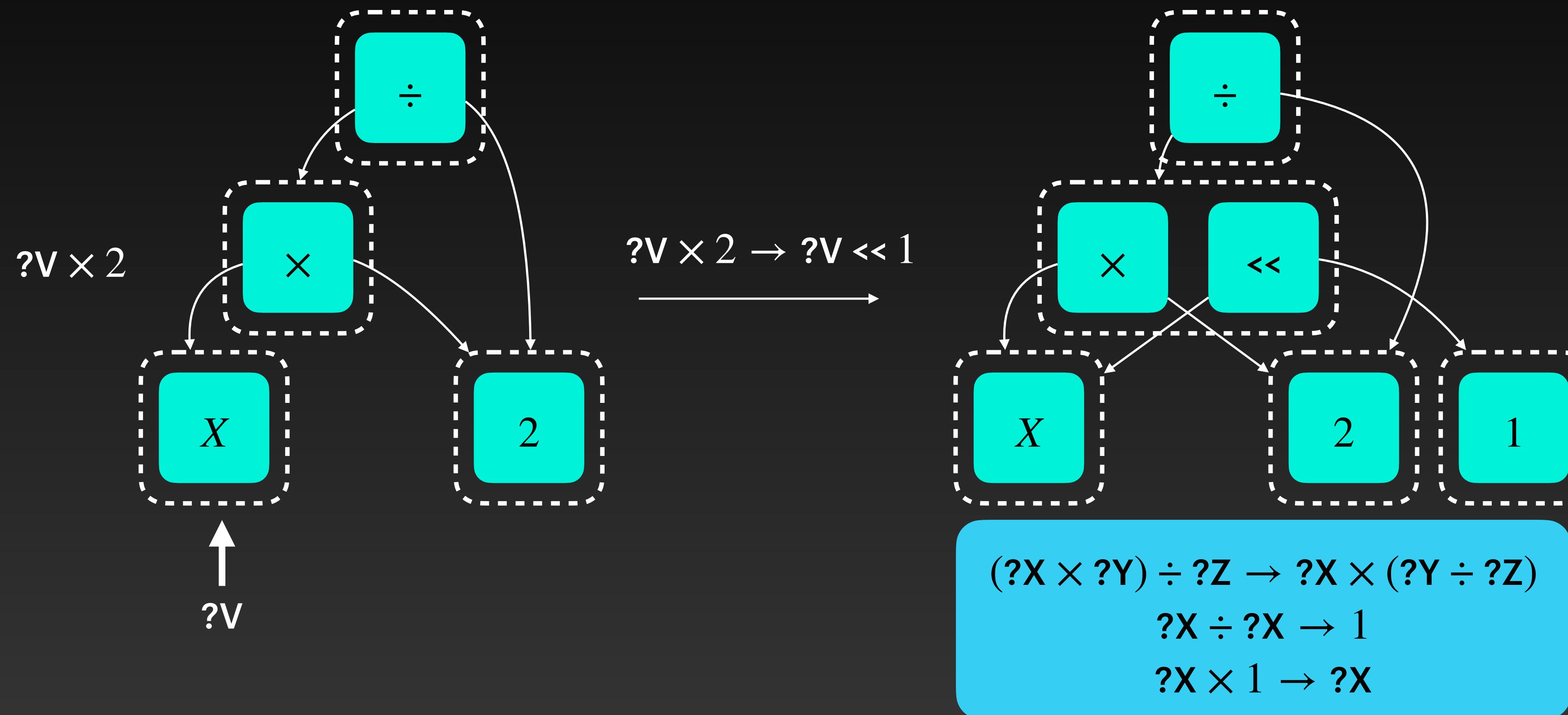
Equality Saturation and E-Graphs

Converting terms to E-Graphs



Equality Saturation and E-Graphs

Program Transformations with Syntactic Rewrites

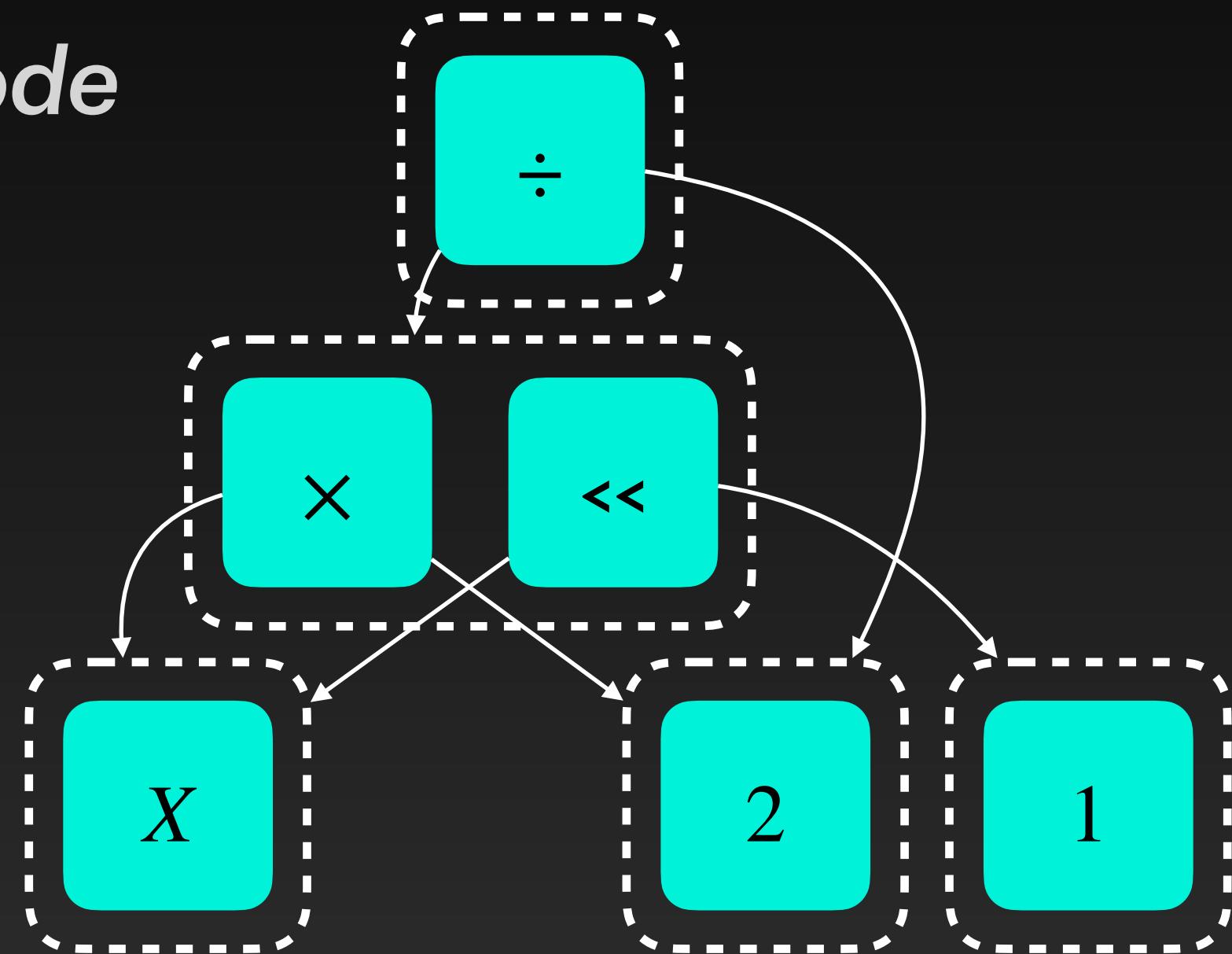


Non-destructive rewriting

Equality Saturation and E-Graphs

Term Extraction

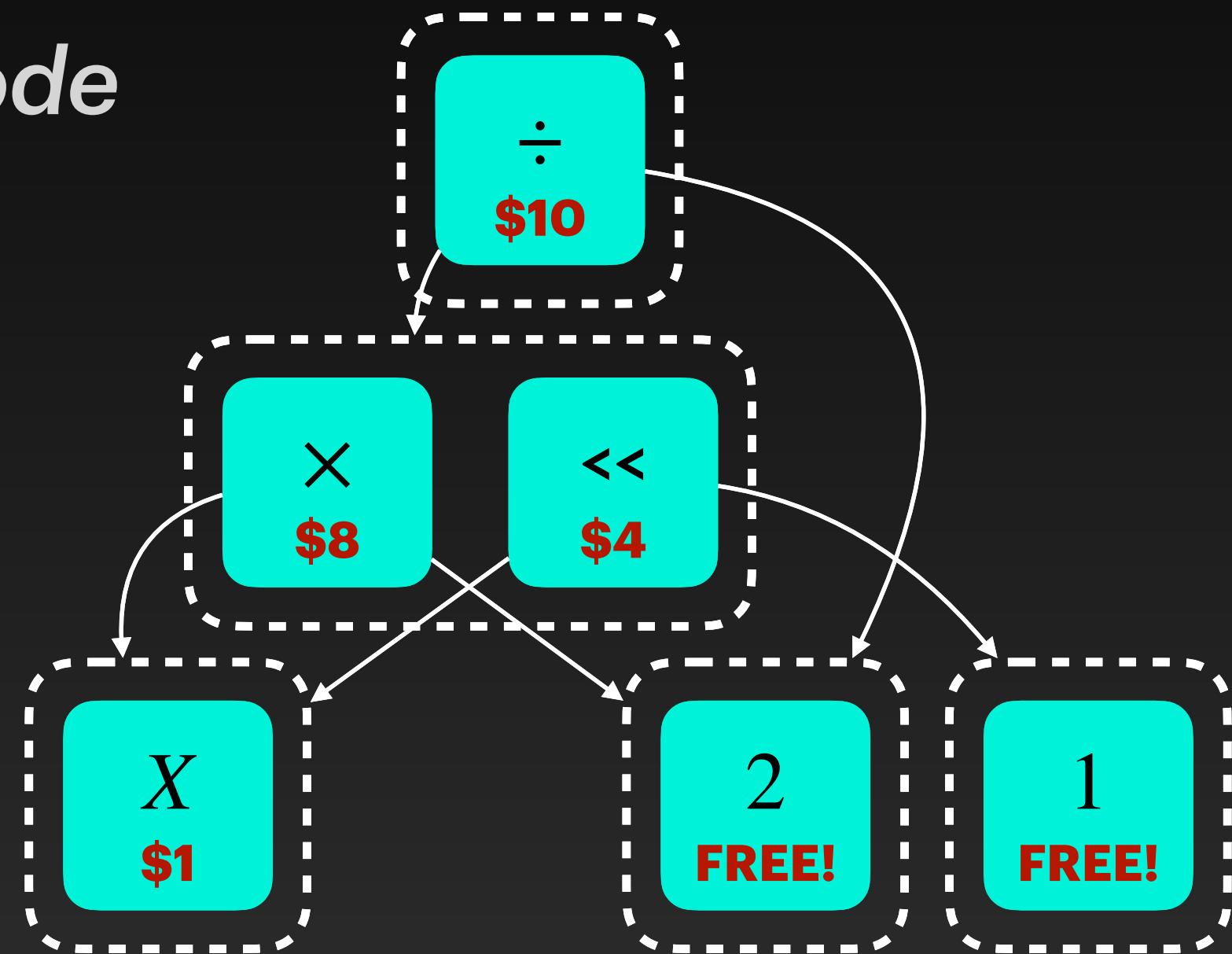
1. Assign a cost for each E-Node



Equality Saturation and E-Graphs

Term Extraction

1. Assign a cost for each E-Node



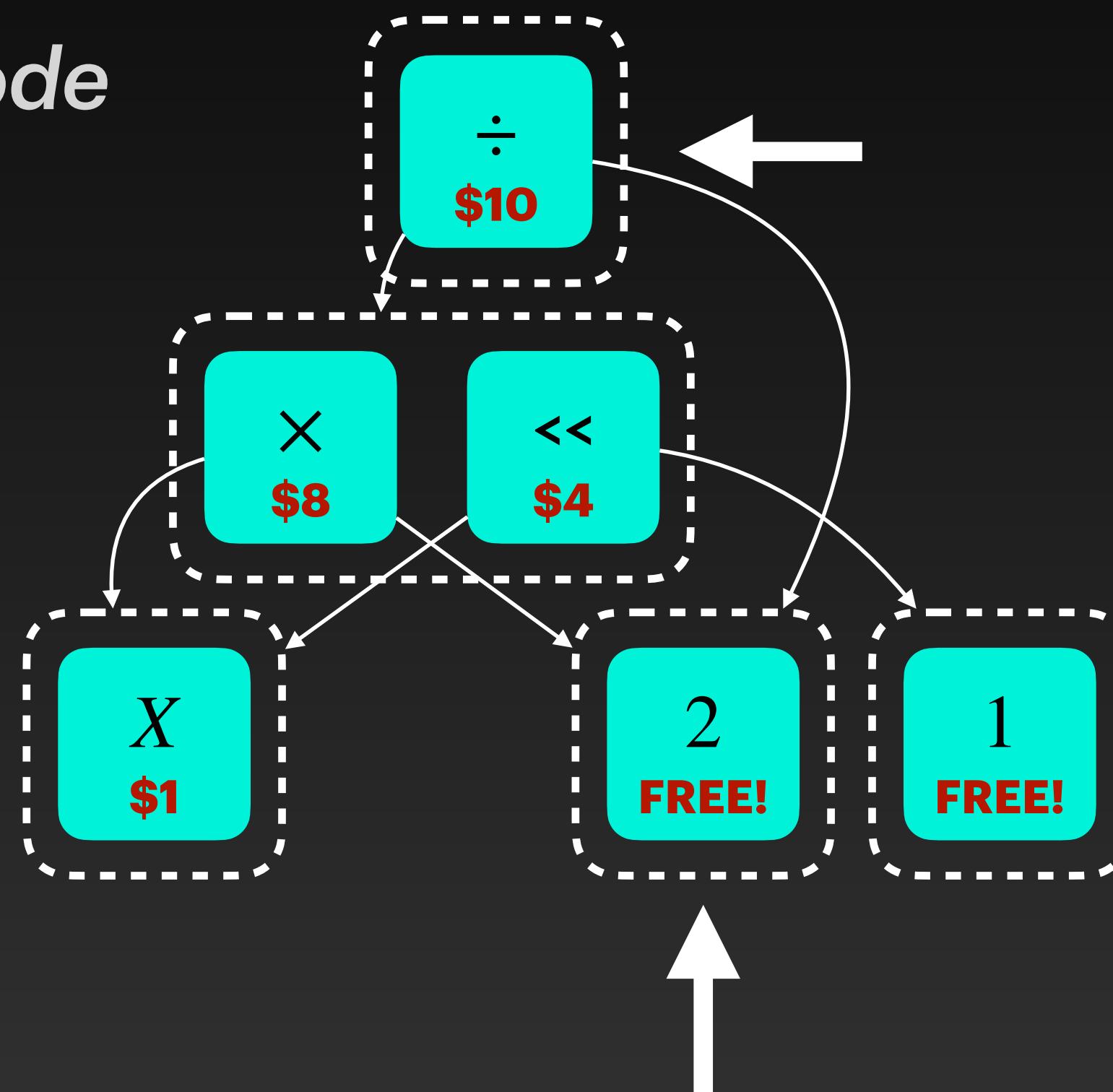
Equality Saturation and E-Graphs

Term Extraction

1. Assign a *cost* for each *E*-Node

2. Pick the min-cost term

Attempt: Greedy



$$(X \ll 1) \div 2$$

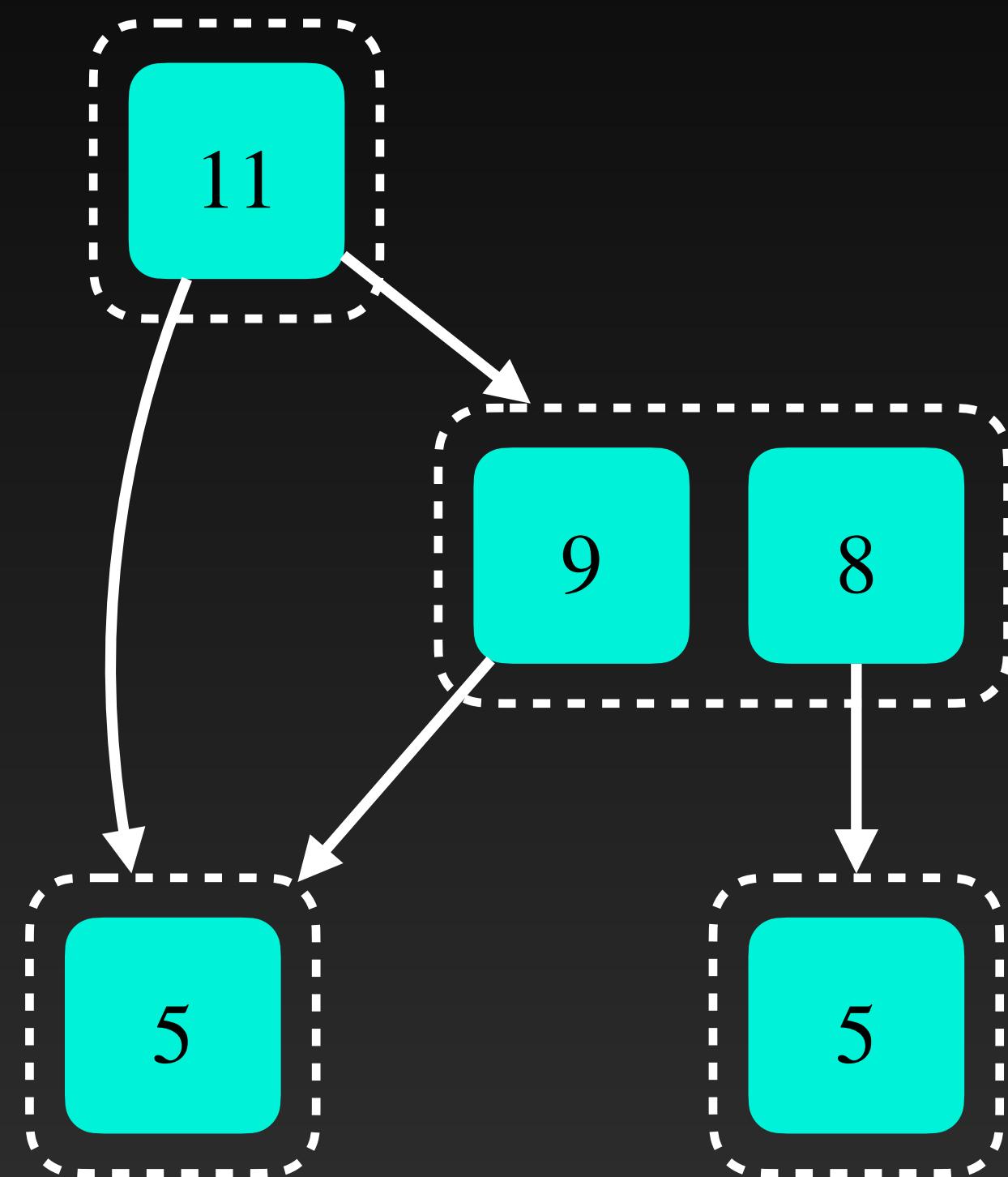
$$\text{Cost} = 10 + 4 + 1 = 15$$

Is That It?

