CatsTail: Packet programs synthesis via Equality Saturation

DEYUAN MIKE HE, Princeton University, USA
YINWEI DAI, Princeton University, USA

The limited on-chip resources bring several challenges to compiling switch programs onto target hardware, which makes it difficult for switch users to run and test their programs. Previous works attempted to formulate the compilation problem as program synthesis problems and synthesize code that can be mapped to target hardware by querying synthesis oracles. To generate programs that minimize stage allocation, the synthesis oracle is invoked multiple times, which drastically slows down the compilation speed. Because of this, real-time debugging on real hardware is nearly impossible. In this project, we propose CatsTail, an equality saturation-based resource synthesizer for P4 programs that achieves orders of magnitude compilation speed up compared with previous works. CatsTail utilizes a data structure called e-graphs, which represents equivalent representations of programs, and encodes program transformations as syntactic rewrite rules. In addition to general-purpose program transformations, CatsTail also introduces a set of target-dependent rewrite rules that match certain computation patterns and generate programs that are directly mappable to target architectures.

1 INTRODUCTION

Mapping switch programs to programmable switches has been extensively studied since the emergence of reconfigurable switches [Bosshart et al. 2013] and programming languages targeting their models [Bosshart et al. 2014; Gao et al. 2020]. The major challenge of mapping user-defined programs to programmable switches is due to the limited computing resources: the vanilla input program may require more stages than the hardware provides because of the table dependencies or a giant match table may not fit into one physical stage.

A previous work, CaT [Gao et al. 2023], proposes to use sketch-guided synthesis (SKETCH) [Solar-Lezama 2008] to generate a min-depth program to reduce stage usage. CaT first performs some generic program transformations and then queries SKETCH with grammars of computations that are supported by the target architecture. However, SKETCH requires massive parallelism and hardly scales if the computation is complicated because it has to search through a large space of candidates with the majority that are trivially incorrect (e.g. incorrect constants). According to the evaluation of CaT, the resource synthesizer takes over 1,000 seconds to generate a valid result for a simple program (e.g. stateful firewall). This fundamental drawback of SKETCH is because the synthesizer only utilizes the semantics of the input program without considering its *structure* to guide synthesis.

In this project, we introduce CatsTail, a resource synthesizer based on a novel technique from the programming languages community called equality saturation (EqSat) [Tate et al. 2009]. Informally, equality saturation takes an initial program and a set of *syntactic* rewrite rules that preserve program semantics as inputs. Then EqSat applies the rewrite rules to the initial program while memorizing semantically equivalent programs modulo rewrites. EqSat terminates when it hits a time limit, or the space of programs equivalent to the initial input program has been fully explored. The workflow of CatsTail is shown in Figure 1.

CatsTail takes a P4 program as input. In the first step, CatsTail applies If-then-else (ITE) transformation (§ 3.1) by symbolic execution and compiles the P4 programs to a dataflow-based intermediate representation (IR) called `Mio` IR. Then, CatsTail runs equality saturation using `egg` on the program in `Mio` IR with 3 sets of rewrite rules: (1) general-purpose program transformations, (2) table transformations, and (3) target-dependent synthesis rules (§3.3). Finally, after EqSat terminates

Authors' addresses: Deyuan Mike He, Department of Computer Science, Princeton University, USA, dh7120@cs.princeton.edu; Yinwei Dai, Department of Computer Science, Princeton University, USA, yinweid@cs.princeton.edu.

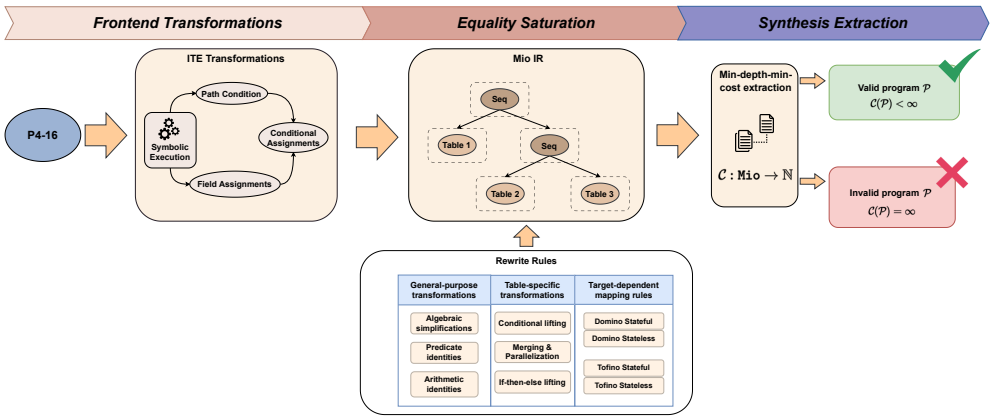


Fig. 1. The workflow of CatsTail synthesizer

(or timeout), the e-graph will be populated with many candidate programs, and we have to choose one candidate out of the entire space. Therefore, we implement a target-aware synthesis extraction (§ 3.4) which encodes minimum computation tree depth as the cost function.

The rest of this report is arranged as follows:

- § 2 briefly goes through the background of equality saturation and egg.
- § 3 explains the design of CatsTail in detail
- § 4 shows the proof of completeness of the target-dependent synthesis rules
- § 5 discusses the evaluation of CatsTail, including the performance improvement comparing CatsTail with SKETCH

2 BACKGROUND

2.1 Equality Saturation

Traditional term rewriting system applies the rewrite rules one at a time and forgets the original term. These term rewriting systems are sensitive to the order in which rewrite rules are applied because some rewrite rules can disable opportunities to run some other rewrite rules. This is a well-known problem called the *Phase ordering problem* in the compiler community. Equality saturation [Tate et al. 2009] (EqSat) is a technique to mitigate this problem. Instead of running the rewrite rules destructively, EqSat applies the rewrites iteratively and keeps both the original term and the new term. EqSat utilizes a data structure called *e-graph* [Nelson and Oppen 1980] that maximizes sharing among sub-expressions to represent the set of equivalent terms efficiently.

2.1.1 e-graphs. E-graphs are made up of *e-classes* and *e-nodes* that represent some terms.

- *Terms.* A term t can be a literal, a symbol, or a function application $f(\vec{x})$ where $x_1, x_2, \ldots, x_n$ are also terms.
- *E-classes.* e-classes are sets of e-nodes. An e-class C represents a term t if there is an e-node in C that represents t.
- *E-nodes.* e-nodes represent terms. If an e-node n represents a literal or a symbol, then n does not have children and n is the literal or the symbol; otherwise, n represents a term $f(\vec{x})$. In this case, n is f and each child C_i of n is an e-class that represents the term x_i.

2.1.2 Syntactic Rewrite. Rewrite rules for terms represented in e-graphs are defined in the form of a left-hand-side pattern (LHS) and a right-hand-side pattern (RHS): $p \Rightarrow q$, denoting matching on

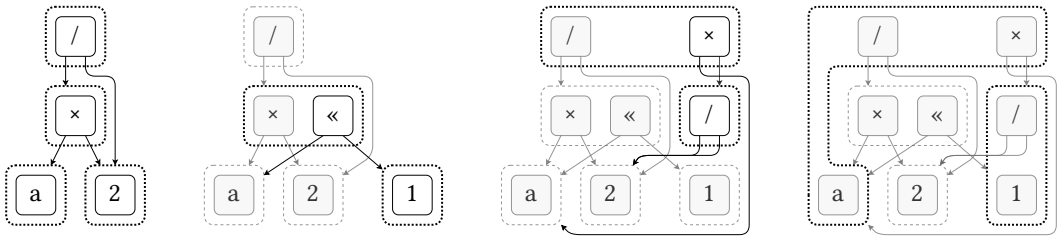


Fig. 2. An example e-graph borrowed from [Willsey et al. 2021]. The first e-graph contains the original input term. The second e-graphs shows the result of applying $?\alpha \times 2 \Rightarrow ?\alpha << 1$. The third e-graph shows the result of applying $?\alpha \times ?\beta/?\eta \Rightarrow ?\alpha \times (?\beta/?\eta)$. Finally, the last e-graph shows the result of applying $?\alpha \times ?\beta/?\beta \Rightarrow ?\alpha$ and $?\alpha/?\alpha \Rightarrow 1$.

p and then instantiating a new term q. LHS and RHS can be terms that contain pattern variables, which stands for wildcard pattern matching. For example, the rewrite rule $?\alpha \times 2 \Rightarrow ?\alpha << 1$ contains a pattern variable $?\alpha$. It matches any term where the operator is $\times$ and the right-hand-side operand is 2 and then instantiates a new term where the operator is $<<$ with the same operands. The instantiated e-node representing $?\alpha << 1$ will be placed in the same e-class as the e-node representing $?\alpha \times 2$. The multiplication-to-shifting rule is a standard optimization in modern compilers. However, it can disable the rule $?\alpha \times ?\beta/?\beta \Rightarrow ?\alpha$ in traditional term rewriting systems (e.g. apply multiplication-to-shift on $x \times 2/2$). On the other hand, EqSat does not face this problem because it remembers both the original and the new term. Figure 2 shows an example of running EqSat on the expression $\alpha \times 2/2$ with the above rules and two additional properties of multiplication and division: $?\alpha \times ?\beta/?\eta \Rightarrow ?\alpha \times (?\beta/?\eta)$ and $?\alpha/?\alpha \Rightarrow 1$.