Term Extraction

When Greedy Fails

Optimal:
 $11 + 9 + 5 = 25$



Greedy:
 $11 + 8 + 5 + 5 = 29$

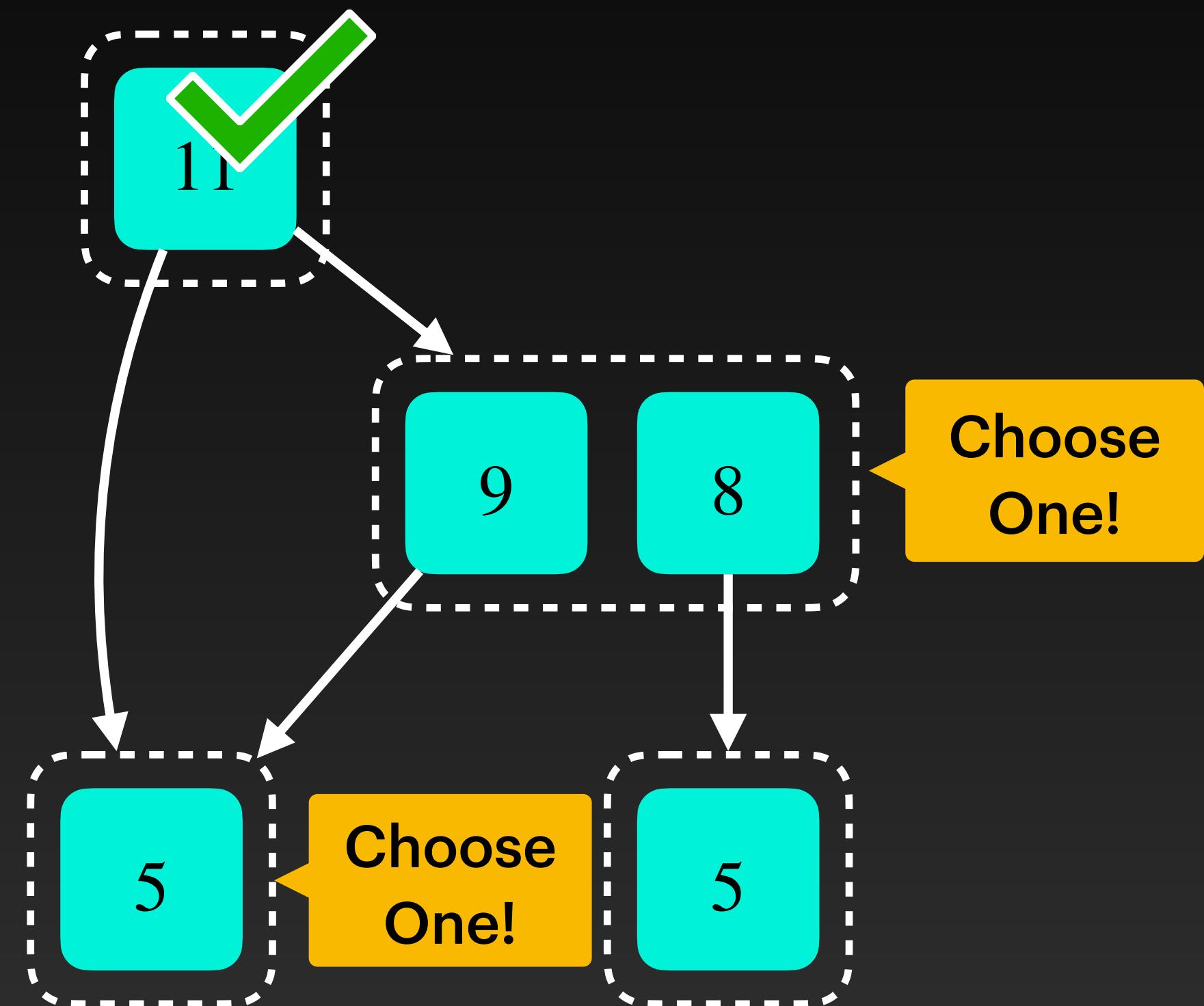
Previous work: ILP-based extraction

Root Constraint:

Extract at least one E-Node from the Root E-Class

Children Constraints:

If an E-Node n is extracted, then for all E-Class C , if C is a child of n , then extract at least one E-Node from C



Objective:

Minimize the sum of costs of extracted E-Node

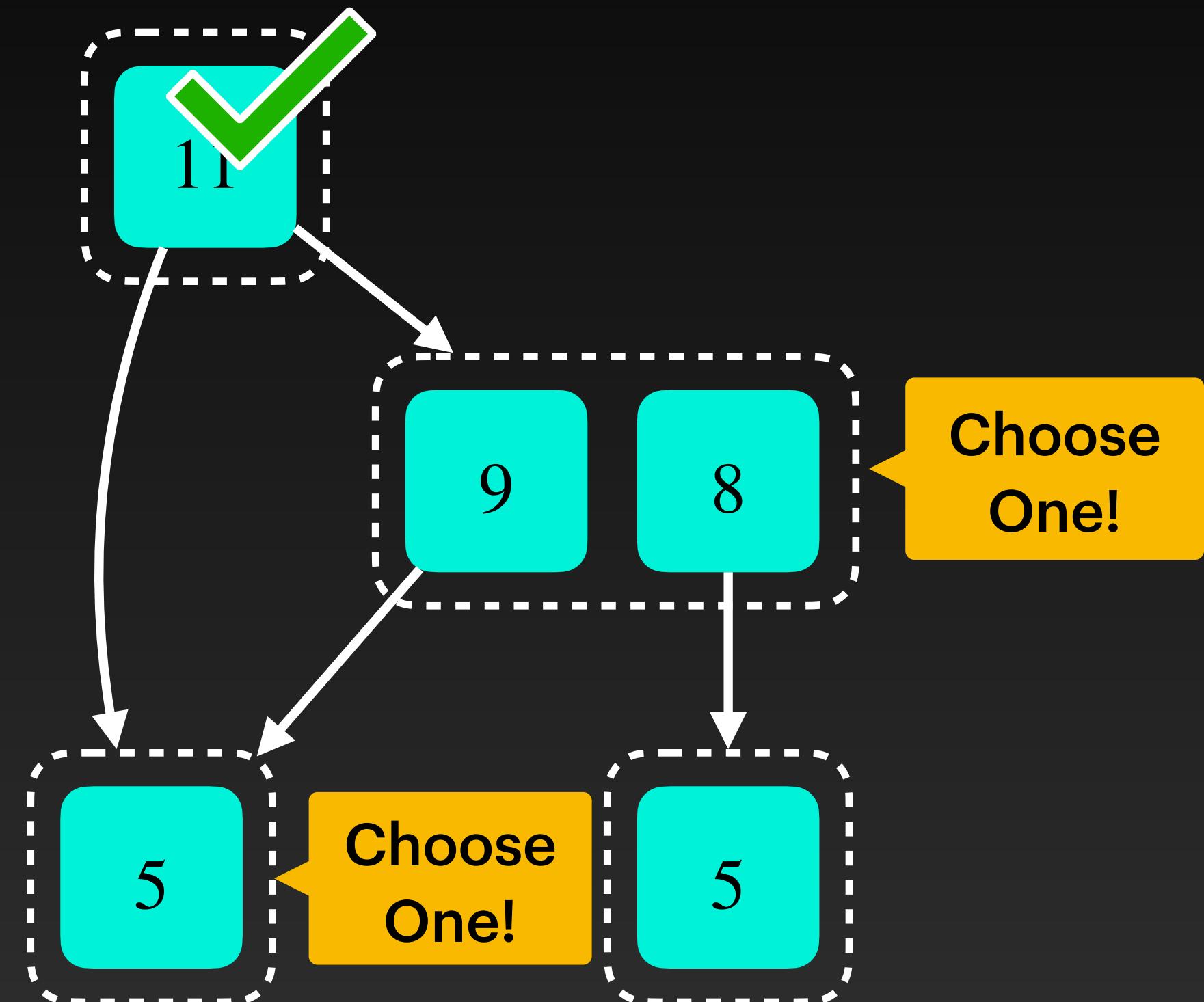
Previous work: ILP-based extraction

Variables: v_x for each e-node x

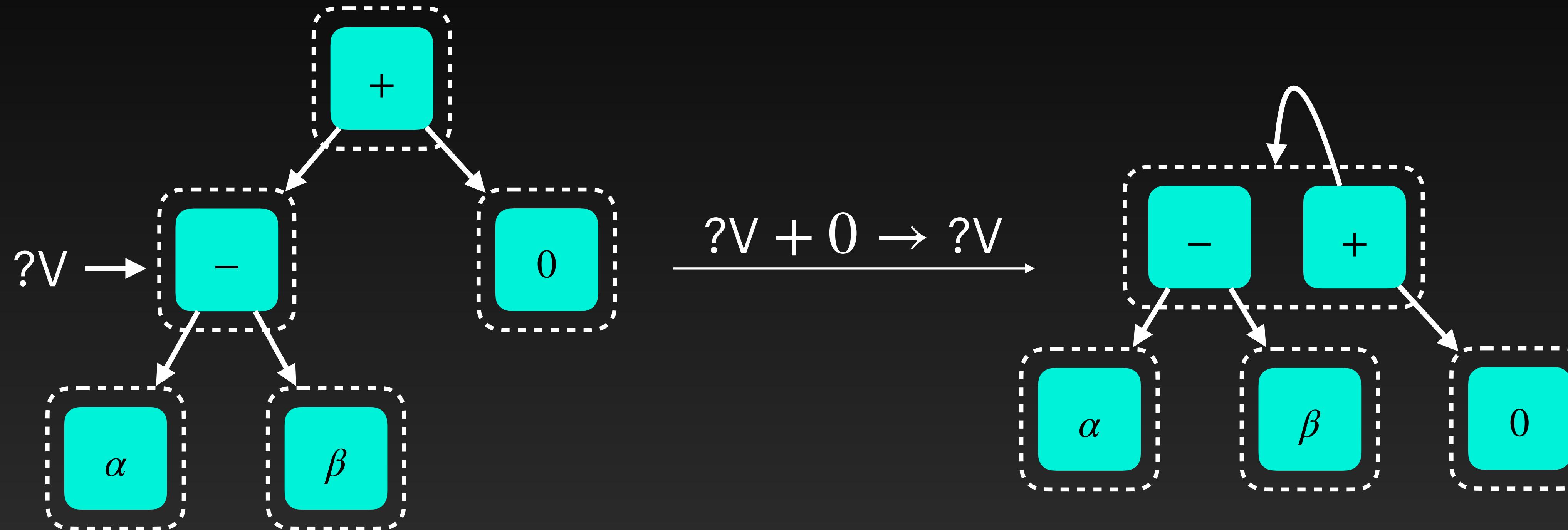
Objective:
 $\min \sum_x \text{cost}(x) \cdot v_x$

Root Constraint: $\sum_{x \in \text{Root}} v_x \geq 1$

Children Constraints: $-v_x + \sum_{y \in C_i} v_y \geq 1$
for each child C_i of x



Previous work: ILP-based extraction Cycles



How to avoid infinite expansions?

Previous work: ILP-based extraction

Topological Order Constraints

Variables: v_x for each e-node x

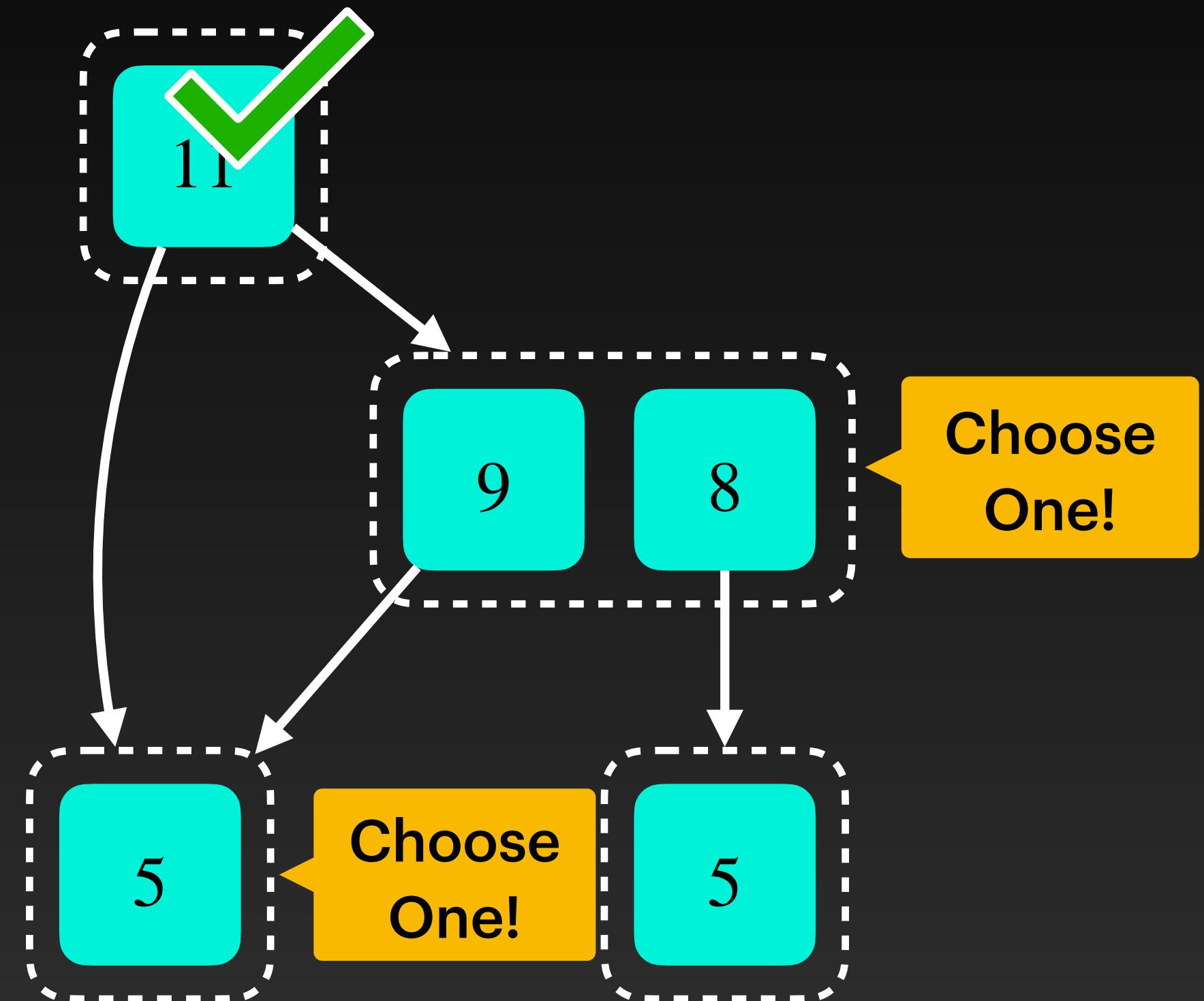
Objective:

$$\min \sum_x \text{cost}(x) \cdot v_x$$

Root Constraint: $\sum_{x \in \text{Root}} v_x \geq 1$

Children Constraints: $-v_x + \sum_{y \in C_i} v_y \geq 1$

for each child C_i of x



Previous work: ILP-based extraction

Topological Order Constraints

Variables: v_x, o_x for each e-node x

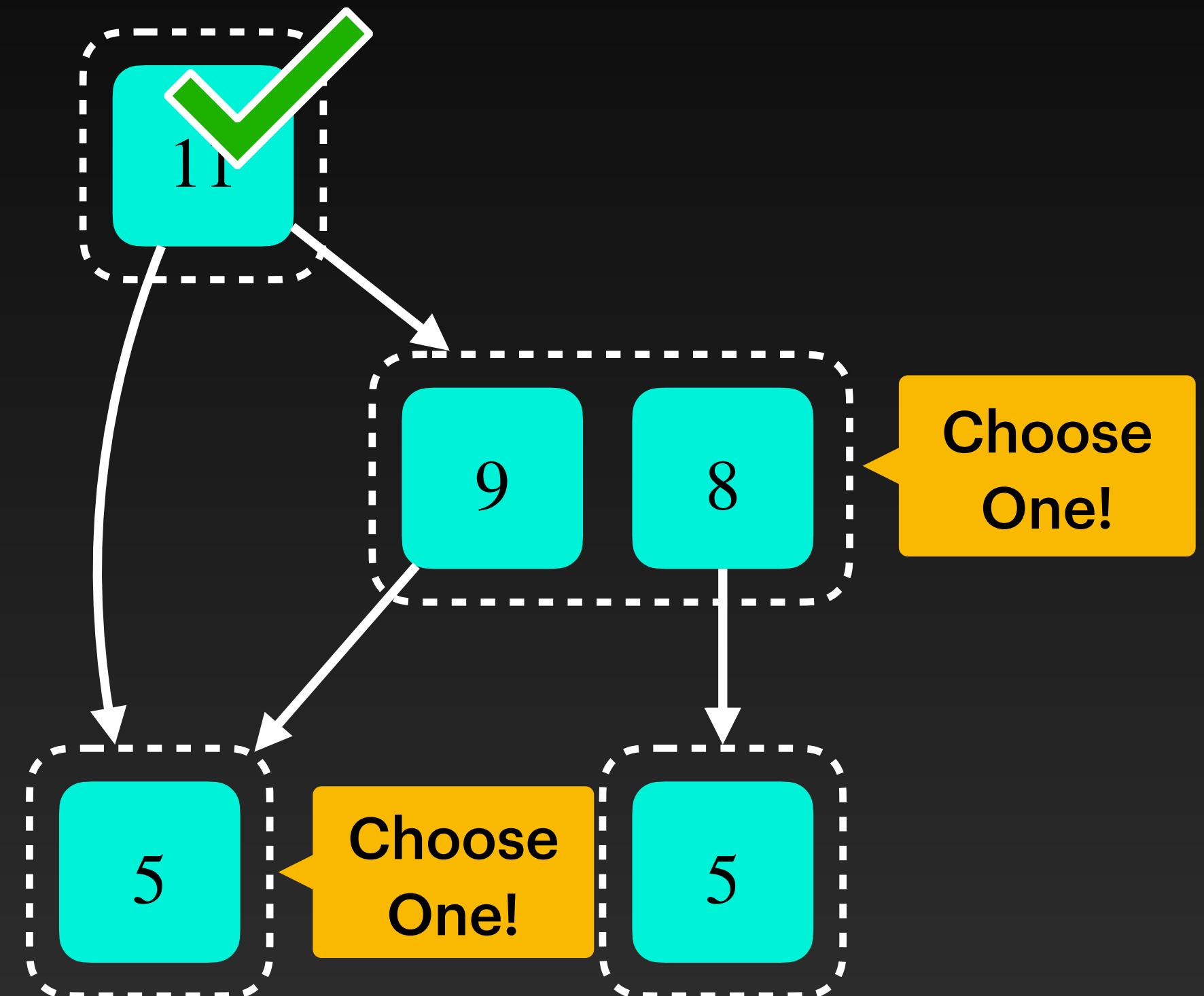
Topological order

Objective:
 $\min \sum_x \text{cost}(x) \cdot v_x$

Root Constraint: $\sum_{x \in \text{Root}} v_x \geq 1$

Children Constraints: $-v_x + \sum_{y \in C_i} v_y \geq 1$
for each child C_i of x

Topological order constraints: $o_y \geq o_x + 1$ (if $v_x = 1$). (y is in some children of x)



Previous work: ILP-based extraction

Topological Order Constraints

Variables: v_x, o_x for each e-node x

Topological order

Objective:
 $\min \sum_x \text{cost}(x) \cdot v_x$

Root Constraint: $\sum_{x \in \text{Root}} v_x \geq 1$

Children Constraints: $-v_x + \sum_{y \in C_i} v_y \geq 1$
for each child C_i of x

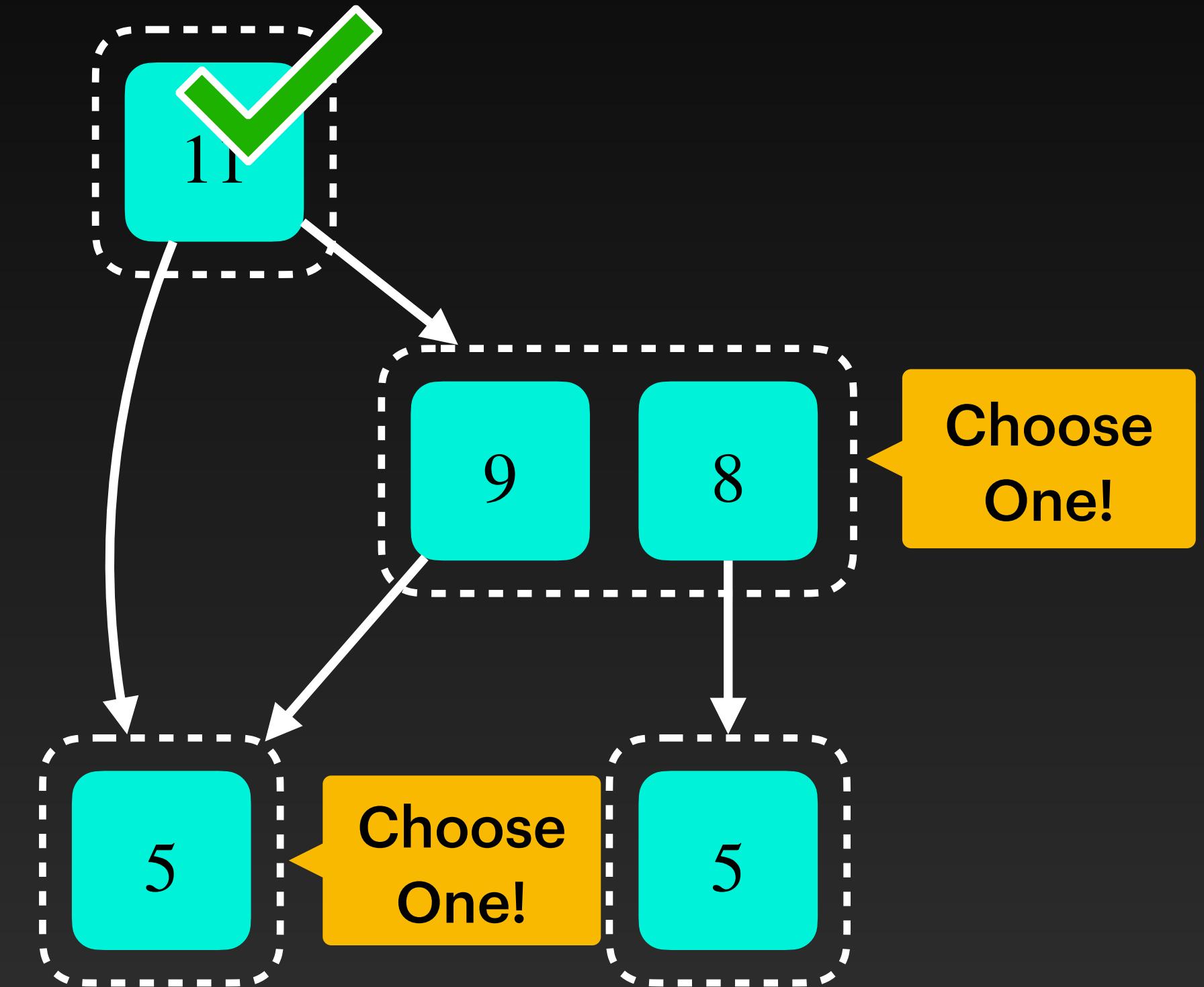
Topological order constraints: $o_y + (1 - v_x) \cdot L \geq o_x + 1$ (y is in some children of x)

L is a large enough constant

Variables: $O(n)$

Constraints: $O(n)$

Search Space: $O(2^n + n^n)$



Our solution 1: ILP + Acyclicity constraints

Variables: v_x for each e-node x

Objective:

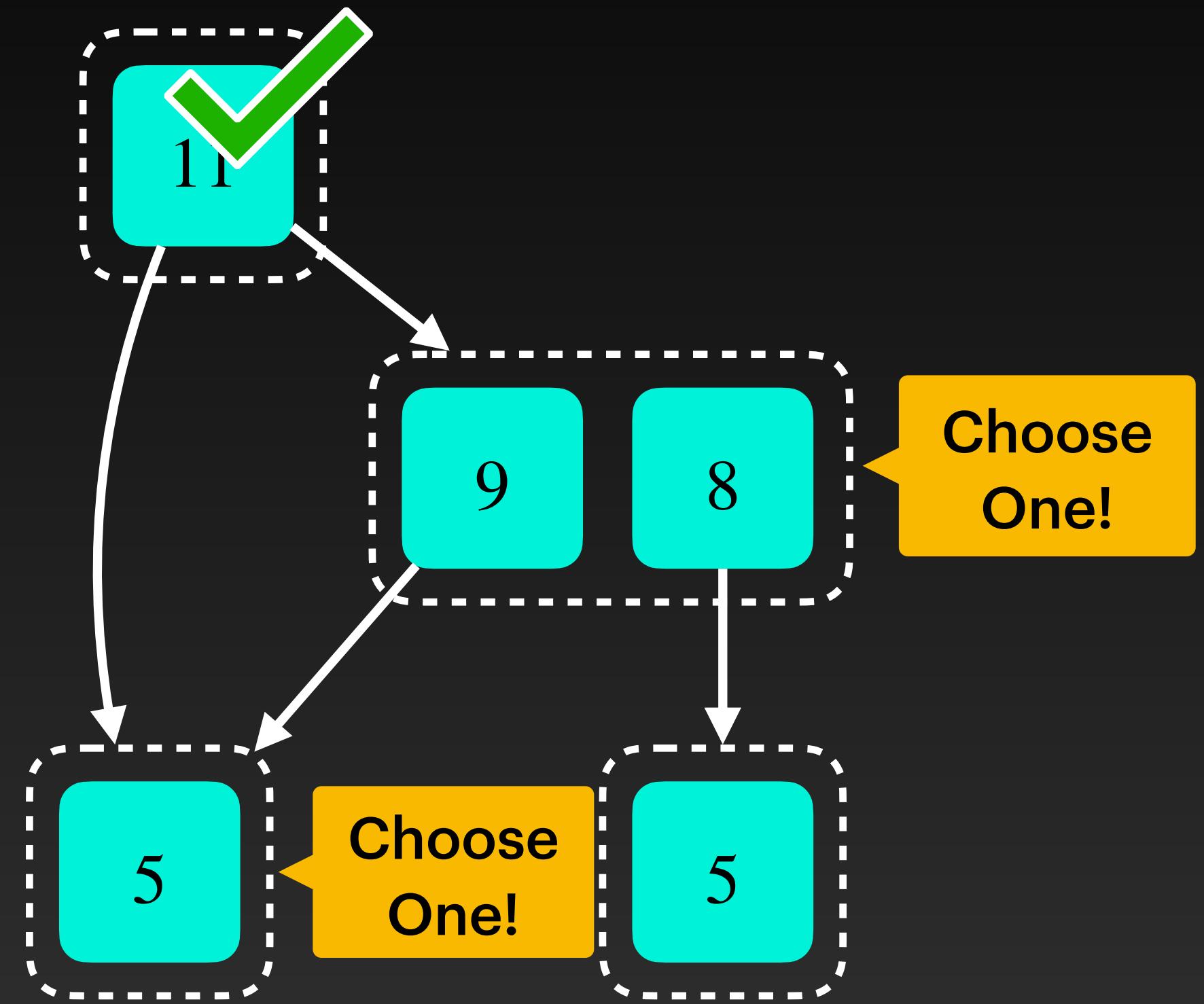
$$\min \sum_x \text{cost}(x) \cdot v_x$$

Root Constraint: $\sum_{x \in \text{Root}} v_x \geq 1$

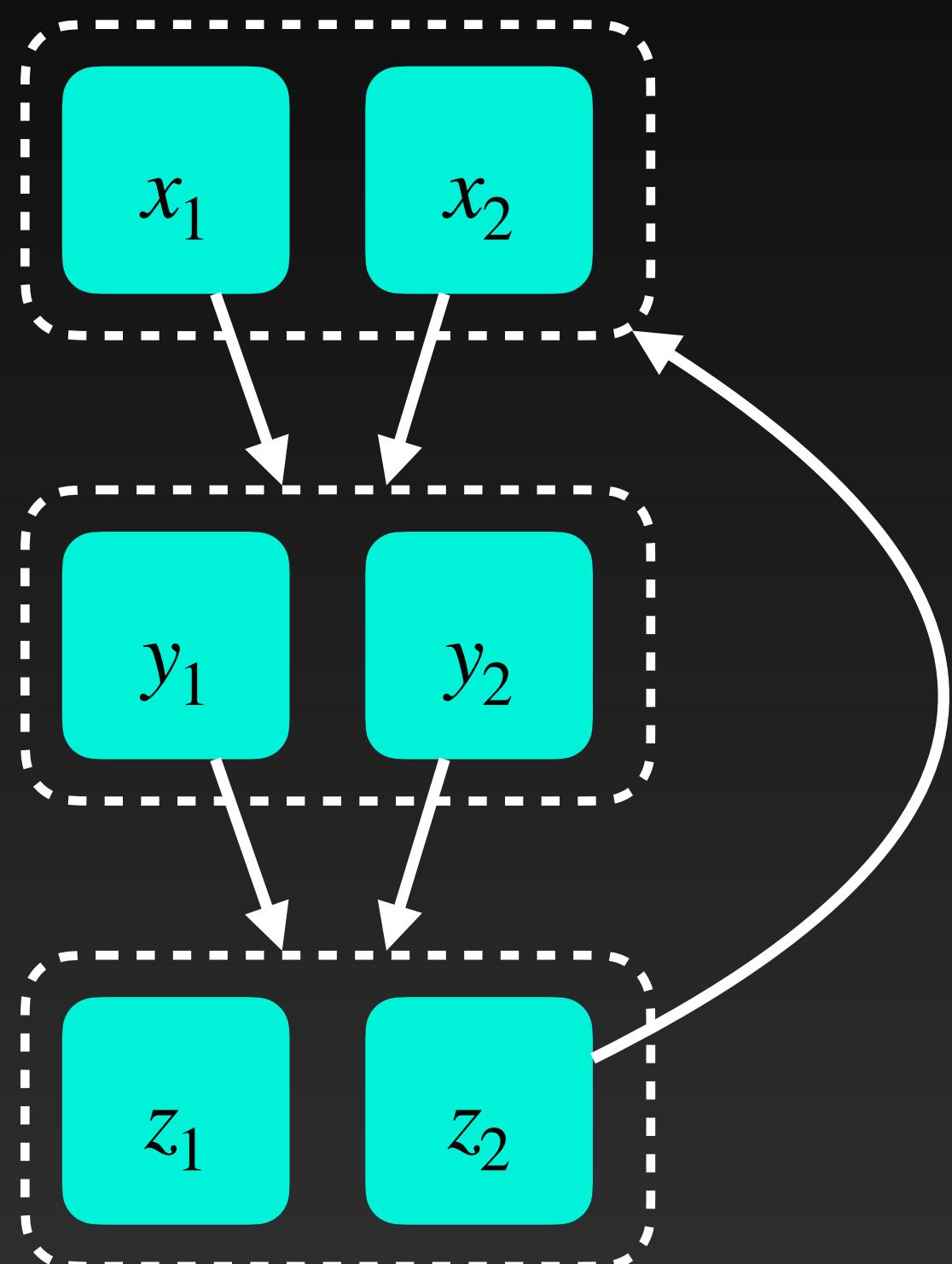
Children Constraints: $-v_x + \sum_{y \in C_i} v_y \geq 1$
for each child C_i of x

Acyclicity Constraints: Do not extract any cycle

Works well when number of cycles is reasonable



Acyclicity constraints



$$\begin{aligned} & Tseitin \left(\bigvee \begin{array}{l} (\neg x_1 \wedge \neg x_2) \\ (\neg y_1 \wedge \neg y_2) \\ \neg z_2 \end{array} \right) \\ & \Leftrightarrow \\ & \bigwedge \left(\begin{array}{l} O_1 \leftrightarrow (\neg x_1 \wedge \neg x_2) \\ O_2 \leftrightarrow (\neg y_1 \wedge \neg y_2) \\ O_1 \vee O_2 \vee \neg z_2 \end{array} \right) \end{aligned}$$

Acyclicity constraints in
ILP formulation

Solution 1: ILP + Acyclicity constraints

Variables: v_x for each e-node x

Objective:

$$\min \sum_x \text{cost}(x) \cdot v_x$$

Root Constraint: $\sum_{x \in \text{Root}} v_x \geq 1$

Children Constraints: $-v_x + \sum_{y \in C_i} v_y \geq 1$

for each child C_i of x

Acyclicity Constraints:

Acyclicity constraints in
ILP formulation

Variables: $O(n)$

Constraints: $O(n \cdot \#cycles)$

Search Space: $O(2^n)$

Solution 2: Weighted Partial MaxSAT

For each E-Node x , create a boolean variable v_x

v_x is $\top \Leftrightarrow x$ is in the extracted term

Must always be satisfied

SAT / UNSAT

Hard Clauses

Root Constraint:

$$\bigvee_{x \in \text{Root}} v_x$$

Children Constraints:

$$v_x \rightarrow \bigwedge_{C \in \text{children}(x)} \bigvee_{x' \in C} v_{x'}$$

Acyclicity Constraints:

$$\text{Tesitin} \left(\bigvee_{C_i} \bigwedge_{x \in C_i \wedge \text{in_cycle}(x)} v_x \right)$$

Soft Clauses

$\neg v_x$ with weight $\text{cost}(x)$

Objective:
Maximizing weight of unextracted E-Nodes

Variables: $O(n)$

Constraints: $O(n \cdot \# \text{cycles})$

Search Space: $O(2^n)$

Term extraction

Complexity

*Solution 1 (**ILP-ACyc**): ILP formulation with acyclic constraints*

*Solution 2 (**WPMAXSAT**): Weighted partial MaxSAT formulation with acyclic constraints*

*Previous work (**ILP-Topo**): ILP with topological order constraints*

Encoding	# Variables	# Constraints	Search Space Complexity
ILP-ACyc WPMAXSAT	$O(n)$	$O(nk)$	$O(2^n)$
ILP-Topo	$O(n)$	$O(n)$	$O(2^n + n^n)$

n : number of E-Nodes

k : number of E-Class cycles

Potentially Exponential

Term extraction

Evaluation benchmarks

Empirically

Implemented a prototype in the egg [1] framework

Workload: term extraction after equality saturation on tensor programs (DNNs) including
MobileNetV2, ResMLP, ResNet-18, ResNet-50, EfficientNet

Rewrite rules from Glenside [2]

- *Image-to-column (im2col) only*
- *Image-to-column (im2col) + simplifications (operator fusion, reordering, etc.)*

A	B	C
D	E	F
G	H	I

A	B	D	E
B	C	E	F
D	E	G	H
E	F	H	I

Im2col of a 3x3 input for a 2x2 kernel

[1] Willsey, M., et al. "egg: Fast and extensible equality saturation," in *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[2] Smith, Gus Henry, Andrew, Liu, Steven, Lyubomirsky, Scott, Davidson, Joseph, McMahan, Michael, Taylor, Luis, Ceze, Zachary, Tatlock. "Pure tensor program rewriting via access patterns (representation pearl)." *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. ACM, 2021.

Term extraction

Benchmark statistics

Unit: 1,000	MobileNetV2		ResMLP		ResNet-18		ResNet-50		EfficientNet	
	Im2Col	Im2Col+ SIMPL	Im2Col	Im2Col+ SIMPL	Im2Col	Im2Col+ SIMPL	Im2Col	Im2Col+ SIMPL	Im2Col	Im2Col+ SIMPL
# E-Nodes	50	20	40	8	35	8	45	40	50	20
# E-Classes	25	6	20	2.5	25	3	22	20	20	7
# Cycles	17	17	15	4	14	4	21	10	16	20

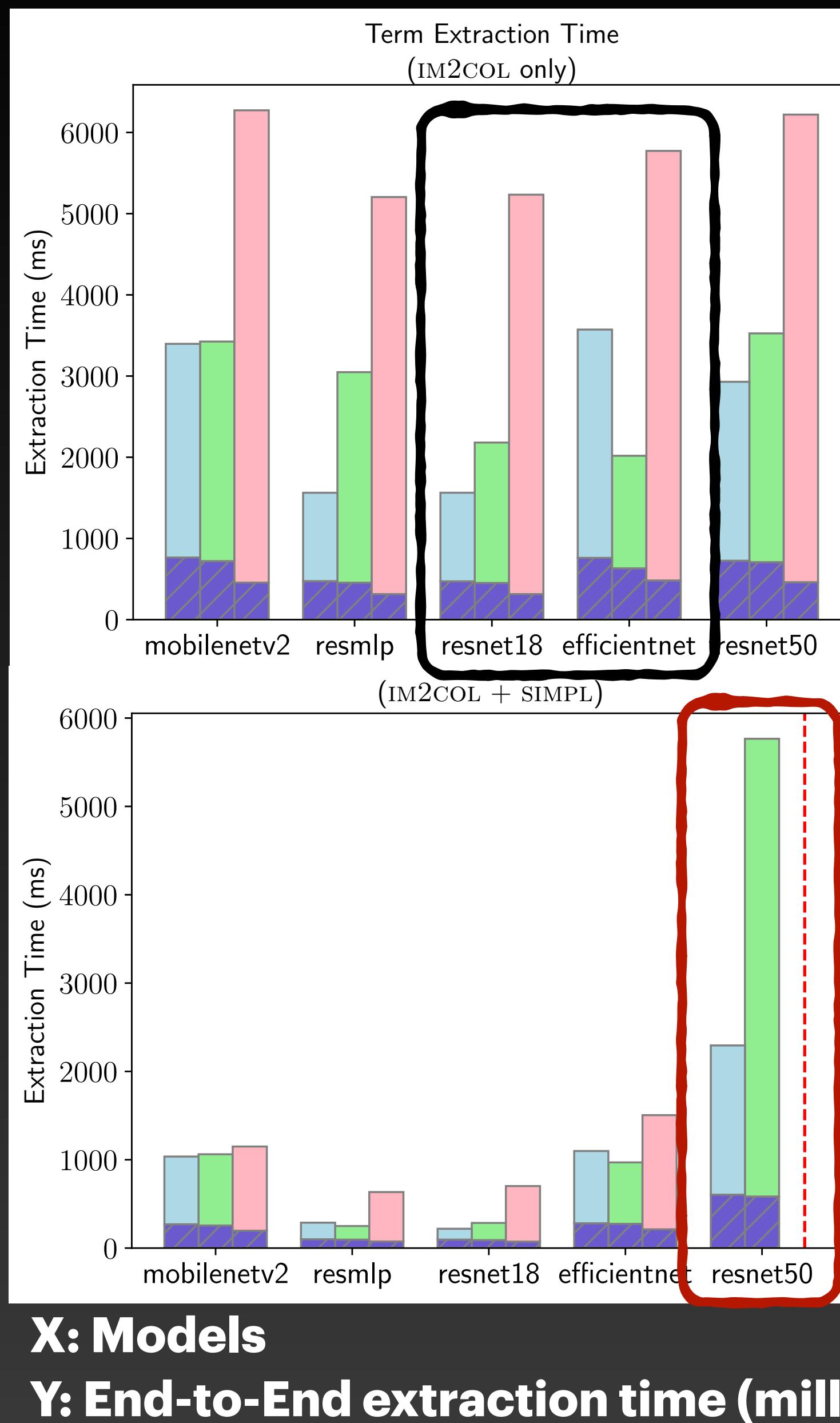
Statistics of saturated E-Graphs (Unit: 1k)

Term extraction

Evaluation results

Upper: Image-to-column rewrite rule only

Lower: Image-to-column + simplifications including
Operator fusion, reordering, etc.