2.2 the egg framework

egg [Willsey et al. 2021] is an EqSat framework that supports defining domain-specific languages (DSLs) and rewrite rules of the DSLs. In addition to standard EqSat utilities, egg extends the e-classes with e-class analysis, a customizable auxiliary data to express static analysis information on terms represented by the e-class. E-class analysis is similar to abstract interpretation [Cousot and Cousot 1977] on e-graphs and enables conditional rewrite rules. For example, § 2.1.2 mentioned the rewrite rule $?\alpha/?\alpha \Rightarrow 1$, which is unsound when $?\alpha$ is matched to 0. Using egg, we can additionally attach e-class analyses that represent whether the terms in the e-class are semantically equivalent to 0 and check the analysis data before triggering the rewrite rule. The augmented conditional rewrite rule can be defined as $?\alpha/?\alpha \Rightarrow 1$ *if* $?\alpha \neq 0$.

3 CATSTAIL SYNTHESIZER

CatsTail synthesis pipeline has 3 stages. First, CatsTail takes in a P4 program as input and performs some frontend transformations to convert it into the representation we designed for EqSat, `Mio` IR. Then CatsTail creates the initial e-graph that contains the input program in `Mio` IR and invokes interfaces provided by egg [Willsey et al. 2021] to run EqSat. We designed 3 sets of rewrite rules to maximize the opportunity mapping computations to the target switch architecture (§ 3.3):

RR 1 *General-purpose transformations* encapsulate properties of arithmetic operators, algebraic simplifications, predicate logic formula identities, etc.

RR 2 *Table transformations* encode table splitting (when necessary), parallelization, and merging.

Program	$P := B^*$
Block	$B := S \mid \mathbf{Seq}(S, S)$
Stmt	$S := \mathbf{Assign}(V, E) \mid \mathbf{If}(E, B, B)$
Expr	$E := E + E \mid E - E \mid -E \mid \ldots$
Literals	$V := x_1, x_2, \ldots \mid 1, 2, 3, \ldots \mid \mathbf{true} \mid \mathbf{false}$

Fig. 4. Syntax of table actions in P4 programs

RR 3 *Target-dependent synthesis rules* match against target-supported computations derived from arithmetic logic unit (ALU) grammars (§ 3.3) and instantiate terms of invocations to corresponding ALU operators.

After running EqSat (terminates or timeout), CatsTail uses a *min-depth* cost model $C_{\mathcal{A}}$ to select a candidate program for target architecture $\mathcal{A}$. The cost of a term t given by $C_{\mathcal{A}}$ is the tree depth of t if t is a mappable computation (Figure 7a) to $\mathcal{A}$; otherwise, C will assign an ∞ cost to t. CatsTail employs the greedy extractor provided by egg [Willsey et al. 2021] to minimize the cost of the extracted candidate program. If the extracted program $\mathcal{T}$ satisfies that $C_{\mathcal{A}}(\mathcal{T}) < \infty$, then $\mathcal{T}$ is mappable to $\mathcal{A}$, and $C_{\mathcal{A}}(\mathcal{T})$ gives the number of stages $\mathcal{T}$ will take on $\mathcal{A}$. If $C_{\mathcal{A}}(\mathcal{T}) = \infty$, then there exists some computation in $\mathcal{T}$ that is mappable to $\mathcal{A}$. The latter case is attributed to the *incompleteness* of **RR 1** and **RR 2** (§ 4.2), and in § 4, we prove that **RR 3** is *complete*.

3.1 Frontend Transformations

P4 programs are written in an imperative language, whereas EqSat in egg [Willsey et al. 2021] requires a dataflow-based representation. Therefore, in the frontend transformation stage, CatsTail converts the P4 program, specifically actions of tables defined in the P4 program, into dataflow-based representation.

Algorithm 1 Example program

```
procedure FUNC(f1, f2)
    x ← f1 + f2
    y ← f1 − f2
    if y ≥ 0 then
        x ← x/2
    else
        x ← x × 2
    end if
end procedure
```

$$\Gamma^\star = \{x \mapsto ite(f_1 - f_2 \geq 0, \frac{f_1 + f_2}{2}, (f_1 + f_2) \times 2);\ y \mapsto f_1 - f_2\}$$
$$\Sigma^\star = \{x \mapsto \top;\ y \mapsto \top\}$$

After running SymEx on FUNC shown on the left side, we obtain $\Gamma^\star$ and $\Sigma^\star$ above. Then, the program in `Mio` IR can be constructed from $\Gamma^\star$. Note that $\Sigma^\star$ at the end of SymEx will always be $\top$.

Fig. 3. An example of the result of SymEx on FUNC (on the left). The resulting $\Gamma^\star$ and $\Sigma^\star$ are shown on the right.

We implement the ITE transformation using *symbolic execution* (SymEx) over table actions. Each table action is a loop-free code block with branching, which can be captured by the syntax in Figure 4. SymEx recursively evaluates the program with symbolic inputs. SymEx takes 4 arguments as inputs: $\mathcal{P}$, the input table action described in the syntax shown in Figure 4, Γ a variable-to-value

$$\textsc{eval-seq}\quad \frac{\Gamma_1, \Sigma_1 \xrightarrow{s_1}_\kappa \Gamma_2, \Sigma_2 \qquad \Gamma_2, \Sigma_2 \xrightarrow{s_2}_\kappa \Gamma_3, \Sigma_3}{\Gamma_1, \Sigma_1 \xrightarrow{\mathbf{Seq}(s_1,s_2)}_\kappa \Gamma_3, \Sigma_3}$$

$$\textsc{eval-assign}\quad \frac{}{\Gamma, \Sigma \xrightarrow{\mathbf{Assign}(v,e)}_\kappa [v \mapsto \textsc{subst}(e, \Gamma, \Sigma)]\Gamma, [v \mapsto \kappa]\Sigma}$$

$$\textsc{eval-if}\quad \frac{\begin{array}{c}\Gamma, \Sigma \xrightarrow{s_1}_c \Gamma_{s_1}, \Sigma_{s_1} \qquad \Gamma, \Sigma \xrightarrow{s_2}_{\neg c} \Gamma_{s_2}, \Sigma_{s_2} \\ \mathcal{I} = dom(\Gamma_{s_1}) \cap dom(\Gamma_{s_2}) \qquad \mathcal{J} = dom(\Gamma_{s_1}) \setminus \mathcal{I} \qquad \mathcal{K} = dom(\Gamma_{s_2}) \setminus \mathcal{I} \\ \mathcal{L} = \{v \mid v \in \mathcal{I} \wedge \Sigma_{s_1}(v) \neq \Sigma_{s_2}(v)\} \qquad \Gamma_1^\star = \{v \mapsto \textsc{gen-ite}(v, c, \Gamma_{s_1}, \Gamma_{s_2}) \mid v \in \mathcal{L}\} \\ \Gamma_2^\star = \{v \mapsto \textsc{gen-ite}(v, c, \Gamma_{s_1}, \Gamma) \mid v \in \mathcal{J}\} \qquad \Gamma_3^\star = \{v \mapsto \textsc{gen-ite}(v, \neg c, \Gamma_{s_2}, \Gamma) \mid v \in \mathcal{K}\} \\ \Sigma_1^\star = \{v \mapsto \kappa \mid v \in \mathcal{L} \cup \mathcal{J} \cup \mathcal{K}\} \qquad \Sigma_2^\star = \{v \mapsto \Sigma(v) \mid v \in \mathcal{I} \setminus \mathcal{L}\}\end{array}}{\Gamma, \Sigma \xrightarrow{\mathbf{if}(c,s_1,s_2)}_\kappa \Gamma_1^\star \cup \Gamma_2^\star \cup \Gamma_3^\star \cup \{v \mapsto \Gamma(v) \mid v \in \mathcal{I} \setminus \mathcal{L}\}, \Sigma_1^\star \cup \Sigma_2^\star}$$

Fig. 5. Semantics of the SymEx over P4 table actions where $\textsc{gen-ite}(v, c, \Gamma_1, \Gamma_2) \stackrel{\text{def}}{=} v \mapsto ite(c, \Gamma_1(v), \Gamma_2(v))$ and $\textsc{subst}(e, \Gamma, \Sigma) \stackrel{\text{def}}{=} \textsc{reduce}(\lambda ev.[\textsc{gen-ite}(v, \Sigma(v), \Gamma, \textsc{id})/v]e, e, dom(\Gamma))$

mapping, Σ a variable-to-path-condition mapping and the current path condition κ. Figure 3 shows an example of running SymEx over a simple imperative program with conditional.