ILP-Topo timeouts (300s)

Solving WP MAXSAT and ILP-ACyc
is **~3x faster** than solving ILP-Topo

For a larger input, solving ILP-Topo (previous work) timeouts after
300s while solving WP MAXSAT and ILP-ACyc takes a few seconds

Optimality is guaranteed by all encodings

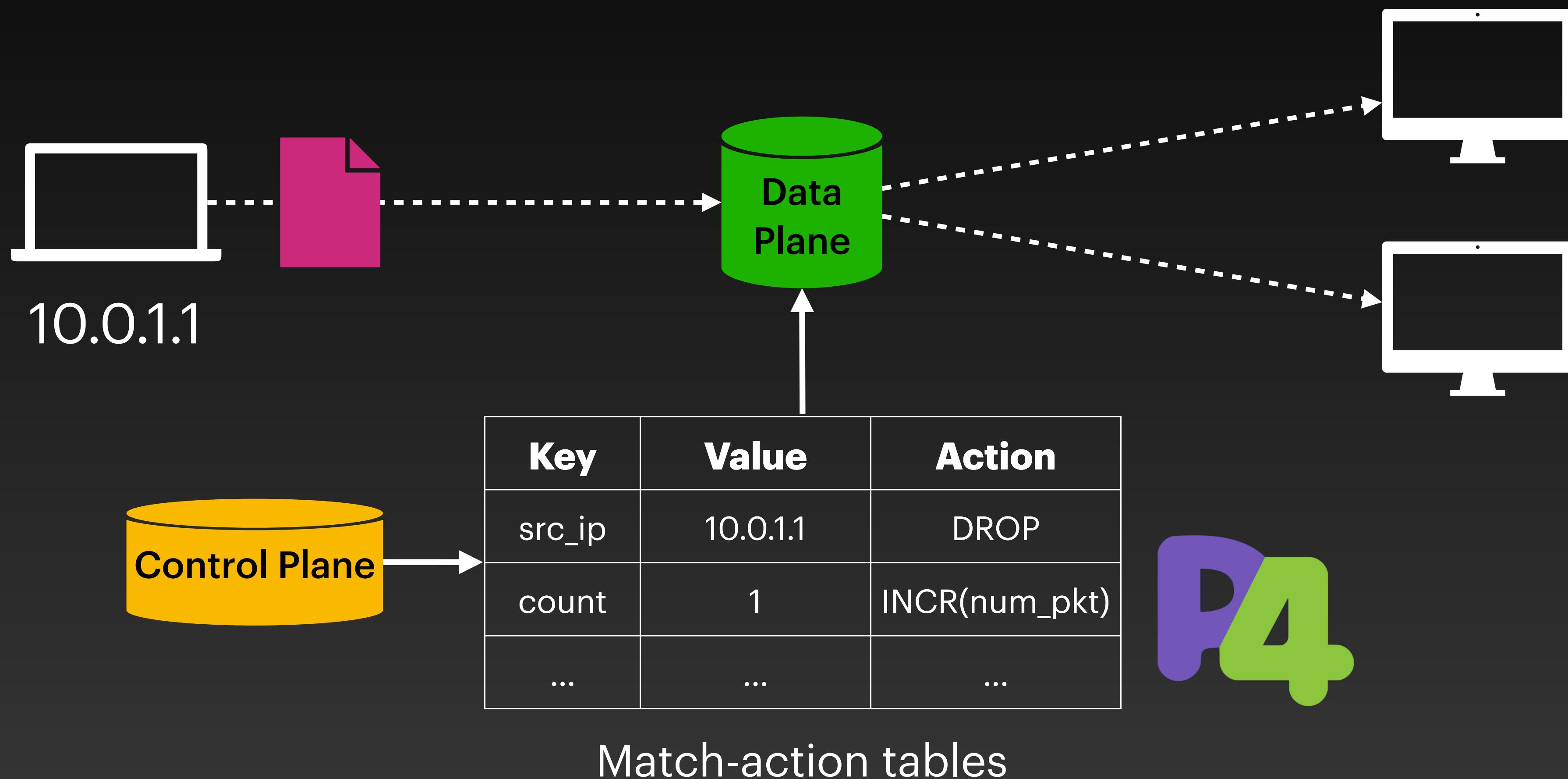


EGRAPHS'23 Workshop paper

<https://www.cs.princeton.edu/~dh7120/assets/papers/EGRAPHS2023.pdf>

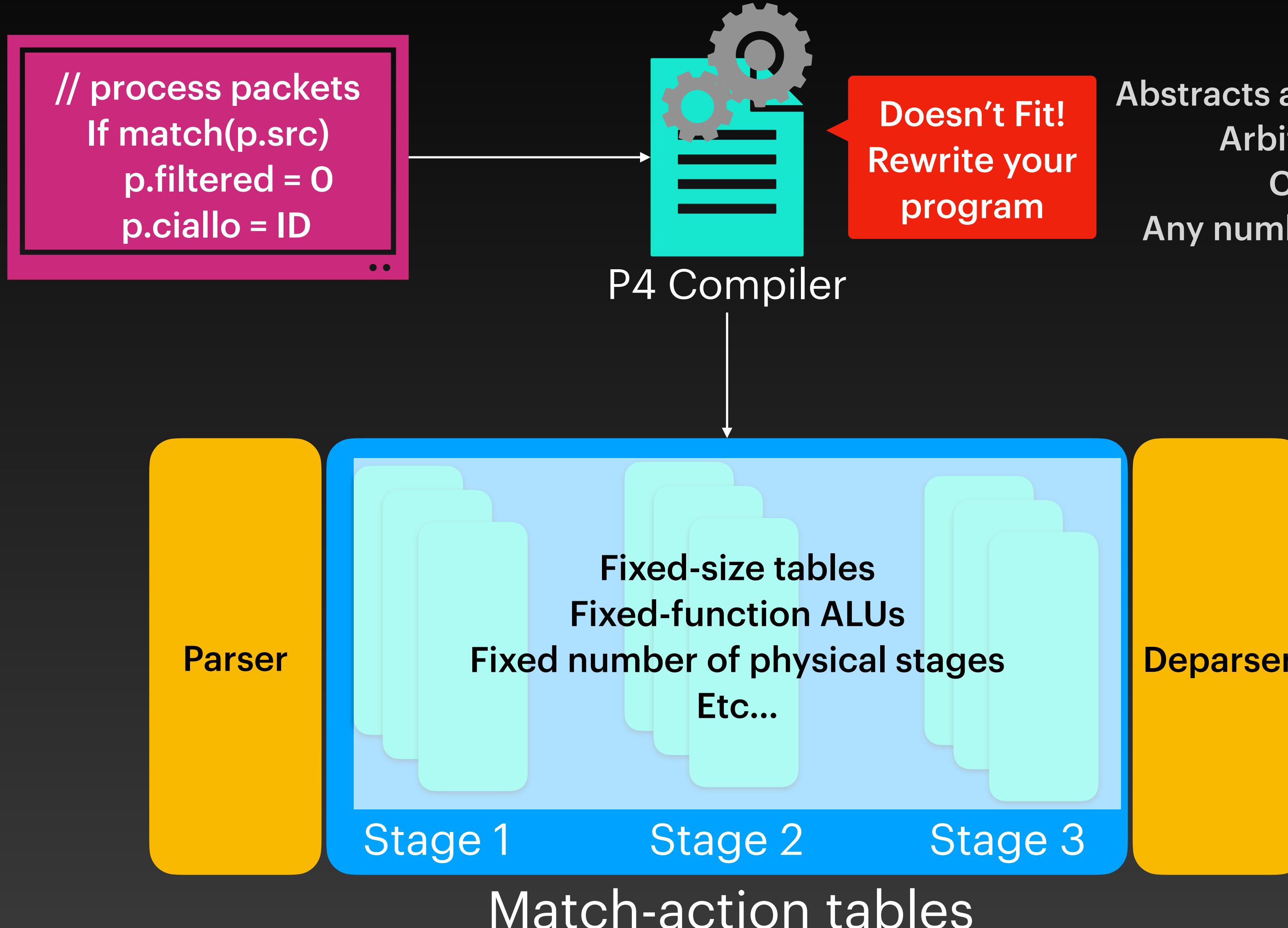
CatsTail: P4 Resource Synthesis using Equality Saturation

Programmable switches



Mapping to programmable switches is hard

P4



Abstraction away hardware details
Arbitrary computes
Control flows
Any number of logical stages

Mapping to programmable switches is hard

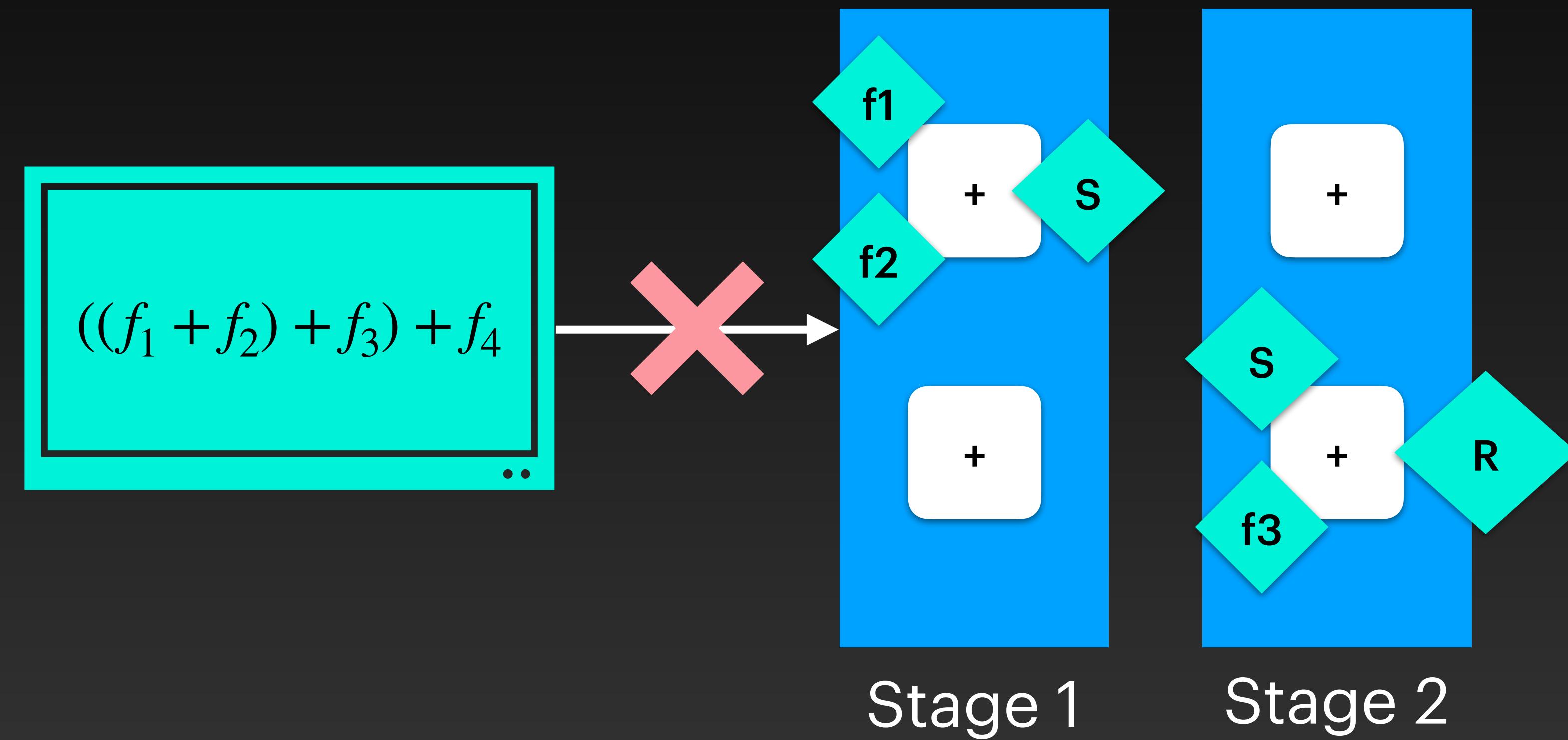
Challenge 1: Limited # of Stages

Challenge 2: Table Dependencies

Challenge 3: Targeting different backends

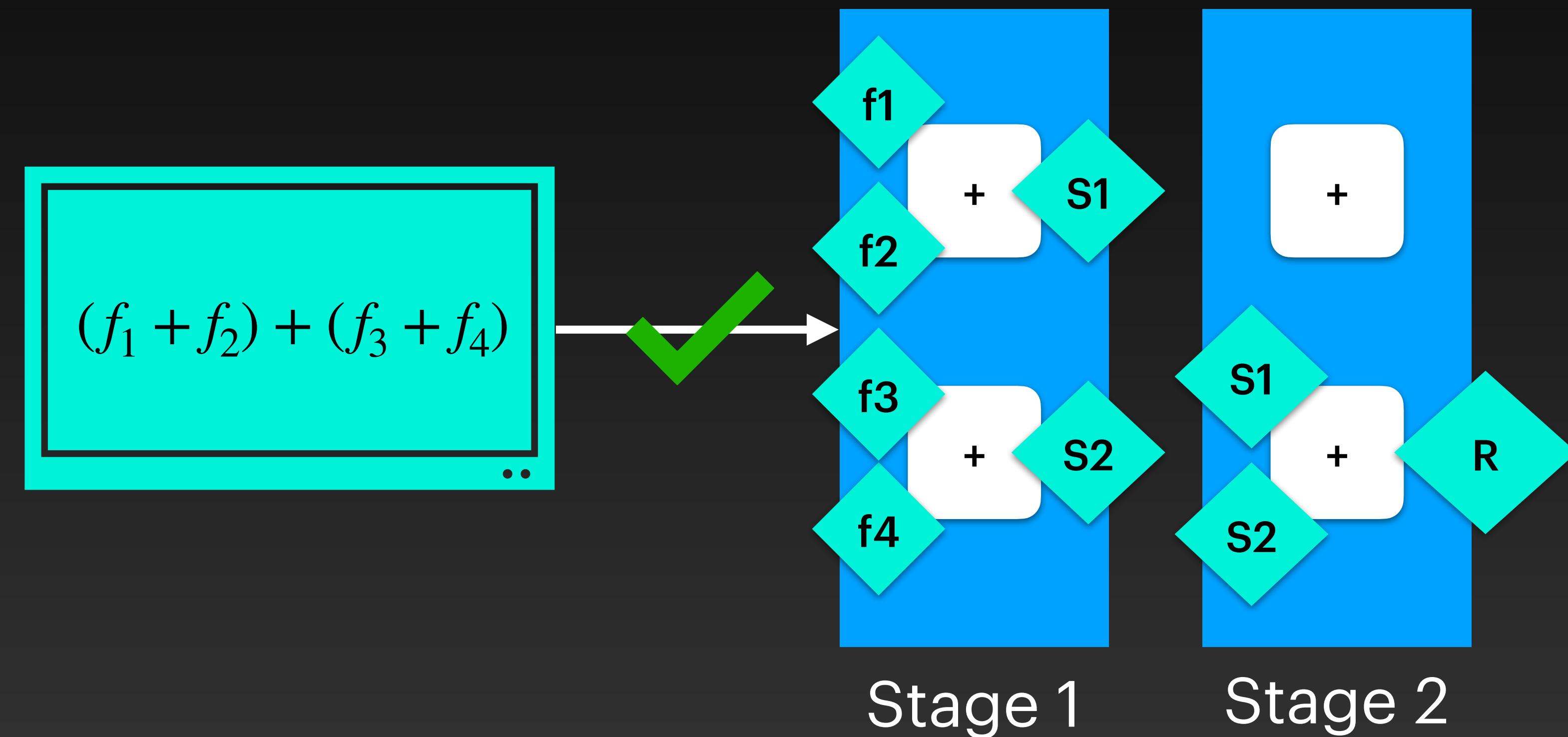
Mapping to programmable switches is hard

Challenge 1: Limited # of Stages



Mapping to programmable switches is hard

Challenge 1: Limited # of Stages

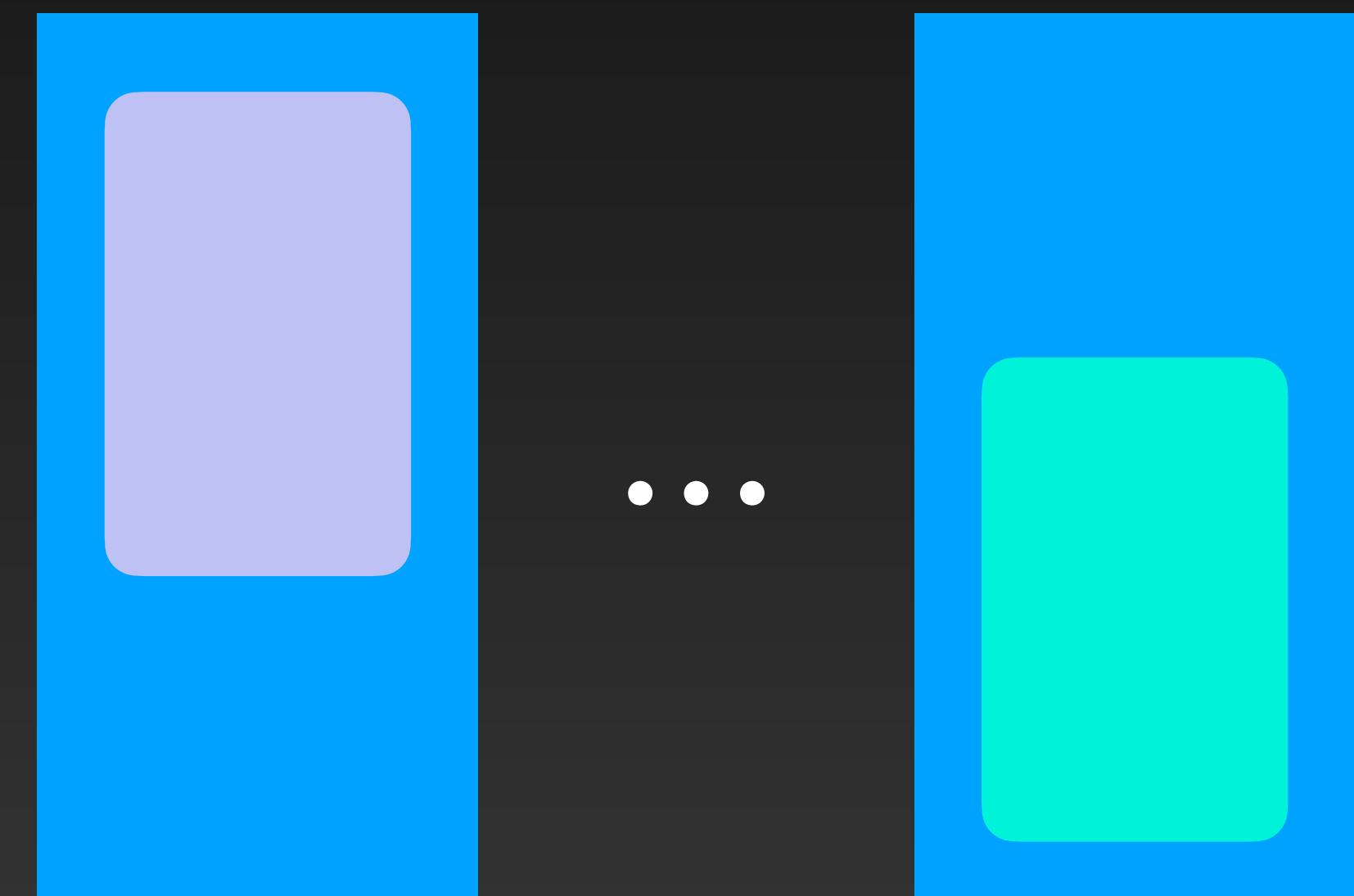
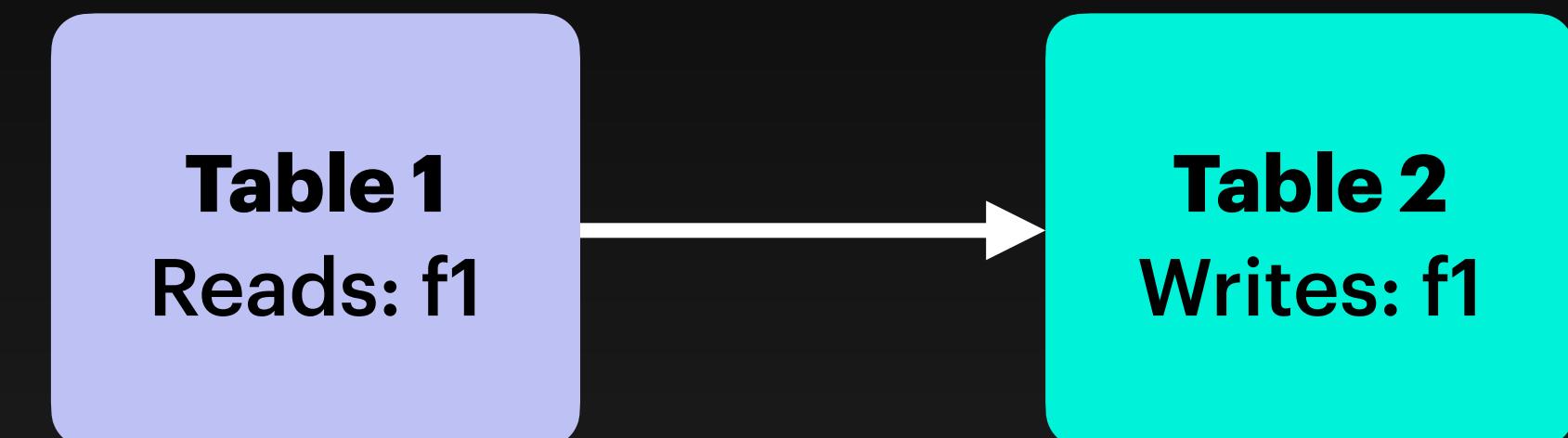