Γ keeps track of the most up-to-date value of each variable defined in $\mathcal{P}$ and Σ maintains a *guard* for each variable, denoting the condition of assigning a variable v to the value $\Gamma(v)$. The outputs of SymEx are updated mappings $\Gamma^\star$ and $\Sigma^\star$ after executing $\mathcal{P}$ under the path condition κ.

When evaluating an $\mathbf{If}(c, s_1, s_2)$ statement, SymEx evaluates both s_1 and s_2 with path conditions c and $\neg c$ respectively. Then, SymEx constructs the *merged* mappings using the outputs from executing s_1 and s_2. Suppose the output from executing s_1 and s_2 are $\Gamma_{s_1}, \Sigma_{s_1}$ and $\Gamma_{s_1}, \Sigma_{s_2}$ respectively. The merging is based on whether a variable v is changed in both branches. If both s_1 and s_2 assign some value to v, then v appears in $dom(\Gamma_{s_1}) \cap dom(\Gamma_{s_2})$ but $\Sigma_{s_1}(v) \neq \Sigma_{s_2}(v)$. In this case, a conditional assignment $ite(c, \Gamma_{s_1}(v), \Gamma_{s_2}(v))$ is constructed for v. If v is modified in one of s_1 and s_2, then the conditional assignment will only consider the path condition for the branch that alters the value of v. By the formal semantics of the symbolic executor shown in Figure 5, the evaluation of SymEx over an input program $\mathcal{P}$ is $\emptyset, \emptyset \xrightarrow{\mathcal{P}}_\top \Gamma^\star, \Sigma^\star$. The full procedure is shown in the Appendix (§ 7) Algorithm 3.

3.2 Generic Table Representation: Mio IR

The frontend transformation converts control flows from the input P4 programs into purely data flow-based representation, but it only works on the level of table actions. The remaining "controls" are from the order of table applications. To separate table controls from computations performed by each table, we designed a representation called `Mio` IR to encapsulate P4 programs after ITE transformation. `Mio` IR uses a *sea-of-nodes* [Click and Paleczny 1995] representation. An example of a `Mio` IR program is shown in Figure 6. The program represents a table that matches k_1, k_2, k_3 and have 2 actions A_1 and A_2, where A_1 assigns v_1 to $1 + 2 + f_1$ and A_2 sets v_2 to 2. In the figure, green nodes are *compute* nodes and orange nodes are *table control* nodes. Nodes with E operators are *elaborators*. The elaborator nodes set boundaries between compute nodes and control nodes, which makes the representation suitable for EqSat. The specific benefits of posing elaborators between table control nodes and compute nodes will be discussed later in § 3.3. Other table control nodes are: table sequencing $Sequence(\mathcal{T}_1, \mathcal{T}_2)$ and table parallelization $Join(\mathcal{T}_1, \mathcal{T}_2)$. The former stands for placing $\mathcal{T}_1$ before $\mathcal{T}_2$, which enforces a strict topological order of table application. The latter indicates two tables can be arranged into the same logical stage.

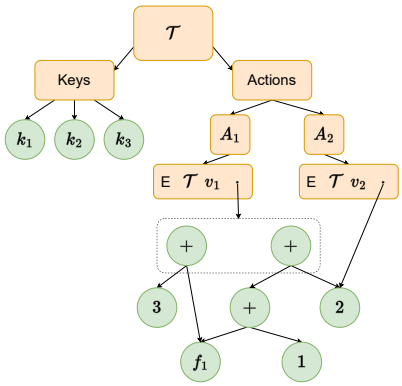


Fig. 6. A table represented in Mio IR that has 3 match keys (k_1, k_2, k_3) and 2 actions A_1 and A_2.

Mio IR is defined using interfaces provided by egg [Willsey et al. 2021] and thus can be directly converted to an e-graph representation for subsequent EqSat procedure.

3.3 Equality Saturation on Mio IR

Given an initial e-graph $\mathcal{G}$, we design 3 sets of rewrite rules for synthesis.

General-purpose Transformations. To explore the space of equivalent programs, we encode a set of rules that captures basic arithmetic properties, such as commutativity and associativity of addition, predicate logic formula identities such as De Morgan law, and properties of conditional assignments.

Example 3.1. The following are simple examples of rewrite rules that perform algebraic simplification, De Morgan's law, and if-then-else identity. $\leftrightsquigarrow$ stands for *bi-directional* rewrite.

$$\textsc{alg-simp}\ \frac{\Gamma \vdash \alpha : \textsc{int}}{\alpha + 0 \rightsquigarrow \alpha} \qquad \textsc{de-morgan}\ \frac{\Gamma \vdash \alpha : \textsc{bool} \qquad \Gamma \vdash \beta : \textsc{bool}}{\neg(\alpha \wedge \beta) \leftrightsquigarrow (\neg\alpha) \vee (\neg\beta)} \qquad \textsc{ite-ident}\ \frac{\Gamma \vdash c : \textsc{bool}}{ite(c, e_1, e_2) \leftrightsquigarrow ite(\neg c, e_2, e_1)}$$

Elaborators are boundaries between compute and table control nodes and take care of the *effects* of each table action. Therefore, rules that work with pure computations do not need to maintain the effects brought by table controls. For instance, in Figure 6, the effect of A_1 is assigning a value to v_1. Since the elaborator separates the effect from the value assigned to v_1, rewrite rules that match on pure computations need not know which variable the matched expression is assigned to. Any term represented by the child e-class of the elaborator are valid candidate values assigned to v_1.

These rules are also composable with each other to perform some term rewriting-based compiler optimizations such as constant folding and the example illustrated earlier in § 2.1.2. The full set of General-purpose rewrite rules is in § 8.1.

Table Transformations. The program after ITE transformation applies tables in a topological order that respects the table dependency of the original program. However, the order might not be the most preferred. For example, suppose a table $\mathcal{T}_1$ is placed before another table $\mathcal{T}_2$ because SymEx sees $\mathcal{T}_1$ before seeing $\mathcal{T}_2$. However, there is no read and write dependency between $\mathcal{T}_1$ and $\mathcal{T}_2$. In this case, if the memory limit permits, $\mathcal{T}_1$ and $\mathcal{T}_2$ can be placed in the same stage. To have a diverse set of candidate programs, we design a set of rewrite rules that (a) alter the table application order;

(b) decompose computations in table actions and span them across stages. The purpose of (b) is to transform computations in candidate programs into preferred shapes that can potentially match the target ALU grammar. For instance, suppose on some switch architecture, the ALU that performs conditional assignment can only compute a single binary predicate ($\alpha \wedge \beta$ or $\alpha \vee \beta$). Suppose there is a computation in some action that requires computing $ite(\alpha \wedge \beta \wedge \eta, e_1, e_2)$. The predicate computes $\wedge$ with 3 inputs, so no matter how the expression is transformed, the computation can never match with the ALU grammar. In this case, the table transformation rewrite rule will "lift" the computation of $\alpha \wedge \beta$ to some previous stage, and make a new packet header vector (PHV) field $f_{\alpha\wedge\beta}$ and instantiates a new conditional assignment $ite(f_{\alpha\wedge\beta} \wedge \eta, e_1, e_2)$ in some later stage. After this transformation, the computation matches the ALU grammar and can be captured by the target-dependent synthesis rule (discussed below) to instantiate invocations to ALU operators.