Mapping to programmable switches is hard

Challenge 2: Table Dependencies

R/W Dependencies

(this example) Write-after-Read
Read-after-Write
Write-after-Write



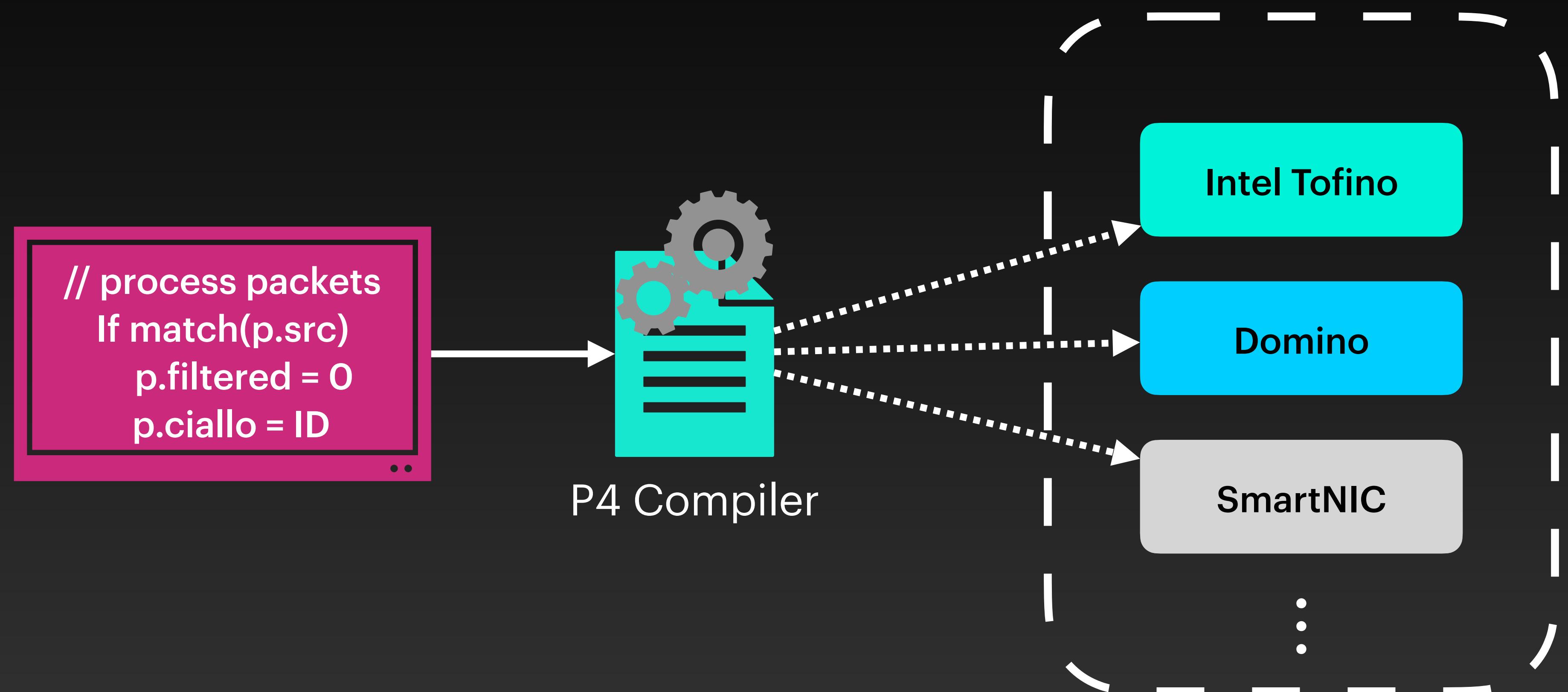
Stage X

Stage Y

$X < Y$

Mapping to programmable switches is hard

Challenge 3: Targeting different backends



Previous work: CaT

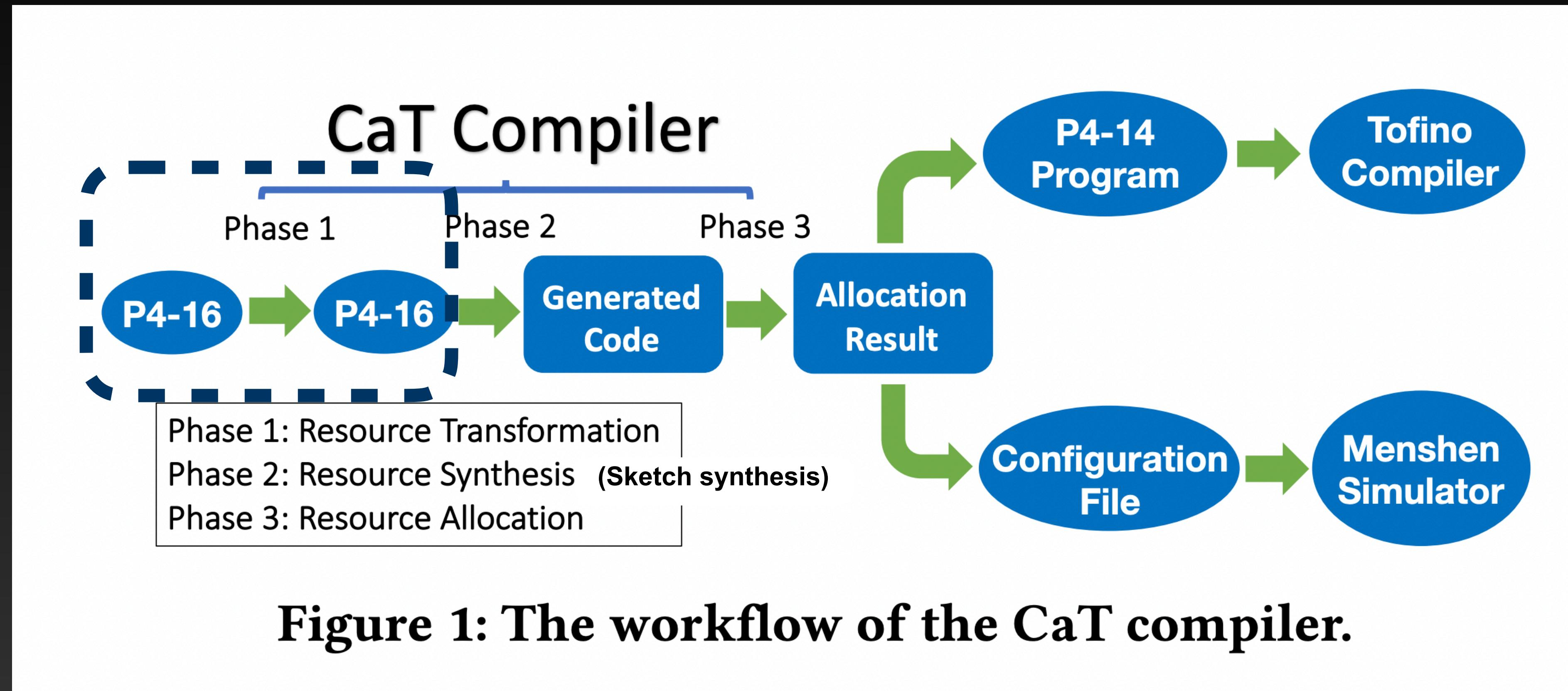
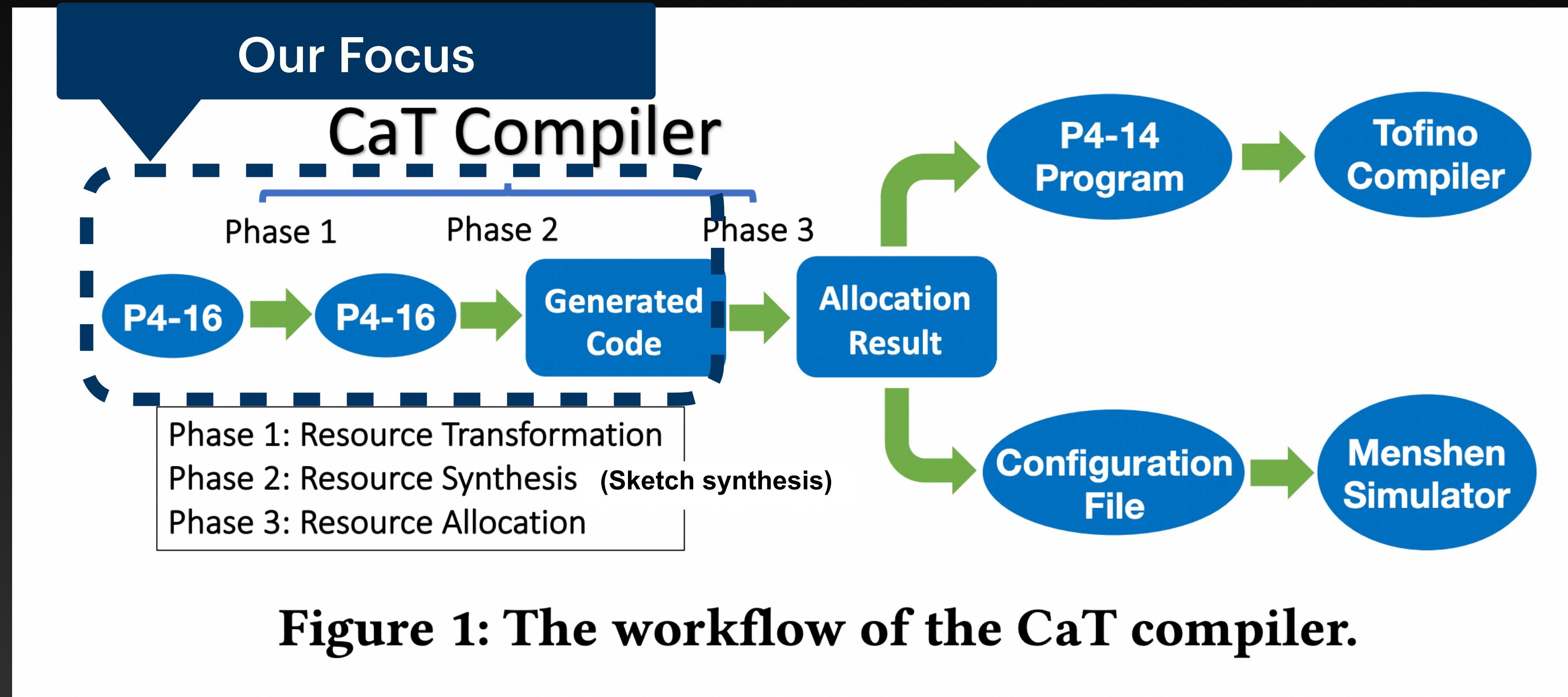
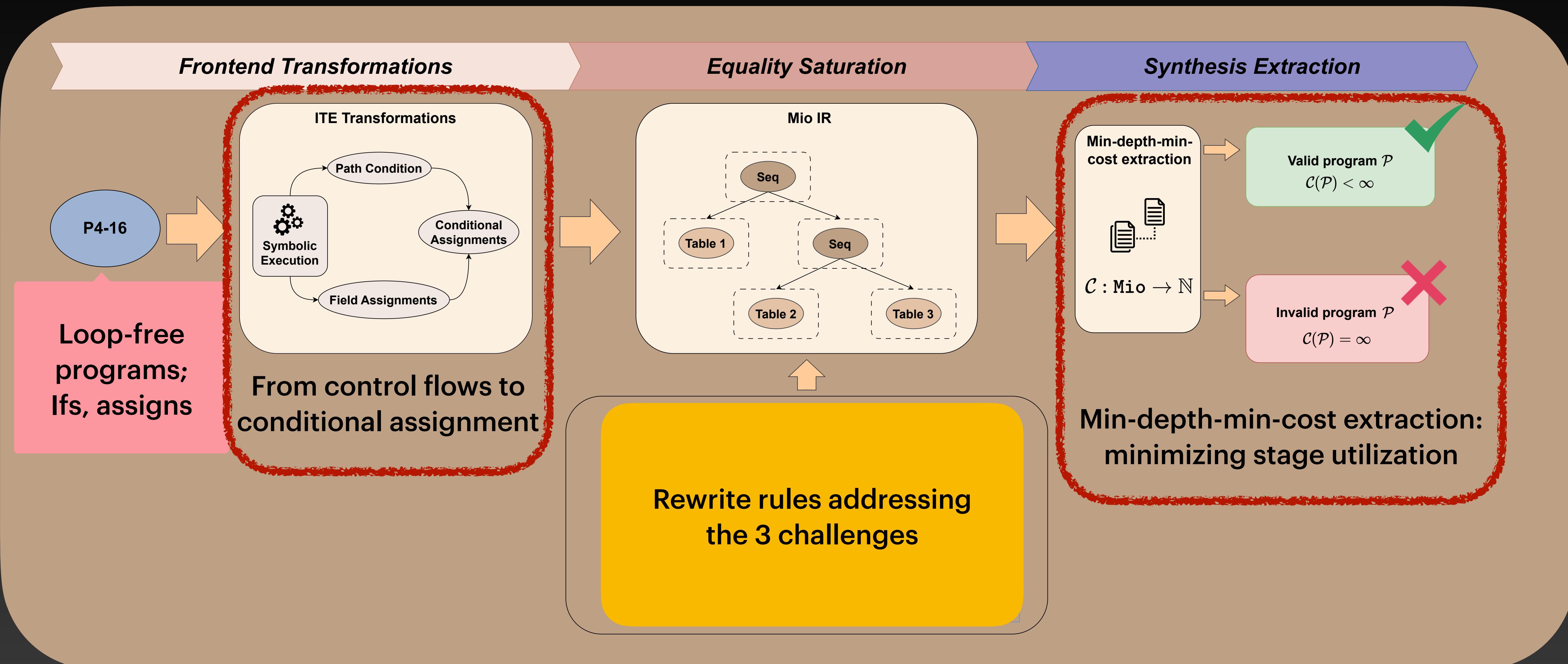


Figure 1: The workflow of the CaT compiler.

Previous work: CaT

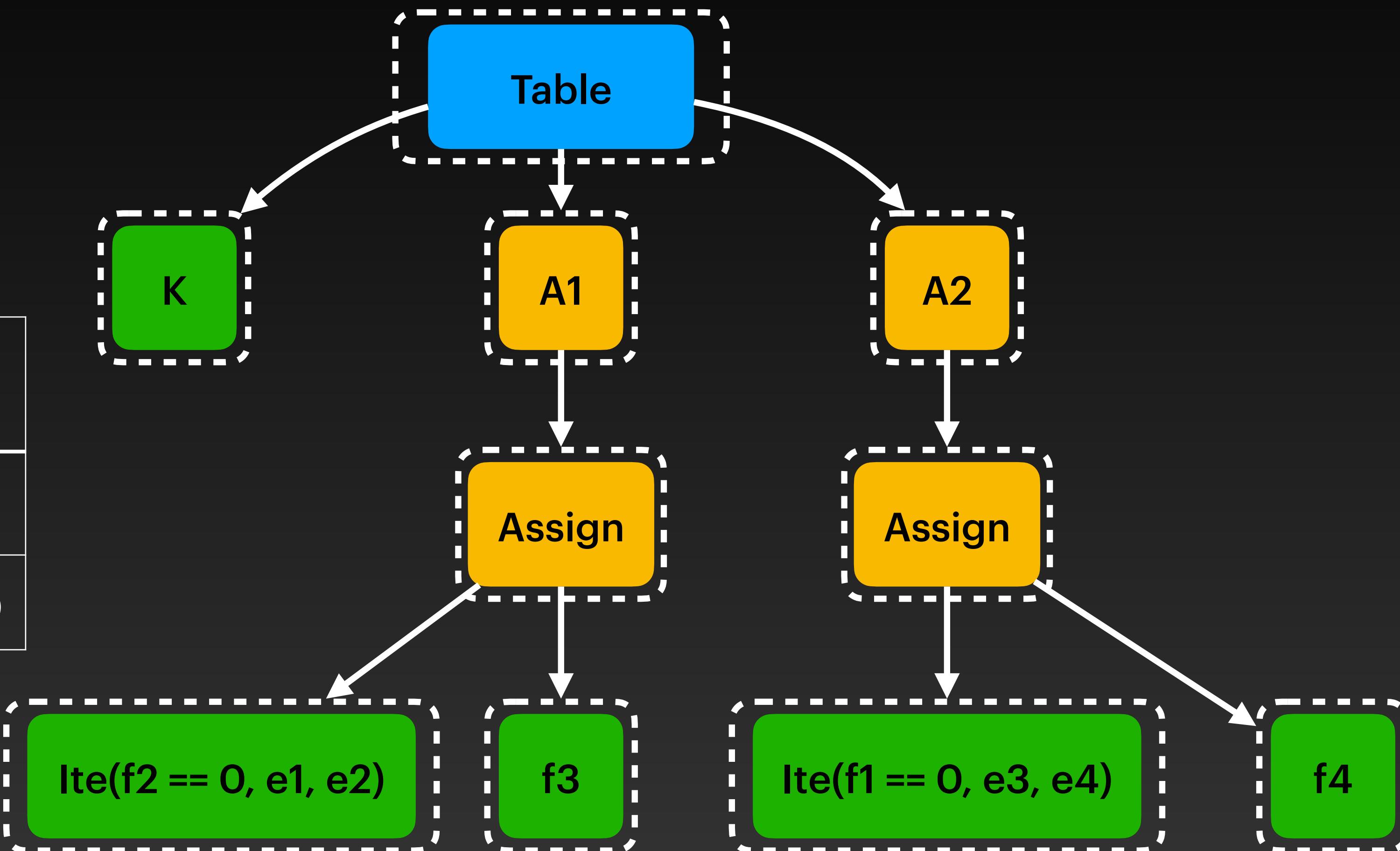


Resource synthesis via Equality Saturation



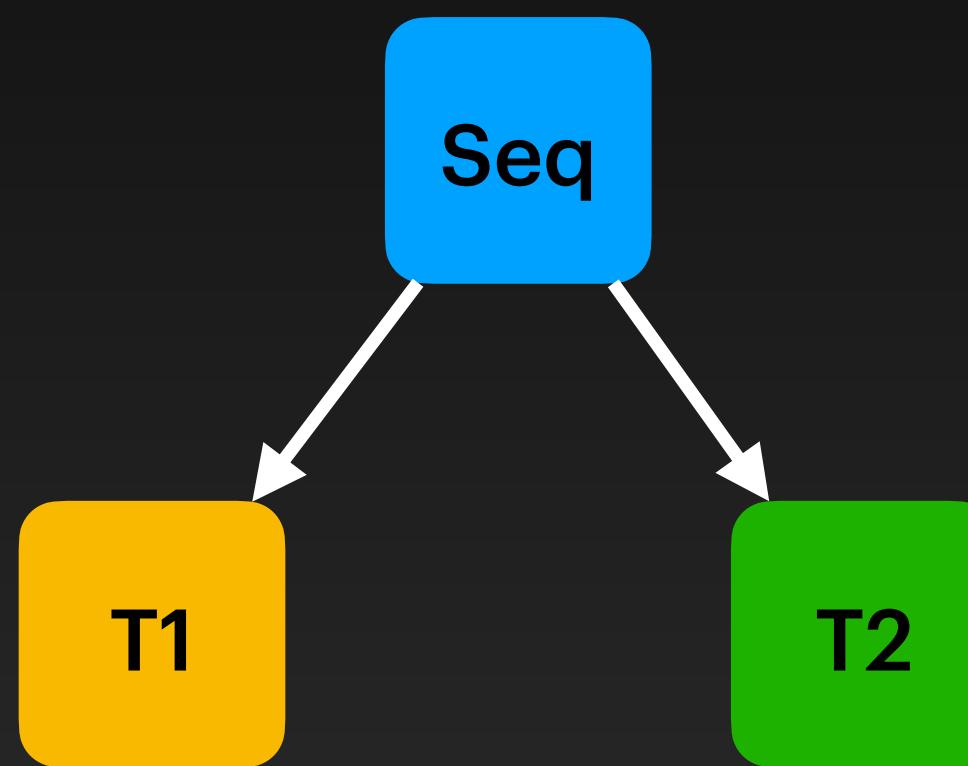
Frontend transformation

Match	Action
K	$\text{hdr.f3} = \text{ite}(\text{hdr.f2} == 0, \text{e1}, \text{e2})$ $\text{hdr.f4} = \text{ite}(\text{hdr.f1} == 0, \text{e3}, \text{e4})$

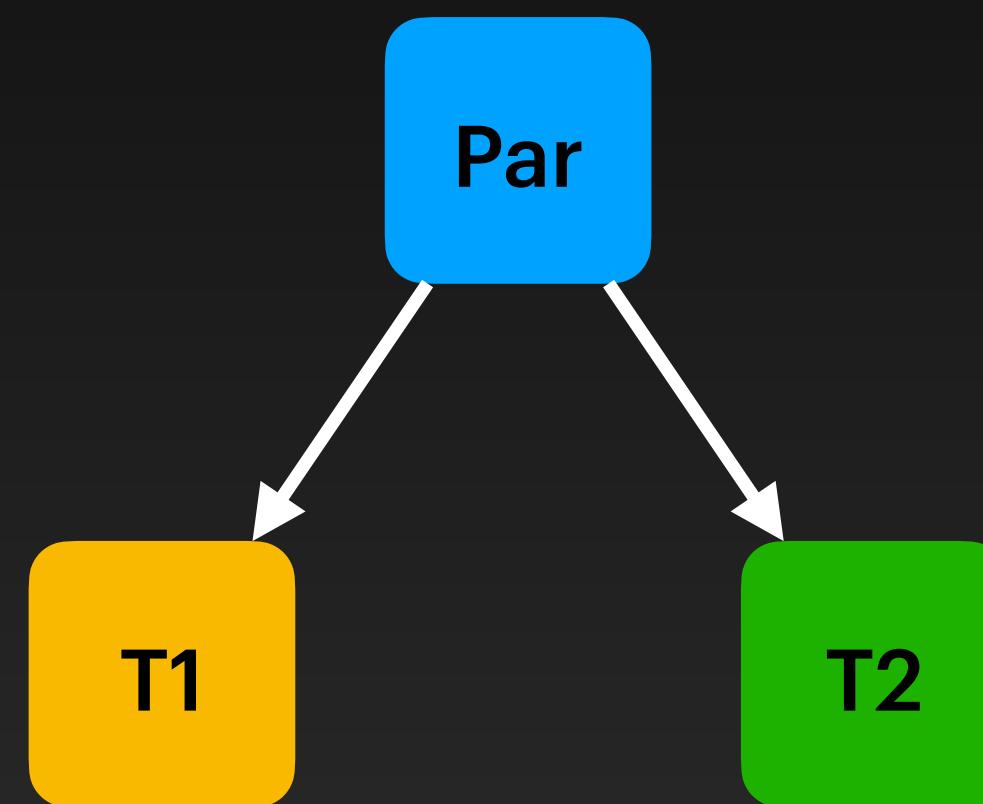


Frontend transformation

Introduce Table operators to allow table transformations



T1 must be placed before T2



T1 and T2 are put in the same stage

Rewrite rules

Challenge 1: Limited resource

General-purpose program transformations

$?x + ?y \Rightarrow ?y + ?x$
 $(?x + ?y) + ?z \Leftrightarrow ?x + (?y + ?z)$
 $?x + 0 \Rightarrow ?x$
 $\sim(?x \& ?y) \Rightarrow \sim?x \mid \sim?y$
 $?x \& ?x \Rightarrow ?x$
 $\mathbf{ite}(\mathbf{true}, ?x, ?y) \Rightarrow ?x$
 $\mathbf{ite}(\mathbf{false}, ?x, ?y) \Rightarrow ?y$
Etc...