Target-dependent Synthesis Rules. Supported computations are different across targets. Therefore, for each backend target, we design a set of *synthesis rules* that are derived from the grammars of supported computations of the target ALU. The LHS pattern of these synthesis rules are shapes of computations that are supported by the target switch ALU. As an axiom, PHV fields (variables) and constants are always considered supported. Computations that take any output from other computations can be matched with LHS if all the arguments of the computation have already been mapped to some ALU operator invocations. For instance, suppose some ALU supports binary addition. Then, the corresponding rewrite rule is

$$\textsc{add}(?x, ?y) \Rightarrow \textsc{alu-add}(?x, ?y) \ \textit{if}\ (\textsc{is-mapped}(?x) \wedge \textsc{is-mapped}(?y))$$

The full set of these rewrite rules can be found in Appendix 8.2. The formal definition of "mappable computations", "mapped computations" will be discussed in detail in § 4.

3.4 Extraction with Target-aware Cost Models

EqSat populates the e-graph with program representations equivalent to the original input program modulo **RR 1**~**RR 3** discussed in the earlier sections. We employ a cost-based architecture-aware extraction to select one candidate program from the (sub)-space EqSat explores. There are 2 objectives of extraction. First, because the rewrite rules do not guarantee the existence of a candidate program that can be compiled to the target switch, any computations that have not been converted to ALU operator invocations have to be filtered. Second, the number of stages required by the extracted candidate is minimized. The two objectives are encapsulated by a cost function $C_{\mathcal{A}}(\mathcal{T})$

Algorithm 2 Target-aware cost function

```
procedure C_A(T)
    if Var(v) = T ∨ Const(c) = T then
        return 0
    else
        f(e⃗) ← T
        if ¬MAPPED(A, f) then
            return ∞                          ▷ unsupported computation
        end if
        c_e ← REDUCE-MAX(λe_i.C_A(e_i), e⃗)
        return c_e + 1
    end if
end procedure
```

(Algorithm 2) that takes in a term $\mathcal{T}$ in `Mio` IR as input. If $\mathcal{T}$ is a PHV field or a constant, we assign a cost of 0. On the other hand, it must be some computation in the form of a function call $f(\vec{e})$. In this case, $C_{\mathcal{A}}$ first checks if $\mathcal{T}$ has already been converted to some ALU operator invocations by the target-dependent synthesis rules. If it is not in the form of an ALU operator, then an ∞ cost is assigned. Otherwise, $C_{\mathcal{A}}$ computes the tree depth of $\mathcal{T}$.

Given the e-graph $\mathcal{G}$ after EqSat, the objective function of the extraction problem is

$$\arg\min_{\mathcal{T} \in \mathcal{G}} C_{\mathcal{A}}(\mathcal{T})$$

In other words, we try to minimize the depth of the $\mathcal{T}$, which is equivalent to the number of stages required for mapping $\mathcal{T}$ to the target $\mathcal{A}$. Solving the optimal $\mathcal{T}$ is hard because $C_{\mathcal{A}}(\cdot)$ is not continuous and $|\mathcal{G}|$, the number of terms represented by $\mathcal{G}$, can be doubly-exponentially many, which makes the extraction problem NP-Hard. Therefore, we use the greedy extractor interface provided by egg to approximate the optimal $\mathcal{T}$. The result in our evaluation (§ 5) shows that the greedy extractor gives good approximations.

4 COMPLETENESS OF REWRITE RULES

An important property of the synthesis rewrite rules is completeness: *if a computation is supported by the ALU on the target platform, then the rewrite rules will instantiate the corresponding ALU operators that do the same computation.* We prove that the target-dependent synthesis rules we defined are *complete*.

4.1 Completeness of Target-dependent synthesis rules

First, we define (formally) ALU operation mappings, the notion of *mappable computations* and *mapped computations*.

Definition 4.1. (ALU Operation Mapping) Define the ALU operation mapping to be $\Delta(\cdot)$, where $dom(\Delta)$ is a set of atomic computations supported by the ALU, which are arithmetic corresponding to ALU-Ops in Figure 7b and $Im(\Delta)$ are the corresponding ALU operators. Δ can be computed from an input ALU grammar. In this proof, we consider an abstract ALU grammar defined in Figure 7b.

Definition 4.2. (Mapped Computations) A computation e is mapped to ALU operations provided by an ALU operation mapping Δ, denoted by $\Delta \models e$, is defined as

$$\textsc{Mapped-Var}\ \frac{Var(v)}{\Delta \models v} \qquad \textsc{Mapped-Const}\ \frac{Const(c)}{\Delta \models c} \qquad \textsc{Mapped-Ind}\ \frac{\begin{array}{c} f \in Im(\Delta) \\ \forall i \in [n].\ \Delta \models e_i \end{array}}{\Delta \models f(e_i)}$$

Mapped computations are inductively defined. PHV fields and constants are always considered mapped. For any computations f that takes in other values as arguments, the computation f is mapped if f is in the image of the target ALU operation mapping (f is an ALU operator) and all the arguments it reads from outputs of some mapped computations.

Running equality saturation with general-purpose transformations and table transformations explores the space of computations equivalent to the initial input program. However, the computation capability of ALUs on the target platform is limited and only arithmetic operations in certain forms can be computed. In CaT [Gao et al. 2023], the supported computations are captured using ALU grammars as defined in Figure 7b. Given the ALU grammar, we define a set of rewrite rules that satisfy the properties below.

$$\textsc{Map-Var}\ \frac{Var(v)}{\Delta \vdash v} \qquad \textsc{Map-Const}\ \frac{Const(c)}{\Delta \vdash c}$$

$$\textsc{Map-Comp}\ \frac{f \in dom(\Delta) \qquad \forall i \in [n].\ \Delta \vdash e_i}{\Delta \vdash f(e_1, \cdots, e_n)}$$

(a) Inference rule for mappable computations

ALU	$E ::= F(E^*) \mid L$
ALU-Ops	$F ::= f_1, f_2, f_2, \ldots$
Var-Const	$L ::= 1, 2, 3, \ldots \mid x_1, x_2, \ldots$

(b) Stateless ALU Grammar

Fig. 7. Mappable computations and ALU grammar

$$\text{RR-Progress-E}\ \frac{\exists i.\ e_i \rightsquigarrow_r e_i'}{f(\vec{e}) \rightsquigarrow_r f(e_1 \ldots, e_i', \ldots e_n)}$$

$$\text{RR-Progress-F}\ \frac{f \in dom(\Delta) \qquad \forall i \in [n].\ \Delta \models e_i \qquad f^\star = \Delta(f)}{f(e_1, \ldots, e_n) \rightsquigarrow_r f^\star(e_1, \ldots, e_n)}$$

RR-Progress-E and RR-Progress-F constrain the rewrite rules such that the call to the corresponding ALU operator $f^\star$ will be instantiated if all the arguments to f have been mapped under Δ by the rewrite rule.

Definition 4.3. (Mappable computations) Given an ALU operation mapping Δ and some computation e, let $\Delta \vdash e$ be the judgment that e is mappable given Δ. The derivation of the judgment is formally defined in Figure 7a.

Similar to Definition 4.2, mappable computations are also inductively defined. A computation f is mappable if f is mapped to some ALU operator by Δ and all of its arguments are mappable. To be clearer about the terminologies used in the proof, the following are equivalences between judgments and statements.

(1) $\Delta \vdash \mathcal{K} \iff \mathcal{K}$ only contains constants, PHV fields and arithmetic operators in $dom(\Delta)$.
(2) $\Delta \models \mathcal{K} \iff \mathcal{K}$ only contains constants, PHV fields and ALU operators in $Im(\Delta)$.

Definition 4.4. Let $\rightsquigarrow_r^*$ be the reflexive transitive closure of $\rightsquigarrow_r$ defined as

$$\rightsquigarrow_r^*\text{-REFL}\ \frac{}{e \rightsquigarrow_r^* e} \qquad \rightsquigarrow_r^*\text{-TR}\ \frac{e_1 \rightsquigarrow_r^* e_2 \qquad e_2 \rightsquigarrow_r e_3}{e_1 \rightsquigarrow_r^* e_3}$$

□

Lemma 4.5. *For all terms e_i, e_i' where $i \in [n]$, if $e_i \rightsquigarrow_r^* e_i'$, then*

$$f(\vec{e}) \rightsquigarrow_r^* f(e_1, \ldots, e_i', \ldots, e_n)$$

Proof. By structural induction on the derivation of $e_i \rightsquigarrow_r^* e_i'$

- $\rightsquigarrow_r^*$REFL. In this case $e_i = e_i'$ and by $\rightsquigarrow_r^*$REFL we have $f(\vec{e}) \rightsquigarrow_r^* f(\vec{e})$
- $\rightsquigarrow_r^*$TR. In this case, there is an $\widehat{e}$ such that $e_i \rightsquigarrow_r^* \widehat{e}$ and $\widehat{e} \rightsquigarrow_r e'$. By the induction hypothesis, we have $f(e_1 \ldots, e_i, \ldots, e_n) \rightsquigarrow_r^* f(e_1, \ldots, \widehat{e}, \ldots, e_n)$. Let $\vec{w} = \langle e_1, \ldots, \widehat{e}, \ldots, e_n \rangle$. We have an i such that $w_i \rightsquigarrow_r e'$ where $w_i = \widehat{e}$. Then, by RR-Progress-E we have $f(\vec{w}) \rightsquigarrow_r f(e_1, \ldots, e', \ldots, e_n)$. Finally, by $\rightsquigarrow_r^*$-TR, we have $f(e_1 \ldots, e_i, \ldots, e_n) \rightsquigarrow_r^* f(e_1, \ldots, e', \ldots, e_n)$

□

Finally, we prove the completeness theorem for rules that satisfy RR-Progress-E and RR-Progress-F.

Theorem 4.6. *(Completeness of Synthesis Rewrite) Let $\rightsquigarrow_r^*$ be the reflexive transitive closure of $\rightsquigarrow_r$. Given an ALU grammar with the associated ALU operation mapping Δ and some computation e, if $\Delta \vdash e$, then $\exists e^\star.\ e \rightsquigarrow_r^* e^\star \wedge \Delta \models e^\star$*

Proof. By structural induction on the premise $\Delta \vdash e$

- map-var. In this case $e = v$ for some variable v. Let $e^\star$ be v. By reflexivity of $\rightsquigarrow_r^*$, we have $v \rightsquigarrow_r^* v$. Then by mapped-var, $\Delta \models e^\star$
- map-const. Similar to the previous case.
- map-comp. In this case $e = f(e_1, \ldots, e_n)$.We have $f \in dom(\Delta)$ and $\forall i \in [n].\ \Delta \vdash e_i$. By induction hypothesis, we have that for any $i \in [n]$, there exists some $e^\star$ such that $e_i \rightsquigarrow_r^* e_i^\star$ and $\Delta \models e_i^\star$. Since Δ is well-defined at f, let $f^\star = \Delta(f)$. Then, by case analysis on $e_i \rightsquigarrow_r^* e_i^\star$
  (1) Suppose all such cases are by $\rightsquigarrow_r^*$-refl. Subsequently, we have $\forall i.\ e_i = e_i^\star$ and $f(e_1, \ldots, e_n) \rightsquigarrow_r^* f(e_1^\star, \ldots, e_n^\star)$ by $\rightsquigarrow_r^*$-refl. By RR-Progress-F, we have $f(e_1^\star, \ldots, e_n^\star) \rightsquigarrow_r f^\star(e_1^\star, \ldots, e_n^\star)$. By $\rightsquigarrow_r^*$-tr, $f(e_1, \ldots e_n) \rightsquigarrow_n^* f^\star(e_1^\star, \ldots, e_n^\star)$. Since $f^\star \in Im(\Delta)$ and $\forall i \in [n].\ \Delta \models e_i^\star$, by Mapped-Ind we have $\Delta \models f^\star(e_1^\star, \ldots, e_n^\star)$.
  (2) Suppose there are some $i \in \mathcal{S} \subseteq [n]$ such that $e_i \rightsquigarrow_r^* e_i'$ and $e_i' \rightsquigarrow_r e_i^\star$, and $\forall j \notin \mathcal{S}, e_j \rightsquigarrow_r^* e_j^\star \wedge e_j = e_j^\star$. WLOG, suppose $\mathcal{S}$ is a singleton set, otherwise, we can apply this proof multiple times.
  By Lemma 4.5, we have $f(e_1, \ldots, e_i, \ldots, e_n) \rightsquigarrow_r^* f(e_1^\star, \ldots, e_i', \ldots, e_n^\star)$. Then, by RR-Progress-E and $e_i' \rightsquigarrow_r e_i^\star$, we have $f(e_1^\star, \ldots, e_i', \ldots, e_n^\star) \rightsquigarrow_r f(e_1^\star, \ldots, e_i^\star, \ldots, e_n^\star)$. By $\rightsquigarrow_r^*$-tr, we have $f(e_1, \ldots, e_i, \ldots, e_n) \rightsquigarrow_r^* f(e_1^\star, \ldots, e_i^\star, \ldots, e_n^\star)$. Since $f \in dom(\Delta)$, by RR-Progress-F, we have $f(e_1^\star, \ldots, e_i^\star, \ldots, e_n^\star) \rightsquigarrow_r f^\star(e_1^\star, \ldots, e_i^\star, \ldots, e_n^\star)$. By $\rightsquigarrow_r^*$-tr, $f(e_1, \ldots, e_i, \ldots, e_n) \rightsquigarrow_r^* f^\star(e_1^\star, \ldots, e_i^\star, \ldots, e_n^\star)$. Finally, since $f^\star \in Im(\Delta)$ and $\forall i \in [n].\ \Delta \models e_i^\star$, we conclude that $\Delta \models f^\star(e_1^\star, \ldots, e_i^\star, \ldots, e_n^\star)$

□

4.2 Discussion

Soundness of rewrite rules. The rewrite rules provided to EqSat are axiomatized. This means if there is an unsound rewrite (e.g. rewriting 1 to 0) then the entire procedure of EqSat can be unsound. We have not formalized the rewrite rules we provided to EqSat, but it could be done by checking the semantics equivalence using interactive proof assistants such as Coq [Bertot and Castéran 2013] and Lean [Moura and Ullrich 2021]. The soundness can also be verified by automated theorem proving using Z3 [De Moura and Bjørner 2008] or CVC5 [Barbosa et al. 2022].

(In)Completeness of General-purpose transformations and Table transformations. We have shown that the target-dependent synthesis rules are *complete* (Theorem 4.6). But can we also show that the other two sets of rewrite rules are also complete? In the context of general program transformations, the definition of completeness is that the given set of rewrite rules are able to explore *all* possible equivalent representations of the initial input program. The key to having a complete set is to encode fundamental properties of the arithmetic operators (e.g. commutativity and associativity of addition and multiplication), and general-purpose transformations are encoded using all the available properties of computations that can be expressed in `Mio` IR. Checking the completeness of these transformations is considered future work.

5 IMPLEMENTATION AND EVALUATION

CatsTail is implemented with about 4900 lines of Rust code. The core dependency is egg [Willsey et al. 2021]. The codebase is publicly available on GitHub[1]. The core language and frontend transformations (§ 3.2) are implemented in language.rs. The rewrite rules (§ 3.3) are implemented under rewrites directory. The greedy extractor (§ 3.4) is implemented in extractor.rs.

5.1 Evaluation and Experiments Setup

We address the following research questions (RQs):

RQ1: Resource synthesis speed. Can CatsTail be faster on synthesizing computations to target switches compared to CaT [Gao et al. 2023]?

RQ2: Quality of extraction approximation. How good is CatsTail on approximating the optimal computation on a target switch using the greedy extractor?

RQ3: Rewrite rules. How to improve the rewrite rules if the extractor cannot find a mappable computation?

We evaluated CatsTail with benchmarks provided by the artifact of CaT [Gao et al. 2023] perform resource synthesis for Intel Tofino ALUs ($\mathcal{A}_i$) and Domino (Banzai) ALUs ($\mathcal{A}_b$). The set of benchmarks include The benchmarks include Blue Increase [chang Feng et al. 2002], Flowlet Switching [Sinha et al. 2004], Sampling [Sivaraman et al. 2016], Marple flow [Narayana et al. 2017] and RCP [Tai et al. 2008].

To answer *RQ1*, for each benchmark instance, we run CatsTail with $\mathcal{A} = \mathcal{A}_i$ and $\mathcal{A} = \mathcal{A}_b$ respectively, and measure the end-to-end time.

To answer *RQ2*, after CatsTail yields an output $\mathcal{T}$, we compute its cost $C_{\mathcal{A}}(\mathcal{T})$. If $\mathcal{T}$ has a non-infinity cost, then we take the cost as the number of stages required to map $\mathcal{T}$ on $\mathcal{A}$, otherwise, we conclude CatsTail fails to synthesize for $\mathcal{A}$ using the original input program.