52 Rules

Challenge 2: Table Dependencies

Table Transformations

Table parallelization
Subexpression lifting
Table merging
Etc...

10 Rules

Challenge 3: Different backends

Synthesis rewrites

1-1 to sketch grammars in
CaT (Gao et al.)

$?x + ?y \Rightarrow \mathbf{alu_add} ?x ?y$
if **mapped**(?x) & **mapped**(?y)

$?V = \mathbf{ite} (?x == ?y, ?x + ?z, ?x) \Rightarrow$
stateful_alu(if, ?V, ?x == ?y, ?x + z, ?x)
if ...

Tofino: 11 Rules Domino: 21 Rules

Table transformations

Goals:

- Explores different topological orders of applying tables
- Parallelizing table placements
- Decomposing computations
- Eliminate table dependencies

Table transformations

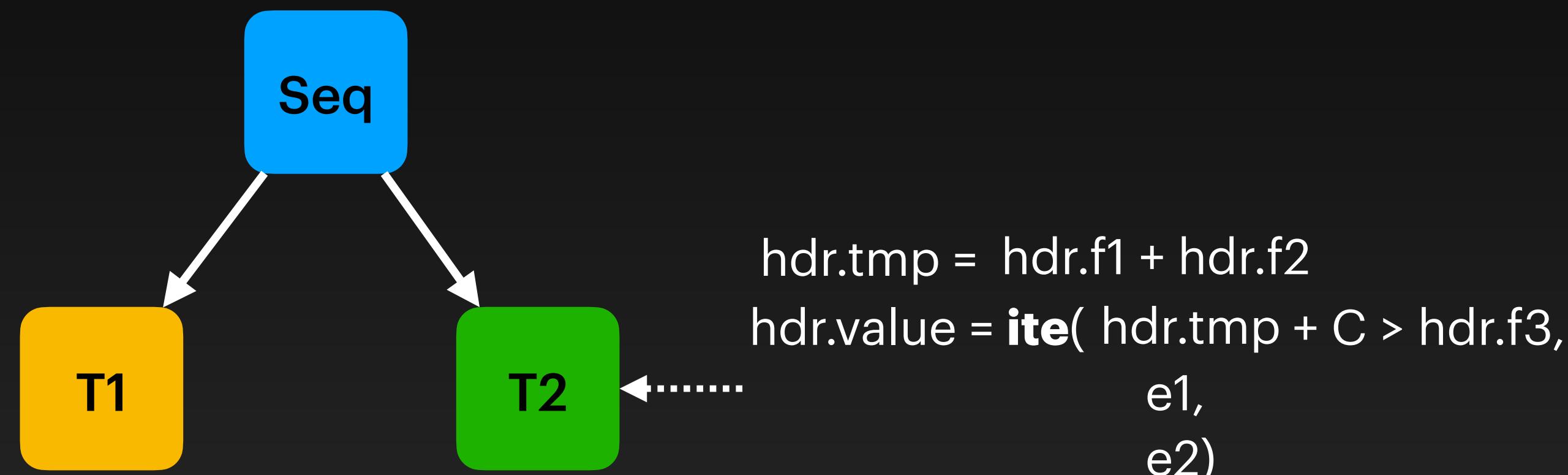
Decomposing computations



Lift computes with depth > 3

Table transformations

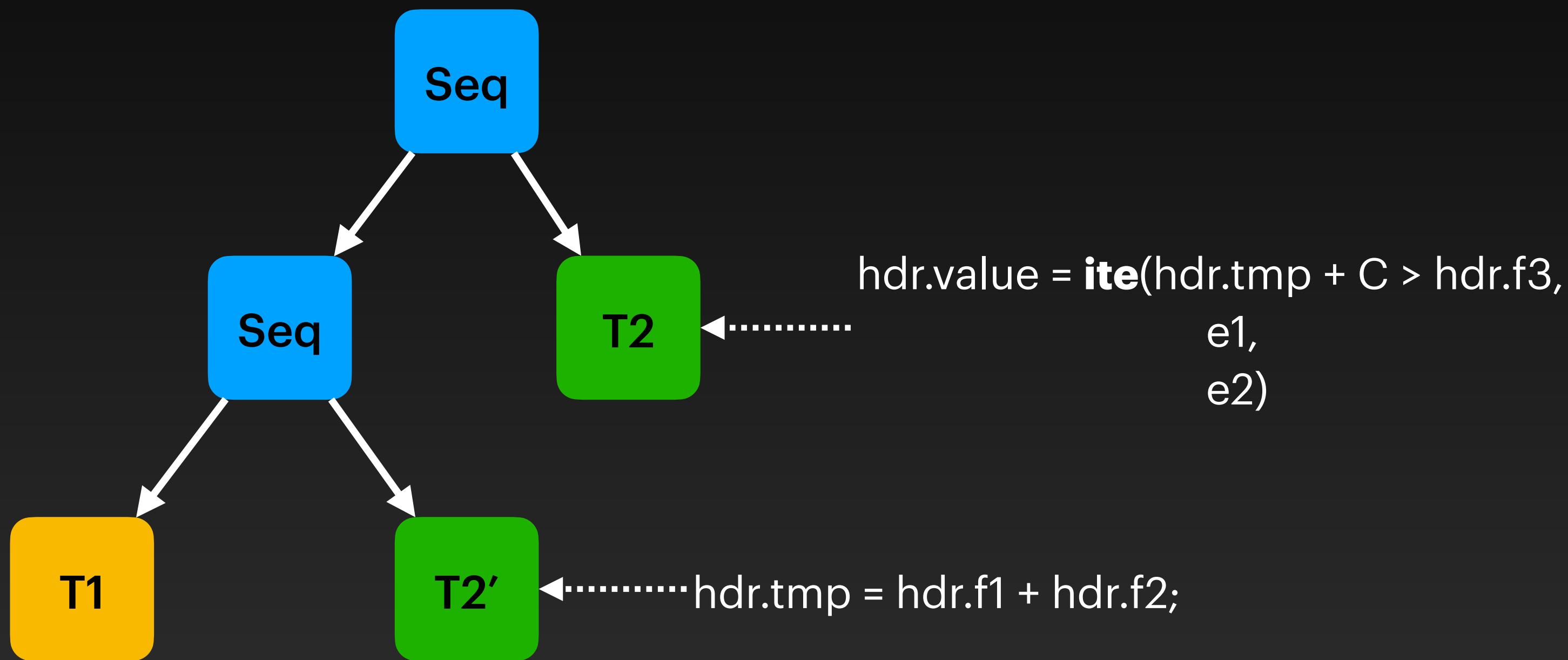
Decomposing computations



Lift computes with depth > 3

Table transformations

Decomposing computations



Can be done if split computation does not involve global variables

Synthesis rewrites

Target-dependent rewrite rules

Based on ALU Grammars used for Sketch-guided synthesis in CaT (Gao et al.)

Stateless ALUs

Pure computations

Stateful ALUs

May modify a register file in the ALU
(representing global variables)

SKETCH: a Syntax-guided Synthesis-based technique; Program sketches with holes

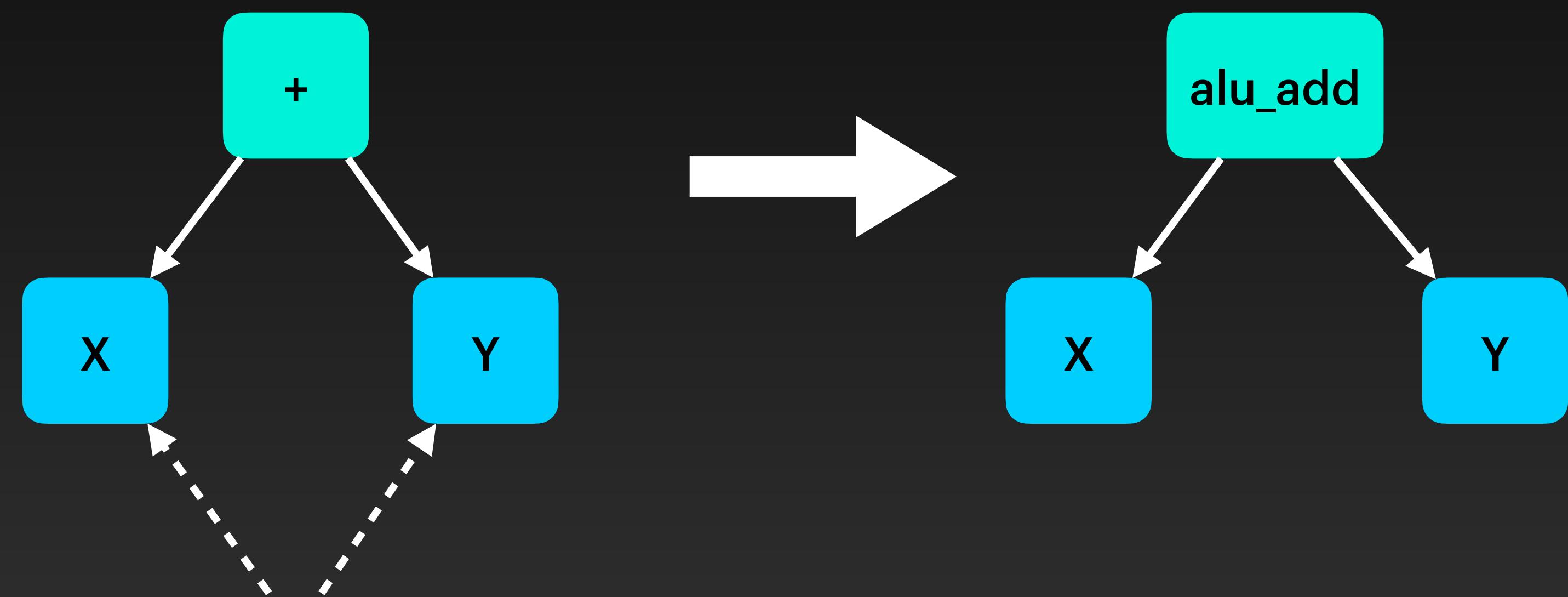
R. Alur *et al.*, "Syntax-guided synthesis," *2013 Formal Methods in Computer-Aided Design*, Portland, OR, USA, 2013, pp. 1-8, doi: 10.1109/FMCAD.2013.6679385.

Solar-Lezama, A. (2009). The Sketching Approach to Program Synthesis. In: Hu, Z. (eds) *Programming Languages and Systems*. APLAS 2009. Lecture Notes in Computer Science, vol 5904. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-10672-9_3

Synthesis rewrites

Stateless ALUs

Inductively defined based on Sketch grammars



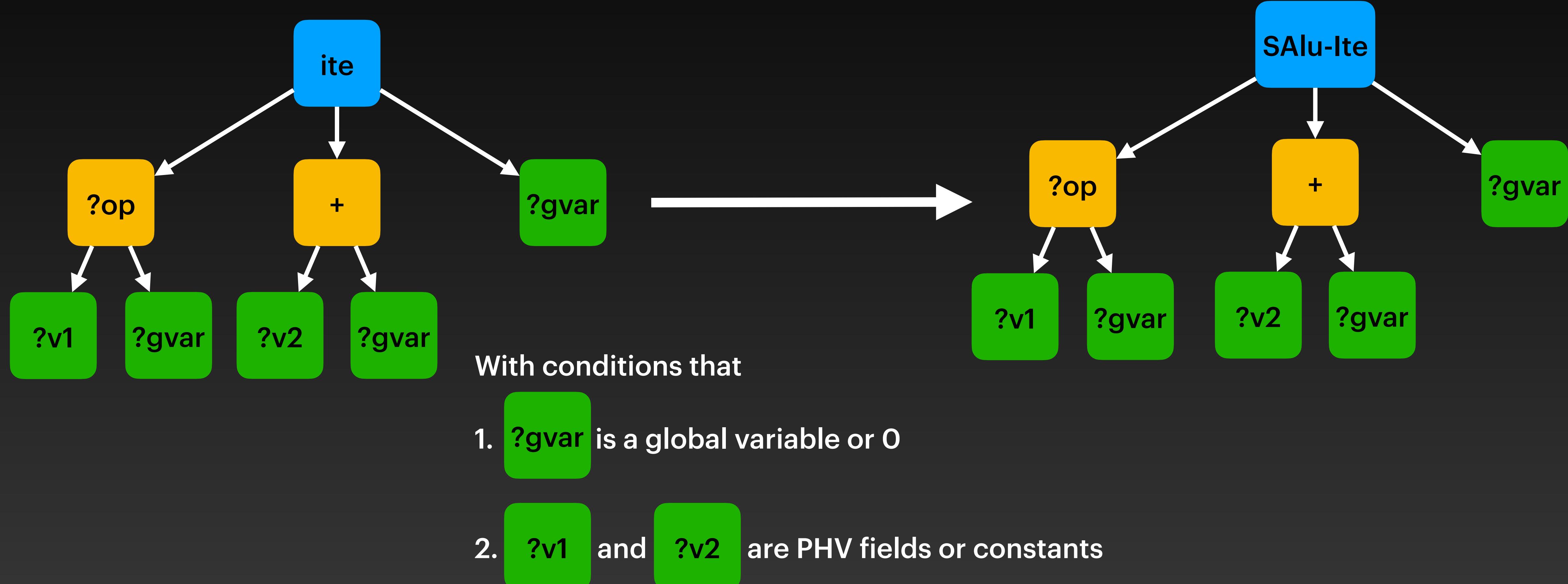
Base Case: X and Y are literals or PHV field variable

Induction Step: X and Y represent stateless ALU computations

Synthesis rewrites

Stateful ALUs

Based on Sketch grammars



Limitations: a global variable is not read/written by two different tables

Rewrite rules

Efficiently explores the space of candidate mappings by composing the rewrite rules via Equality Saturation

General-purpose program transformations

$?x + ?y \Rightarrow ?y + ?x$
 $(?x + ?y) + ?z \Leftrightarrow ?x + (?y + ?z)$
 $?x + 0 \Rightarrow ?x$
 $\sim(?x \& ?y) \Rightarrow \sim?x \mid \sim?y$
 $?x \& ?x \Rightarrow ?x$
 $\mathbf{ite}(\mathbf{true}, ?x, ?y) \Rightarrow ?x$
 $\mathbf{ite}(\mathbf{false}, ?x, ?y) \Rightarrow ?y$
Etc...

52 Rules

Table Transformations

Table parallelization
Subexpression lifting
Table merging
Etc...

10 Rules

Synthesis rewrites

1-1 to sketch grammars in
CaT (Gao et al.)

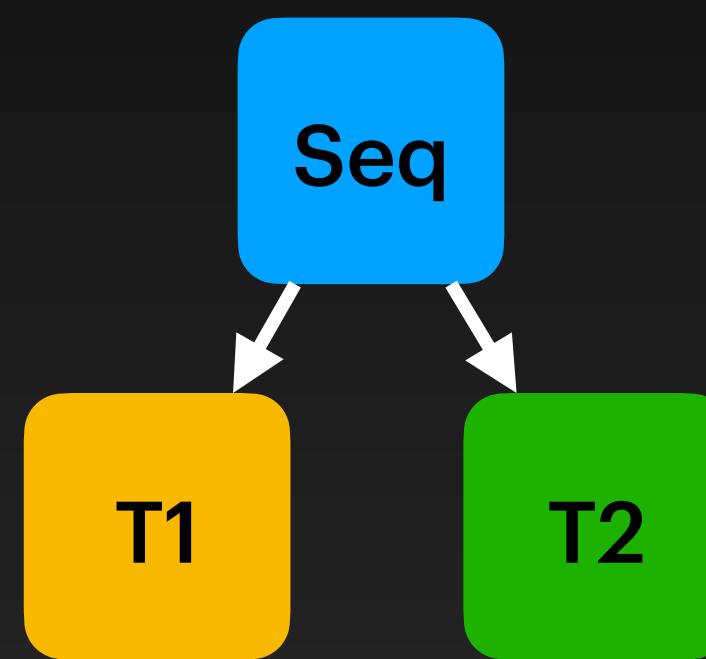
$?x + ?y \Rightarrow \mathbf{alu_add} ?x ?y$
if **mapped**(?x) & **mapped**(?y)

$?V = \mathbf{ite}(?x == ?y, ?x + ?z, ?x) \Rightarrow$
stateful_alu(if, ?V, ?x == ?y, ?x + z, ?x)
if ...