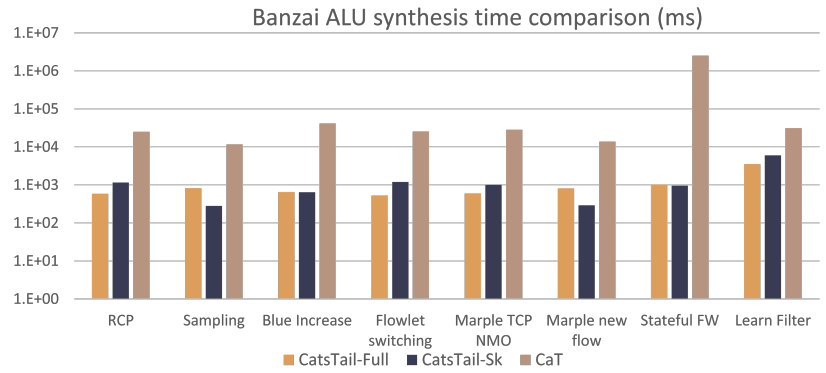
To answer *RQ3*, we developed the 3 sets of rewrite rules from scratch. We elaborate on our experience when developing the rewrite rules for CatsTail later in this section.

5.2 Speed of resource synthesis

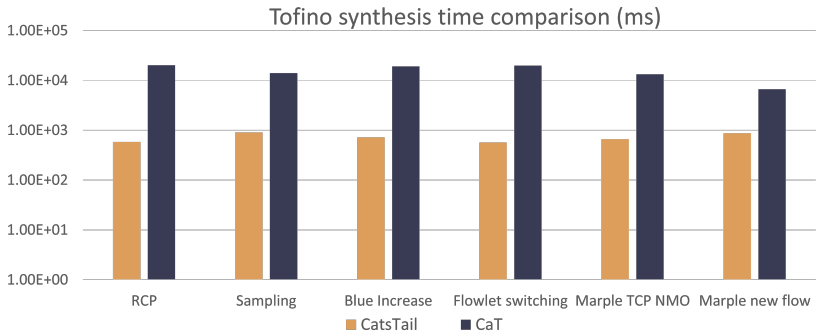
We evaluate CatsTail on 8 benchmarks from CaT [Gao et al. 2023] and compare the resource synthesis time with CaT. [Gao et al. 2023] The target switches are Intel Tofino ALUs and Domino Banzai ALUs. For both targets, we use the same set of general-purpose program transformation rules and table transformation rules. We have 1 set of target-dependent synthesis rules for Tofino switches bit 2 sets for Domino switches. For Domino switches, this is because computations supported by Domino ALUs are defined in several different grammars. For each grammar, we have a corresponding synthesis rule. A set of target-dependent synthesis rules is `Full`, which means we aggregate all the synthesis rules of Domino switches (§ 8.2) into a monolithic set of rewrite rules. The other set `Sk` contains synthesis rules tuned for each benchmark to match the ALU grammar used by SKETCH [Solar-Lezama 2008] in CaT [Gao et al. 2023] for fair comparisons. For example, the Blue Increase benchmark in CaT [Gao et al. 2023] uses the grammar PRED-RAW, then `Sk` used in the corresponding benchmark in CatsTail also only contains the synthesis rule derived from PRED-RAW.

We set a 5-second timeout for EqSat and measured the time to synthesize the 8 benchmarks. In all the experiments, CatsTail is at least an order of magnitude faster than CaT [Gao et al. 2023] on resource synthesis. For harder cases such as stateful firewall, the synthesis speed is improved by

[1]https://github.com/AD1024/CatsTail



(a) Synthesis time comparison for Domino (Banzai) ALUs. CatsTail-Full uses all the synthesis rules and CatsTail-Sk only uses rules derived from ALU grammars that were used by CaT [Gao et al. 2023] for the specific benchmarks.



(b) Synthesis time comparison for Intel Tofino ALUs

Fig. 8. Synthesis time comparisons

about 3 orders of magnitude. These preliminary results show good evidence to use CatsTail for resource synthesis given the fast synthesis speed.

5.3 Number of stages required after resource synthesis

To evaluate whether the greedy extraction gives good approximations to the optimal, we compare the number of stages the synthesized program from CatsTail and CaT [Gao et al. 2023] takes. The number of required stages for an extracted candidate is computed using the cost function discussed in § 3.4. The results are shown in Table 1. When taking Intel Tofino as the target switch, CatsTail is able to synthesize programs with stage usage as good as the synthesized program from CaT [Gao et al. 2023]. For Domino switches, CatsTail can find better candidates thanks to the composition of the general-purpose and table transformation rules. Therefore, even though CatsTail employs a greedy heuristic to extract the candidate program, the synthesized results are good approximations to optimal solutions. In subsequent work, we will investigate which set of rewrite rules enabled CatsTail to discover a program that has a smaller stage usage than CaT [Gao et al. 2023].

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	3	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	-	-

5.4 Experience of developing the rewrite rules

We experienced synthesis failure during the development of the general-purpose and table transformation rules. The reason that CatsTail could not synthesize a mappable computation for a target switch is because of the *incompleteness* of these rewrite rules. Notice that computations that are mapped to the target switch are only instantiated by the target-dependent synthesis rules, which are derived from some ALU grammars. Moreover, the triggering of a rewrite rule is based on syntactic matches on the left-hand-side pattern. Therefore, the synthesis rules are not triggered until there is some term t represented in the e-graph that matches exactly the structure of mappable computations generated by the ALU grammar. If a computation t' is semantically equivalent to t but in a different structure, the rewrite rule will not be triggered. Thus, the goal of designing the general-purpose and table transformation rules is to transform t' into t by some combinations of the rules. It is hard to show and guarantee that these sets of rewrite rules are complete, meaning that they can discover all the candidates that are semantically equivalent to the input program. However, we have a methodology to improve the completeness when we encounter synthesis failures.

We first augment the extractor to report the unsupported computations in the extracted candidate program. When CatsTail fails to synthesize a program, we check the report given by the extractor and inspect whether the structure of the term is equivalent to some computations that can be recognized by the ALU grammar. If so, we specify several rules to alter the sub-structures of the unsupported computation and turn them into the syntactic form that can be recognized by the ALU grammar. In future work, we would like to explore how the completeness of the rewrite rules can be automatically improved when a synthesis failure happens.

6 CONCLUSION

In this report, we introduce CatsTail, an equality saturation-based resource synthesizer for packet programs. CatsTail utilizes both the semantics and the structure of input programs and leverages equality saturation to explore the space of equivalent program representations using 3 sets of rewrite rules: general-purpose program transformations, table transformations, and target-dependent synthesis rules. Moreover, we proved the completeness of the target-dependent synthesis rules, which guarantees that mappable computations will eventually be mapped to some computations on the target switches. The evaluation shows significant performance improvement over the SKETCH [Solar-Lezama 2008] synthesizer, which is a good sign of replacing SKETCH with CatsTail in the packet program compilation pipeline. The greedy heuristic employed by CatsTail to select a

candidate program for output gives good approximations: the stage usage of programs synthesized by CatsTail is also at least as good as results given by CaT [Gao et al. 2023]. In the future, we would like to explore how to improve the completeness of the rewrite rules and integrate CatsTail into a part of an end-to-end packet program compilation flow. We would also like to explore automatically computing the target-dependent synthesis rules from ALU grammars.

REFERENCES

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (jul 2014), 87–95. https://doi.org/10.1145/2656877.2656890

Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (aug 2013), 99–110. https://doi.org/10.1145/2534169.2486011

Wu chang Feng, K.G. Shin, D.D. Kandlur, and D. Saha. 2002. The BLUE active queue management algorithms. *IEEE/ACM Transactions on Networking* 10, 4 (2002), 513–528. https://doi.org/10.1109/TNET.2002.801399

Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. *SIGPLAN Not.* 30, 3 (mar 1995), 35–49. https://doi.org/10.1145/202530.202534

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 435–450. https://doi.org/10.1145/3387514.3405879

Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 72–88. https://doi.org/10.1145/3582016.3582036

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3098822.3098829

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (apr 1980), 356–364. https://doi.org/10.1145/322186.322198

Shan Sinha, Srikanth Kandula, and Dina Katabi. 2004. Harnessing TCP ' s Burstiness with Flowlet Switching. https://api.semanticscholar.org/CorpusID:9358977

Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.

C.-H. Tai, J. Zhu, and N. Dukkipati. 2008. Making Large Scale Deployment of RCP Practical for Real Networks. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*. 2180–2188. https://doi.org/10.1109/INFOCOM.2008.285

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: a New Approach to Optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (Savannah, GA, USA). ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (jan 2021), 1–29. https://doi.org/10.1145/3434304

7 APPENDIX - SYMEX ALGORITHM FOR ITE TRANSFORMATION

Input $\mathcal{P}, \kappa, \Gamma, \Sigma$ ▷ Program, path condition, context, and path condition mapping
Returns (Γ', Σ') ▷ Updated variable context and path conditions after running $\mathcal{P}$

procedure SymExec($\mathcal{P}$, κ, Γ, Σ)
  if $\mathcal{P} = Seq(s_1, s_2)$ **then**
    $(\Gamma_{s_1}, \Sigma_{s_1}) \leftarrow$ SymExec(s_1, κ, Γ, Σ)
    return SymExec(s_1, Γ_{s_1}, Σ_{s_1})
  else if $\mathcal{P} = Assign(v, e)$ **then**
    return UpdateCtx(Γ, Σ, v, e, κ)
  else if $\mathcal{P} = If(c, s_1, s_2)$ **then**
    $(\Gamma_{s_1}, \Sigma_{s_1}) \leftarrow$SymExec($s_1$, $\kappa \wedge c$, Γ, Σ)
    $(\Gamma_{s_2}, \Sigma_{s_2}) \leftarrow$SymExec($s_2$, $\kappa \wedge (\neg c)$, Γ, Σ)
    return merge(Γ, Σ, Γ_{s_1}, Σ_{s_1}, Γ_{s_2}, Σ_{s_2}, κ)
  end if
end procedure

procedure subst(Γ, Σ, e) ▷ Substitute variables in e with values in Γ
  for $v \in dom(\Gamma)$ **do**
    $\kappa \leftarrow \Sigma(v)$
    $e \leftarrow [ite(\kappa, \Gamma(v), v)/v]e$
  end for
  return e
end procedure

procedure UpdateCtx(Γ, Σ, v, e, κ)
  $e^\star \leftarrow$subst(Γ, Σ, e)
  if $\exists v \mapsto e' \in \Gamma$ **then**
    $\kappa_0 \leftarrow \Sigma(v)$
    return $(\Gamma \setminus \{v \mapsto e'\} \cup \{v \mapsto e^\star\}, \Sigma \setminus \{v \mapsto \kappa_0\} \cup \{v \mapsto \kappa\})$
  else
    return $(\Gamma \cup \{v \mapsto e^\star\}, \Sigma \cup \{v \mapsto \kappa\})$
  end if
end procedure

procedure merge(Γ, Σ, Γ_1, Σ_1, Γ_2, Σ_2, κ) ▷ Union of contexts of branches
  $\Gamma^\star \leftarrow \varnothing$
  $\Sigma^\star \leftarrow \varnothing$
  $\mathcal{I} \leftarrow dom(\Gamma_1) \cap dom(\Gamma_2)$
  for $v \in \mathcal{I}$ **do** ▷ Variables modified in both branch
    $\kappa_\top \leftarrow \Sigma_1(v)$
    $\kappa_\bot \leftarrow \Sigma_2(v)$
    if $\kappa_\top \neq \kappa_\bot$ **then**
      $\Gamma^\star \leftarrow \Gamma^\star \cup \{v \mapsto ite(\kappa_\top, \Gamma_1(v), \Gamma_2(v))\}$
      $\Sigma^\star \leftarrow \Sigma^\star \cup \{v \mapsto \kappa\}$
    else ▷ Variables remain unchanged in both branches
      $\Gamma^\star \leftarrow \Gamma^\star \cup \{v \mapsto \Gamma(v)\}$
      $\Sigma^\star \leftarrow \Sigma^\star \cup \{v \mapsto \Sigma(v)\}$

end if
end for
for $v \in (dom(\Gamma_1) \setminus \mathcal{I}) \cup (dom(\Gamma_2) \setminus \mathcal{I})$ **do**
if $v \in dom(\Gamma_1)$ **then**
$e^\star \leftarrow ite(\Sigma_1(v), \Gamma_1(v), \Gamma(v))$
else
$e^\star \leftarrow ite(\Sigma_2(v), \Gamma_2(v), \Gamma(v))$
end if
$\Gamma^\star \leftarrow \Gamma^\star \cup \{v \mapsto e^\star\}$
$\Sigma^\star \leftarrow \Sigma^\star \cup \{v \mapsto \kappa\}$
end for
return $(\Gamma^\star, \Sigma^\star)$
end procedure

8 APPENDIX - REWRITE RULES

8.1 General-purpose transformations

```
#[allow(dead_code)]
pub fn alg_simpl() -> Vec<RW> {
    vec![
        rewrite!("add-sub-convert"; "(+ ?x (neg ?y))" => "(- ?x ?y)"),
        rewrite!("sub-neg-convert"; "(- 0 ?x)" => "(neg ?x)"),
        rewrite!("add-sub-elim"; "(- (+ ?x ?y) ?y)" => "?x"),
        rewrite!("sub-conv"; "(- ?x (- ?y ?z))" => "(+ (- ?x ?y) ?z)"),
        rewrite!("add-conv"; "(+ ?x (- ?y ?z))" => "(- (+ ?x ?y) ?z)"),
        rewrite!("add-zero"; "(+ ?x 0)" => "?x"),
        rewrite!("sub-zero"; "(- ?x 0)" => "?x"),
        rewrite!("double-to-shift"; "(+ ?x ?x)" => "(bitshl ?x 1)"),
        rewrite!("shl-shr-elim"; "(bitshr (bitshl ?x ?y) ?y)" => "?x"),
        rewrite!("shl-sum"; "(bitshl ?x (+ ?y ?z))" => "(bitshl (bitshl ?x ?y)
            ↪ ?z)"),
        rewrite!("shr-to-zero"; "(bitshr ?x ?y)" => "0" if is_greater_eq("?y".
            ↪ parse().unwrap(), 32)),
        rewrite!("sub-elim"; "(- ?x ?x)" => "0"),
        rewrite!("add-comm"; "(+ ?x ?y)" => "(+ ?y ?x)"),
        rewrite!("add-assoc"; "(+ ?x (+ ?y ?z))" => "(+ (+ ?x ?y) ?z)"),
        rewrite!("add-identity"; "?x" => "(+ ?x 0)"
            if is_integer("?x".parse().unwrap())),
    ]
}

#[allow(dead_code)]
pub fn predicate_rewrites() -> Vec<RW> {
    vec![
        rewrite!("and-elim-left"; "(land ?x false)" => "false"),
        rewrite!("and-elim-right"; "(land ?x true)" => "?x"),
        rewrite!("and-comm"; "(land ?x ?y)" => "(land ?y ?x)"),
        rewrite!("and-assoc"; "(land ?x (land ?y ?z))" => "(land (land ?x ?y)
            ↪ ?z)"),
        rewrite!("and-assoc-rev"; "(land (land ?x ?y) ?z)" => "(land ?x (land
            ↪ ?y ?z))"),
        rewrite!("or-elim-left"; "(lor ?x true)" => "true"),
        rewrite!("or-elim-right"; "(lor ?x false)" => "?x"),
        rewrite!("or-comm"; "(lor ?x ?y)" => "(lor ?y ?x)"),
```

```
            rewrite!("or-assoc"; "(lor ?x (lor ?y ?z))" => "(lor (lor ?x ?y) ?z)"),
            rewrite!("demorgan-and"; "(lnot (land ?x ?y))" => "(lor (lnot ?x) (lnot ?y))"),
            rewrite!("demorgan-or"; "(lnot (lor ?x ?y))" => "(land (lnot ?x) (lnot ?y))"),
            rewrite!("demorgan-converse-and"; "(land (lnot ?x) (lnot ?y))" => "(lnot (lor ?x ?y))"),
            rewrite!("demorgan-converse-or"; "(lor (lnot ?x) (lnot ?y))" => "(lnot (land ?x ?y))"),
            // rewrite!("ite-collapse"; "(ite ?c1 ?t3 (ite ?c2 ?t1 ?t2))" => "(ite ?c1 ?t3 ?t1)"), // iff c1 <=> not c2
            rewrite!("ite-true"; "(ite true ?t1 ?t2)" => "?t1"),
            rewrite!("ite-false"; "(ite false ?t1 ?t2)" => "?t2"),
            rewrite!("ite-same"; "(ite ?c ?t ?t)" => "?t"),
            rewrite!("ite-combine"; "(ite ?c1 (ite ?c2 ?b1 ?d) ?d)" => "(ite (land ?c1 ?c2) ?b1 ?d)"),
            rewrite!("ite-intro"; "?t" => "(ite true ?t ?t)"
                    if is_integer("?t".parse().unwrap())),
            rewrite!("trivial-comp"; "true" => "(= 0 0)"),
            rewrite!("true-not-false"; "(lnot false)" => "true"),
            rewrite!("false-not-true"; "(lnot true)" => "false"),
            rewrite!("not-not"; "(lnot (lnot ?x))" => "?x"),
            rewrite!("ite-reversed-cond"; "(ite ?c ?b1 ?b2)" => "(ite (lnot ?c) ?b2 ?b1)"),
        ]
    }

    pub fn rel_comp_rewrites() -> Vec<RW> {
        vec![
            rewrite!("not-eq-neq"; "(!= ?x ?y)" => "(lnot (= ?x ?y))"),
            rewrite!("neq-not-eq"; "(lnot (= ?x ?y))" => "(!= ?x ?y)"),
            rewrite!("not-neq-neg"; "(lnot (!= ?x ?y))" => "(= ?x ?y)"),
            rewrite!("not-lt-ge"; "(lnot (< ?x ?y))" => "(>= ?x ?y)"),
            rewrite!("gt-lt"; "(> ?x ?y)" => "(< ?y ?x)"),
            rewrite!("lt-gt"; "(< ?x ?y)" => "(> ?y ?x)"),
            rewrite!("lt-to-zero-cmp"; "(< ?x ?y)" => "(< 0 (- ?x ?y))"),
            rewrite!("lt-comp-lt-0"; "(< ?x ?y)" => "(< (- ?x ?y) 0)"),
            rewrite!("gt-comp-gt-0"; "(> ?x ?y)" => "(< 0 (- ?x ?y))"),
            rewrite!("lt-comp-sub"; "(< (- ?x ?y) ?z)" => "(> ?y (- ?x ?z))"),
            rewrite!("lt-to-zero-check"; "(< ?x ?y)" => "(< (- ?x ?y) 0)"
                    if is_integer("?x".parse().unwrap())
                    if is_integer("?y".parse().unwrap())),
            rewrite!("eq-to-zero-check"; "(= ?x ?y)" => "(!= 0 (- ?x ?y))"
                    if is_integer("?x".parse().unwrap())
                    if is_integer("?y".parse().unwrap())),
            rewrite!("neq-to-zero-check"; "(!= ?x ?y)" => "(!= (- ?x ?y) 0)"
                    if is_integer("?x".parse().unwrap())
                    if is_integer("?y".parse().unwrap())),
        ]
    }
```