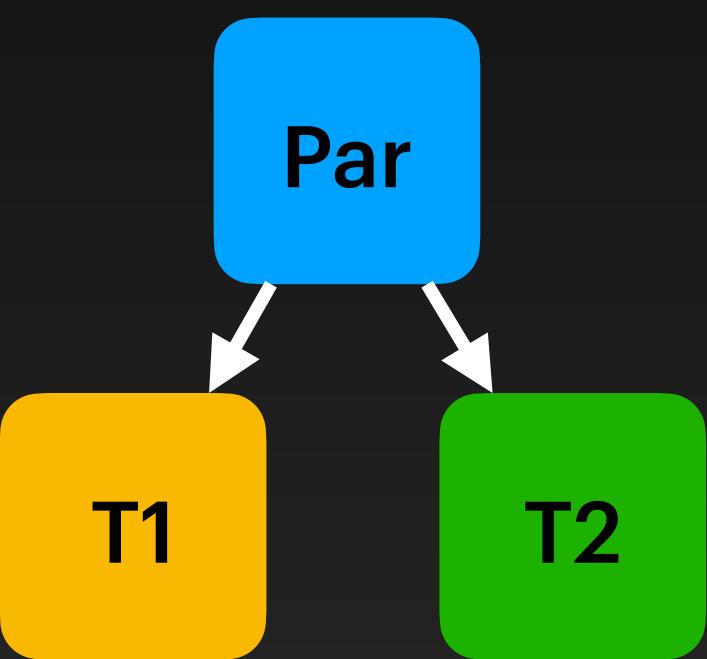
Tofino: 11 Rules Domino: 21 Rules

Extraction

Goal: Extract min-depth computation tree



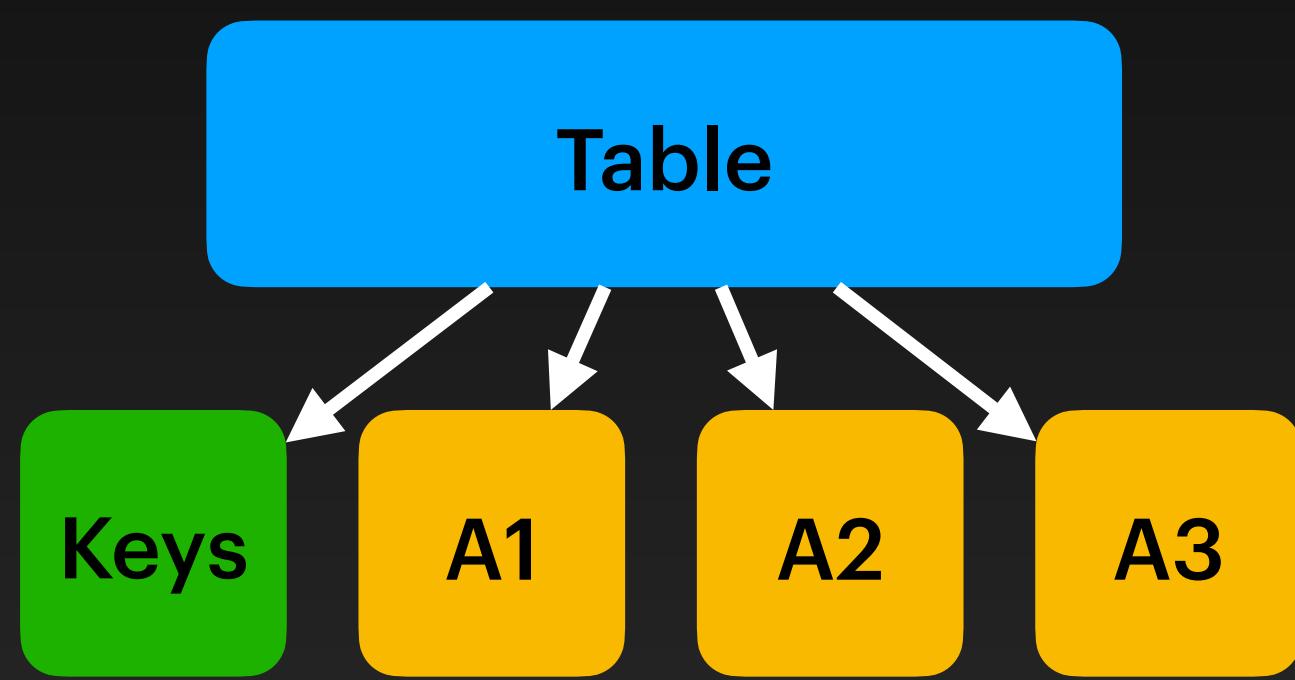
$$\text{cost}(T_1) + \text{cost}(T_2)$$



$$\max(\text{cost}(T_1), \text{cost}(T_2))$$

Extraction

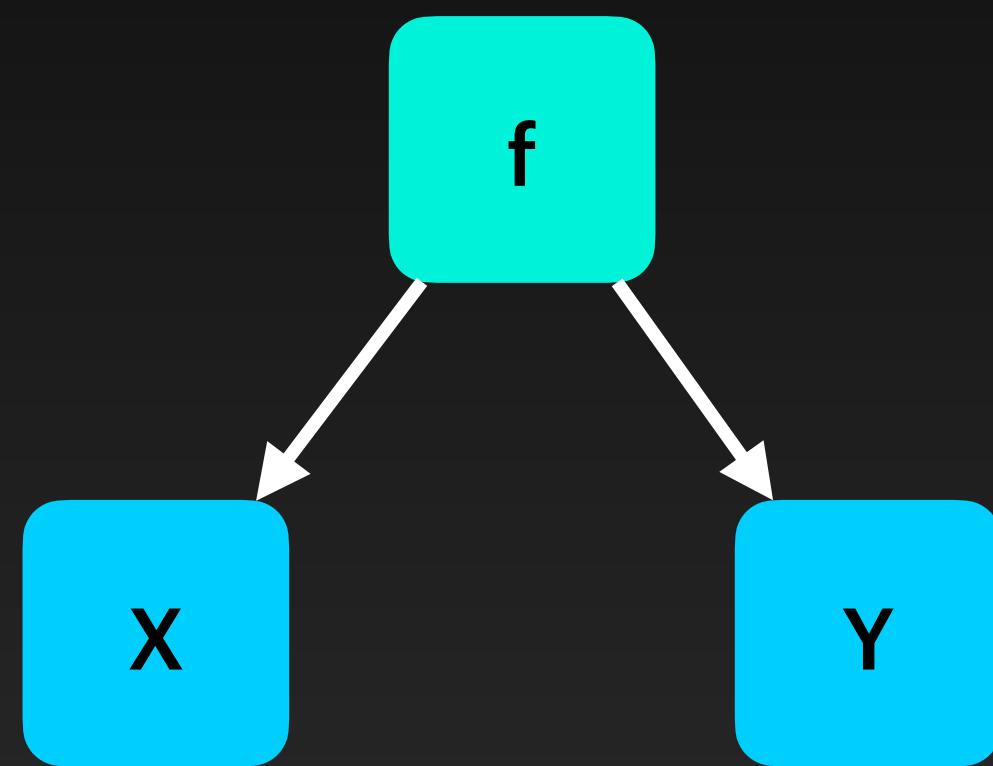
Goal: Extract min-depth computation tree



$$\max_i (\text{Cost}(A_i))$$

Extraction

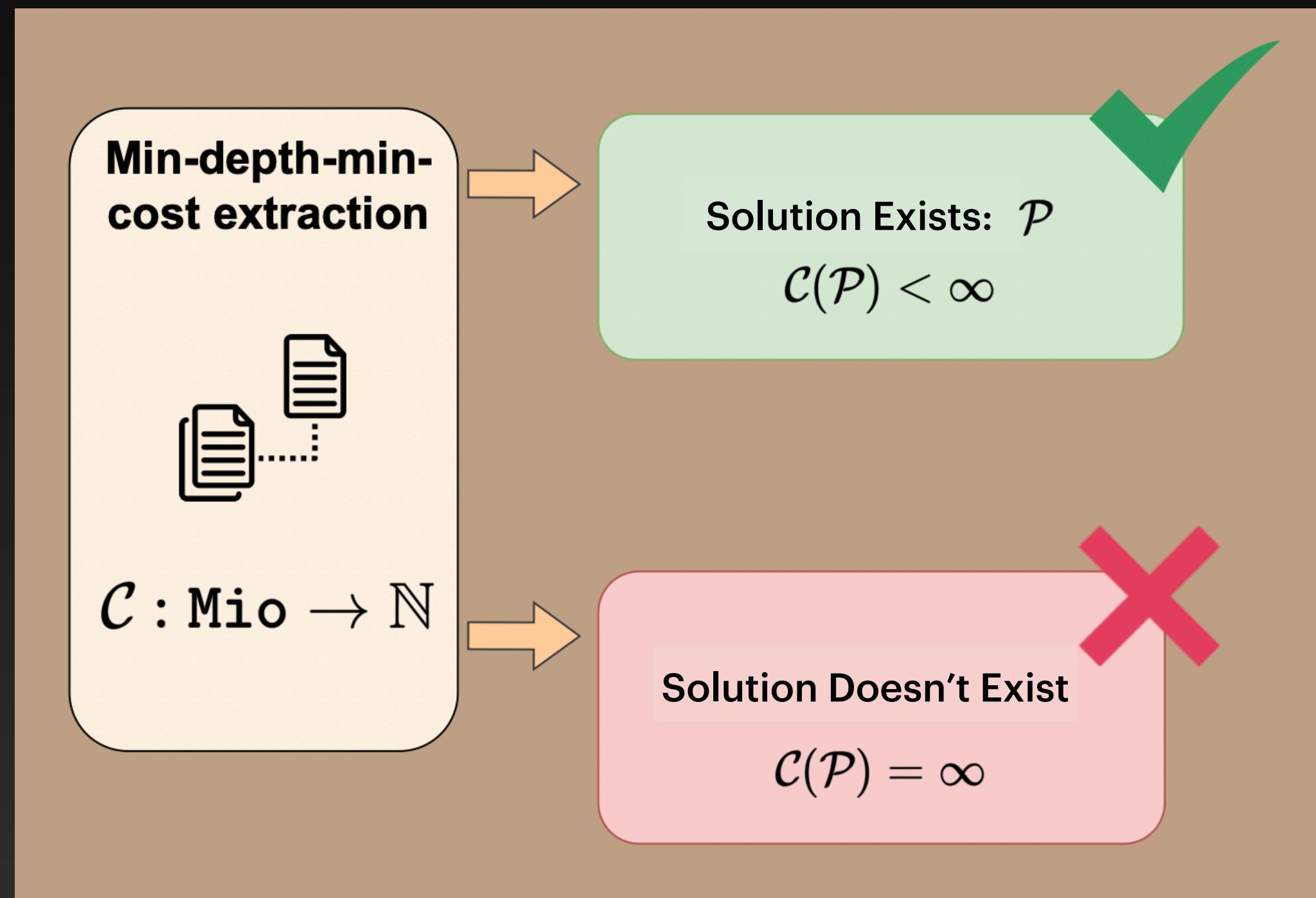
Goal: Extract min-depth computation tree



$$\text{ite} \left(\text{mapped}(f(X, Y)), \max \left(\text{Cost}(X), \text{Cost}(Y) \right) + 1, \infty \right)$$

Only allow extracting computations that are already mapped to target backends

Extraction



$\mathcal{C}(\mathcal{P})$ = Minimum number of stages required to map \mathcal{P}

Evaluations

RQ1: Efficiency of CatsTail: synthesis time compared with the previous work CaT (Gao et al.)

RQ2: Efficacy of CatsTail: stage utilization compared with CaT

RQ3: Does the extraction always succeed?

Evaluations

RQ1: Efficiency of CatsTail: synthesis time compared with the previous work CaT (Gao et al.)

Experiments setup:

Target Backends: Intel Tofino and Domino (Banzai) ALUs

Input programs: 8 P4 programs with real-word applications, including:

Rate control protocol, Packet sampling, Flowlet Switching,
Stateful firewall, Blue increase/decrease, Marple flow

Rewrite Rules:

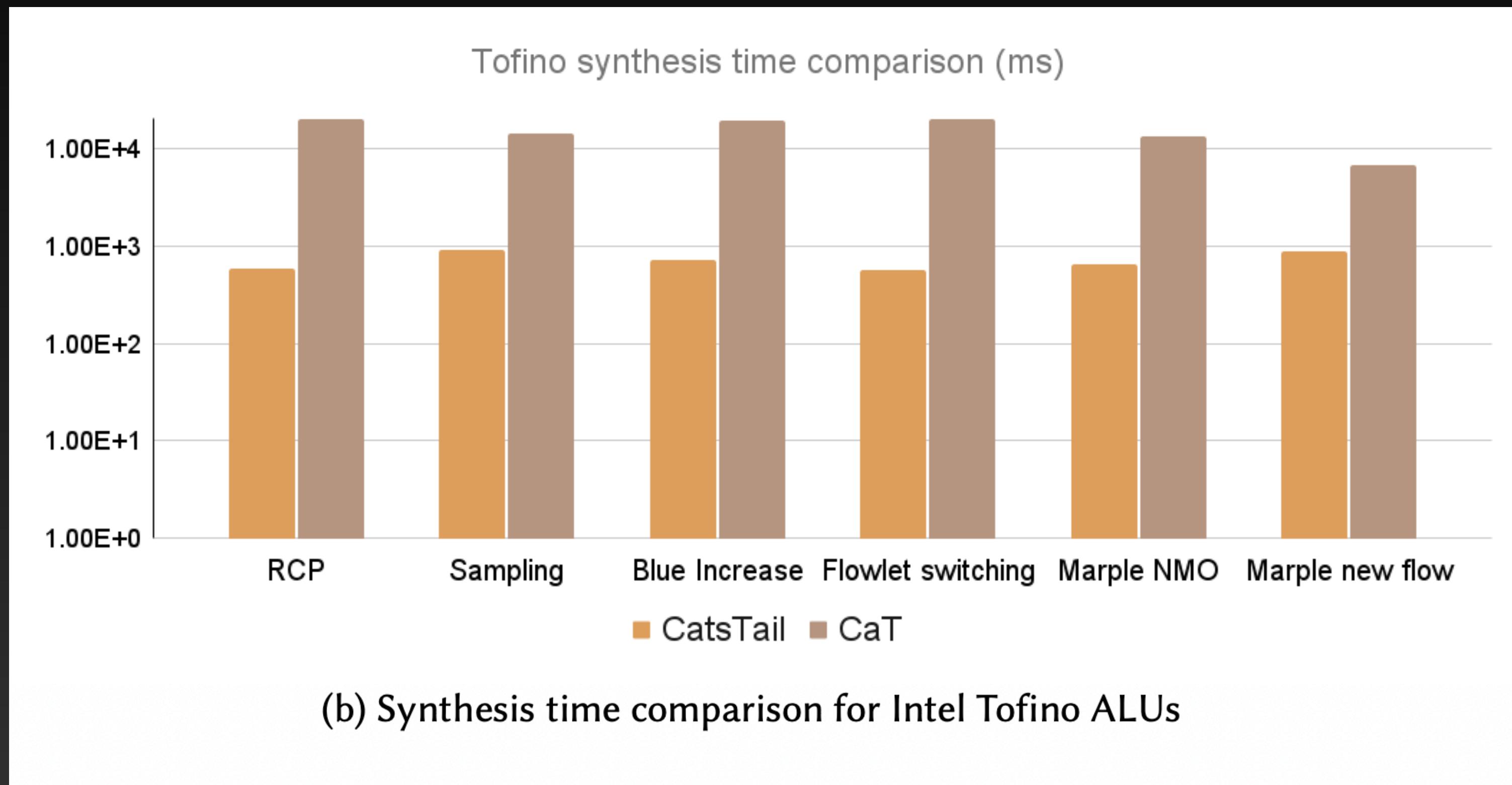
For the Tofino backend, we enable all the synthesis rewrite

For the Domino backend, we ran two sets of experiments:

1. Full: All synthesis rewrite rules
2. Sk: synthesis rewrite rules corresponding to the sketch grammar
CaT used in their benchmark

Evaluations

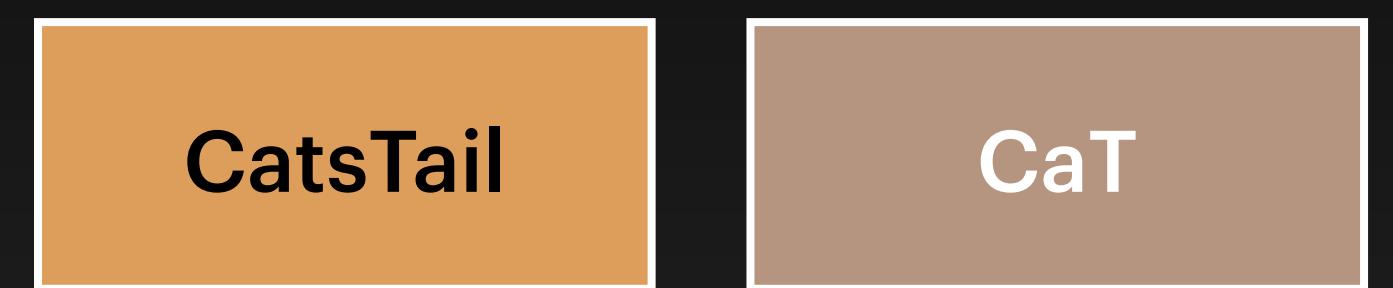
RQ1: Efficiency of CatsTail: synthesis time compared with the previous work CaT (Gao et al.)



X: Benchmark cases.

Y: Synthesis time (ms), in log-scale

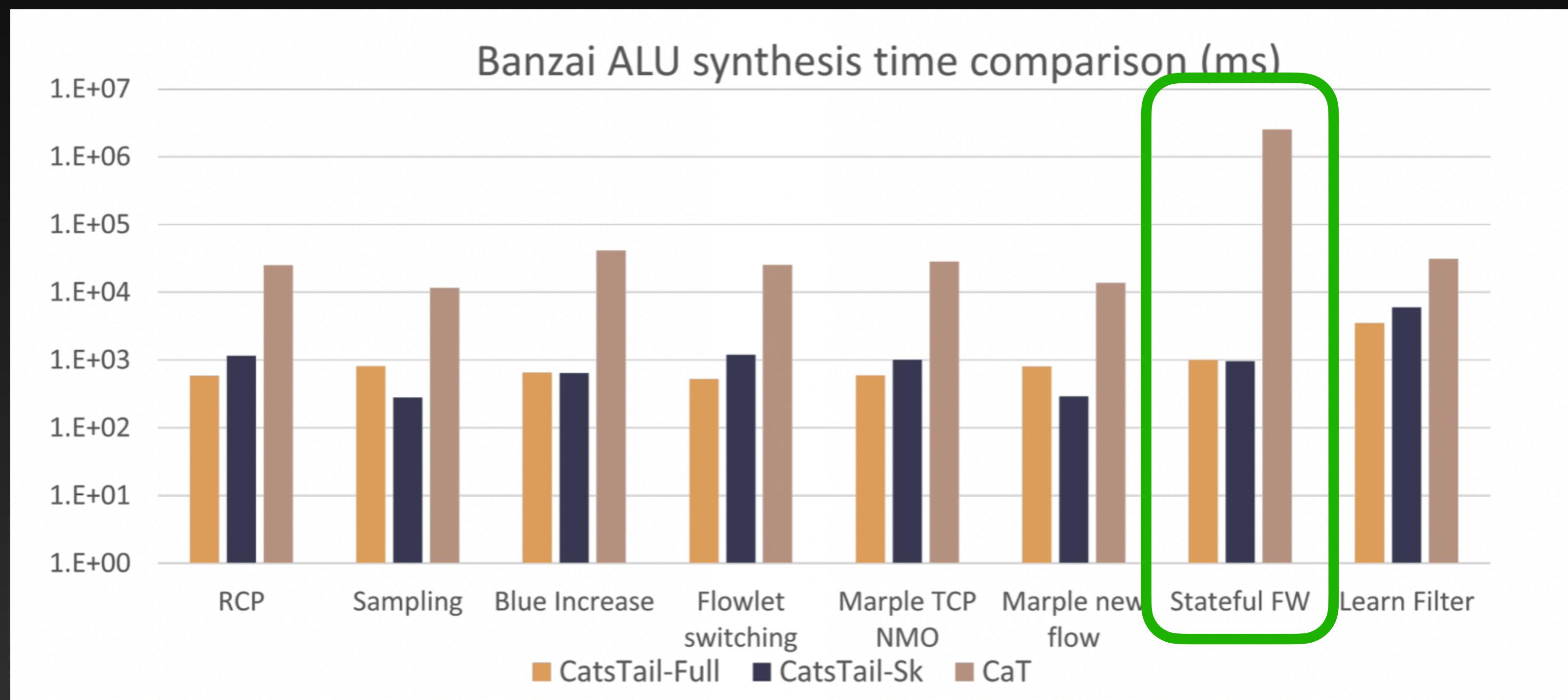
Successfully synthesized



**~an order of magnitude
faster in synthesis**

Evaluations

RQ1: Efficiency of CatsTail: synthesis time compared with the previous work CaT (Gao et al.)



Successfully synthesized
Orders of magnitude faster

CatsTail-Full

CatsTail ran with all rewrite rules

CatsTail-Sk

**Similar to CatsTail-Full except
the synthesis rules only include those
corresponds to sketches used in CaT**

CaT

CaT synthesis time

Evaluations

RQ2: Efficacy of CatsTail: stage utilization compared with CaT

Table 1. Comparison of the number of stages required to map the synthesized program given by CATSTAIL and CaT [Gao et al. 2023] to Intel Tofino switches and Domino switches.

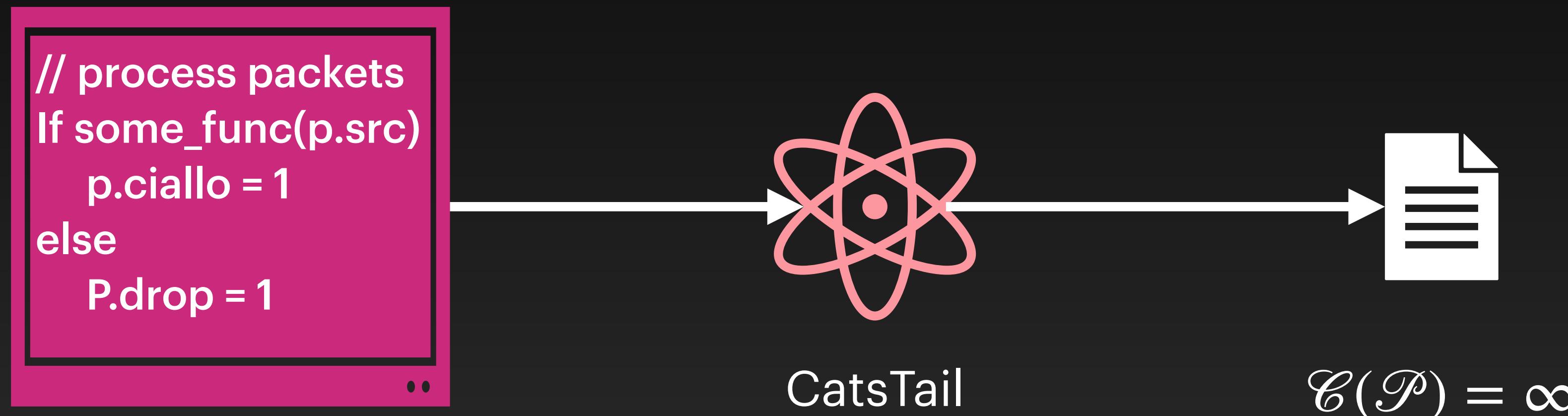
Benchmark	# Stages on Domino		# Stages on Tofino	
	CATSTAIL	CaT	CATSTAIL	CaT
RCP	2	2	1	1
Sampling	2	2	1	1
Blue Increase	4	4	1	1
Flowlet Switching	3	3	2	2
Marple Flow NMO	2	3	2	2
Marple New Flow	2	2	1	1
Stateful Firewall	4	4	-	-
Learn Filter	3	3	-	-

Same numbers of stage utilization

Nested ifs not supported by
Tofino switch

Evaluations

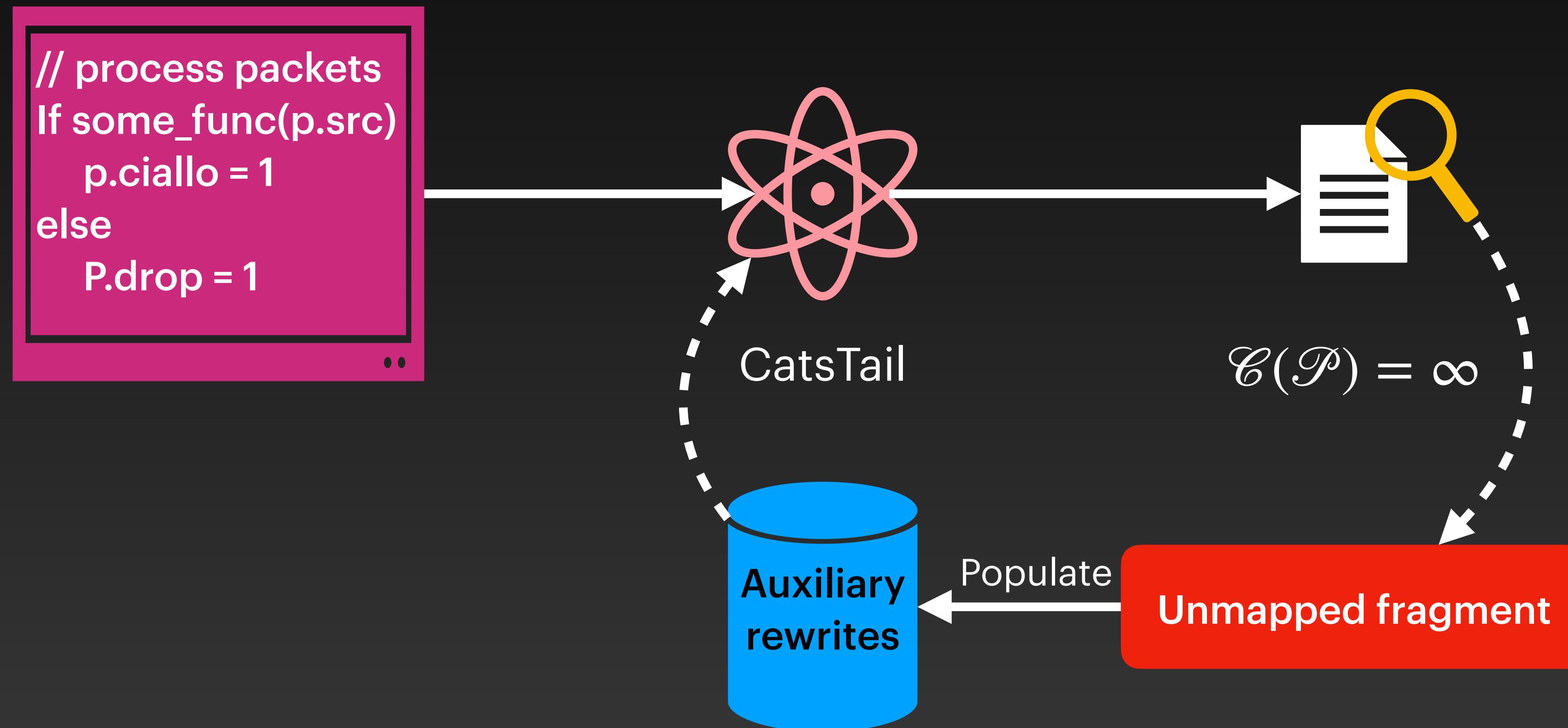
RQ3: Does the extraction always succeed?



Incompleteness of general purpose / table transformation rules

Evaluations

RQ3: Does the extraction always succeed?



Evaluations

RQ1: Efficiency of CatsTail: synthesis time compared with the previous work CaT (Gao et al.)

Orders of magnitude faster compared with CaT, thanks to the scalability of egg

RQ2: Efficacy of CatsTail: stage utilization compared with CaT

Stage utilization is as good as CaT

RQ3: Does the extraction always succeed?

No, but we can work around



Report

[https://www.cs.princeton.edu/~dh7120/
assets/papers/COS539Report.pdf](https://www.cs.princeton.edu/~dh7120/assets/papers/COS539Report.pdf)



Prototype

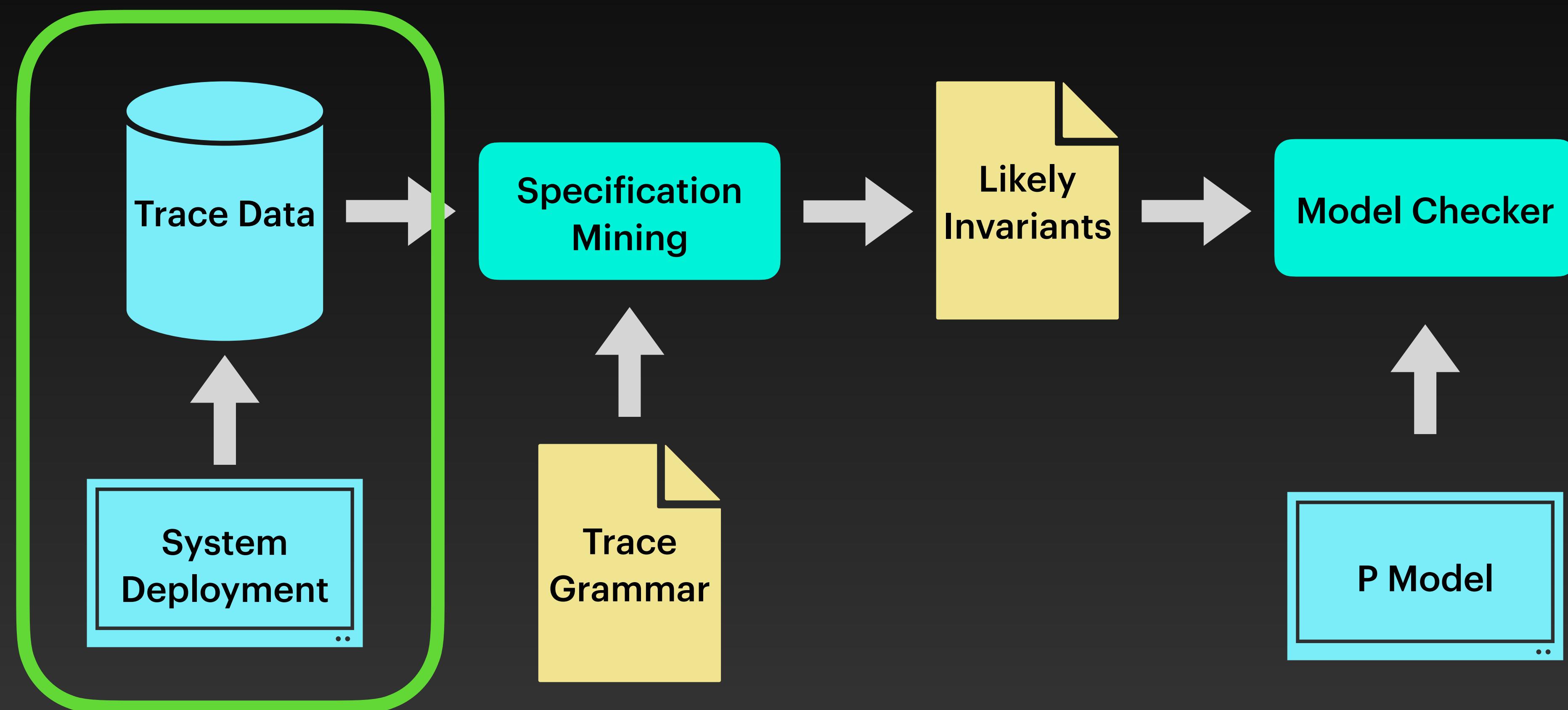
<https://github.com/AD1024/CatsTail/>

Outline

1. Brief introduction to equality saturation
2. Term Extraction for equality saturation **(Part A)**
3. Applying equality saturation for network resource synthesis **(Part B)**
4. (If time permits) Ongoing project of invariant synthesis for distributed systems

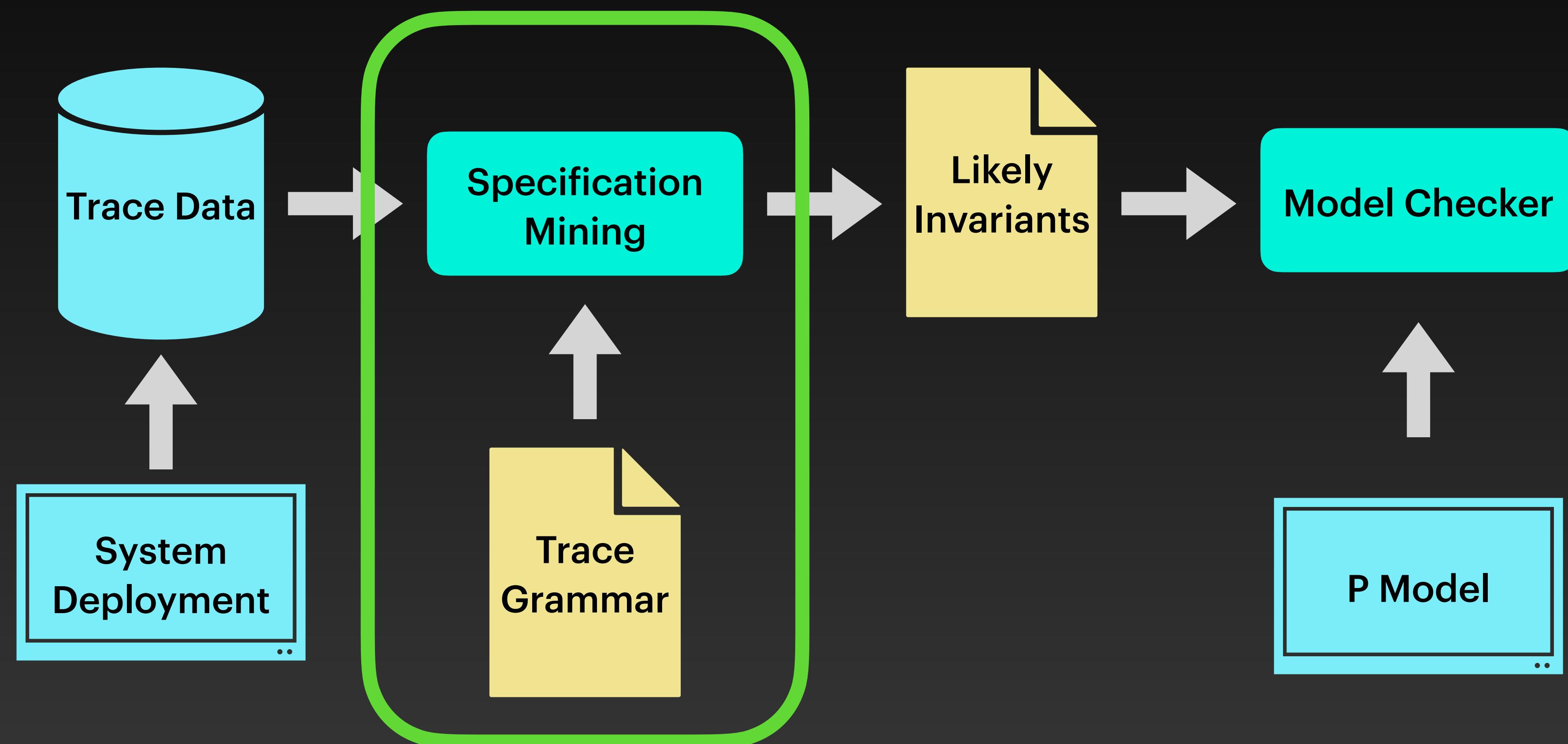
Recent project: PIInfer

Learning invariants for distributed systems from traces



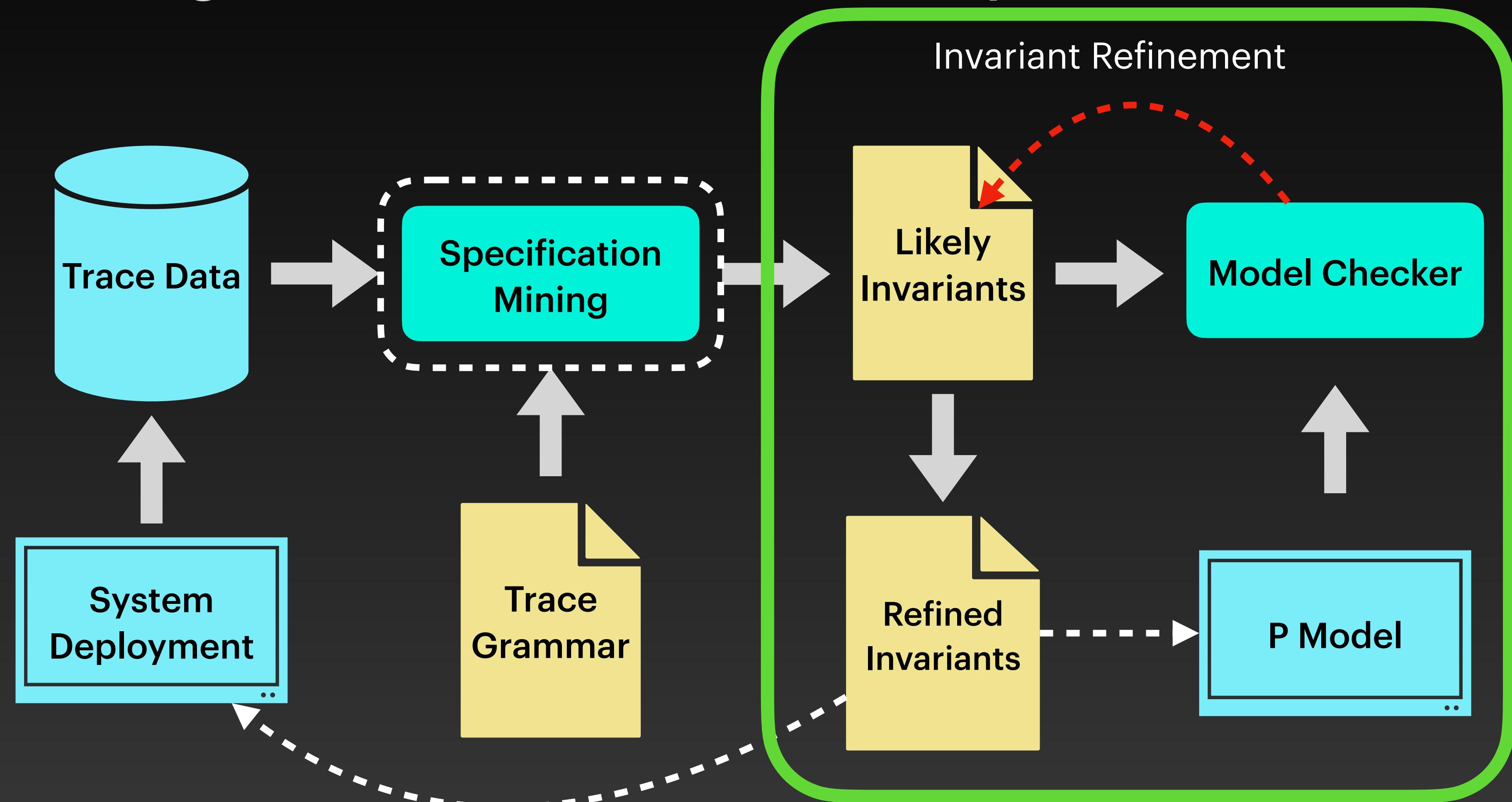
Recent project: PIInfer

Learning invariants for distributed systems from traces



Recent project: PIInfer

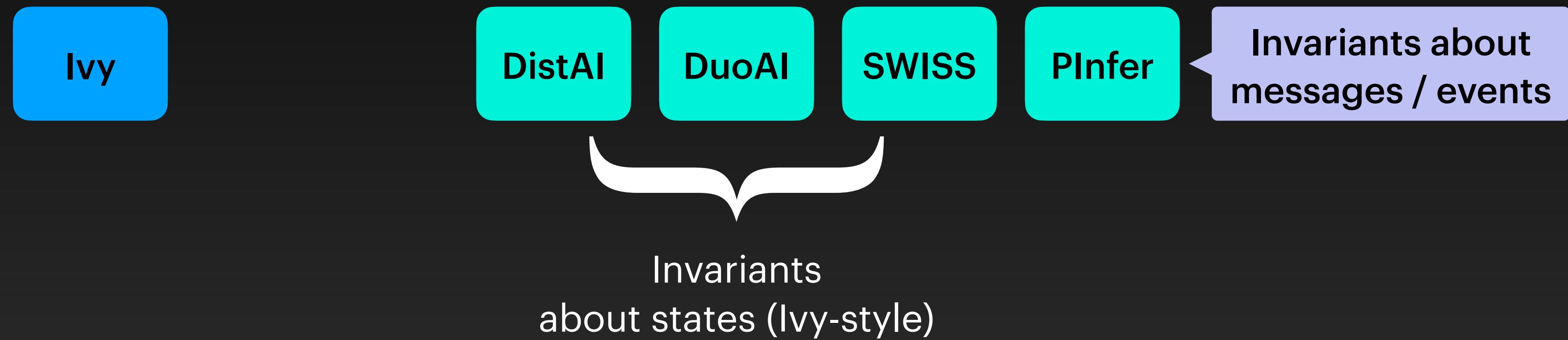
Learning invariants for distributed systems from traces



Invariant learning: Related works

Protocol Definition +
Invariants checking

Enumerate combinations of
predicates and connectives



K. McMillan, O. Padon, "Ivy: A Multi-modal Verification Tool for Distributed Algorithms," in Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II, 2020, pp. 190–202.

Travis Hance, Marijn Heule, Ruben Martins, Bryan Parno. "Finding Invariants of Distributed Systems: It's a Small (Enough) World After All." 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). USENIX Association, 2021.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh. "DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols." 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). USENIX Association, 2022.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, Gabriel Ryan. "DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols." 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association, 2021.

Invariant learning

PIInfer

Challenges:

1. Huge search space: many valid predicates over events and payloads

Brute-force enumeration leads to vacuously true/false invariants, which are not useful for production systems

Trace Grammar that focuses useful predicates

2. Efficiency: enumerating logical connectives is computationally intractable

Formulate invariant learning as a boolean function learning problem

Q & A