8.2 Target-dependent Synthesis Rules

8.3 Tofino Synthesis Rules

```
pub mod stateless {
```

```
#[allow(dead_code)]
    pub fn arith_to_alu() -> Vec<RW> {
        vec![
            rewrite!("add-to-alu"; "(+ ?x ?y)" => { AluApplier::new("arith-alu", "
                ↪ alu-add", vec!["?x", "?y"]) }
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("minus-to-alu"; "(- ?x ?y)" => { AluApplier::new("arith-alu",
                ↪  "alu-sub", vec!["?x", "?y"]) }
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("ite-to-max"; "(ite (> ?x ?y) ?x ?y)" => { AluApplier::new("
                ↪ arith-alu", "alu-max", vec!["?x", "?y"]) }
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("ite-to-min"; "(ite (< ?x ?y) ?x ?y)" => { AluApplier::new("
                ↪ arith-min", "alu-add", vec!["?x", "?y"]) }
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
        ]
    }

    pub fn cmp_to_rel() -> Vec<RW> {
        vec![
            rewrite!("gt-to-rel"; "(> ?x ?y)" => "(rel-alu alu-gt ?x ?y)"
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("lt-to-rel"; "(< ?x ?y)" => "(rel-alu alu-lt ?x ?y)"
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("eq-to-rel"; "(= ?x ?y)" => "(rel-alu alu-eq ?x ?y)"
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("neq-to-rel"; "(!= ?x ?y)" => "(rel-alu alu-neq ?x ?y)"
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("ge-to-rel"; "(>= ?x ?y)" => "(rel-alu alu-ge ?x ?y))"
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("le-to-rel"; "(<= ?x ?y)" => "(rel-alu alu-le ?x ?y))"
                    if is_mapped("?x".parse().unwrap(), None)
                    if is_mapped("?y".parse().unwrap(), None)),
        ]
    }
}

pub mod stateful {

    pub fn conditional_assignments() -> Vec<RW> {
        struct TofinoStatefulAluApplier {
            alu_type: &'static str,
            alu_op: &'static str,
            operands: Vec<Var>,
            table_id: Var,
        }
        impl TofinoStatefulAluApplier {
            fn new(
```

```
55                  alu_type: &'static str,
56                  alu_op: &'static str,
57                  table_id: &'static str,
58                  operands: Vec<&'static str>,
59              ) -> Self {
60                  Self {
61                      alu_type,
62                      alu_op,
63                      table_id: table_id.parse().unwrap(),
64                      operands: operands.into_iter().map(|s| s.parse().unwrap()).
                            ↪ collect(),
65                  }
66              }
67          }
68          impl Applier<Mio, MioAnalysis> for TofinoStatefulAluApplier {
69              fn apply_one(
70                  &self,
71                  egraph: &mut egg::EGraph<Mio, MioAnalysis>,
72                  eclass: Id,
73                  subst: &Subst,
74                  _searcher_ast: Option<&egg::PatternAst<Mio>>,
75                  _rule_name: egg::Symbol,
76              ) -> Vec<Id> {
77                  let elaborations = MioAnalysis::elaborations(egraph, eclass).clone
                        ↪ ();
78                  assert!(
79                      elaborations.len() <= 1,
80                      "conditional␣assignments␣should␣have␣at␣most␣one␣elaboration"
81                  );
82                  let elab_var = if elaborations.len() == 0 {
83                      "tmp".to_string()
84                  } else {
85                      elaborations.iter().cloned().next().unwrap()
86                  };
87                  let alu_op_id = egraph.add(if self.alu_type == "arith-alu" {
88                      Mio::ArithAluOps(self.alu_op.parse().unwrap())
89                  } else {
90                      Mio::RelAluOps(self.alu_op.parse().unwrap())
91                  });
92                  let operands = self.operands.iter().map(|v| subst[*v]).collect::<
                        ↪ Vec<_>>();
93                  let elab_id = egraph.add(Mio::Symbol(elab_var.clone()));
94                  let salu_id = egraph.add(Mio::SAlu(
95                      vec![alu_op_id, elab_id]
96                          .into_iter()
97                          .chain(operands.iter().cloned())
98                          .collect(),
99                  ));
100                 let elaborator_id =
101                     egraph.add(Mio::Elaborate([subst[self.table_id], elab_id,
                            ↪ salu_id]));
102                 let f = egraph.union(elaborator_id, eclass);
103                 if f {
104                     vec![eclass, elaborator_id]
105                 } else {
106                     vec![]
107                 }
```

```
            }
        }
        vec![rewrite!("cond-assign-to-salu";
                "(E␣?t␣?v␣(ite␣?rel␣?lhs␣?rhs))" =>
                { TofinoStatefulAluApplier::new("arith-alu", "alu-ite", "?t", vec!
                    ↪ ["?rel", "?lhs", "?rhs"]) }
            if is_n_depth_mapped("?rel".parse().unwrap(), 2, Some(false))
            if is_n_depth_mapped("?lhs".parse().unwrap(), 1, Some(false))
            if is_n_depth_mapped("?rhs".parse().unwrap(), 1, Some(false))
        )]
    }
}
```

8.4 Domino (Banzai ALU) Synthesis Rules

```
pub mod stateless {
    pub fn arith_to_alu() -> Vec<RW> {
        // https://github.com/CaT-mindepth/minDepthCompiler/blob/weighted-grammar-
            ↪ parallel-final/src/grammars/stateless_domino/stateless_new.sk
        fn neq_0_check_match(x: Var) -> impl Fn(&mut EGraph<Mio, MioAnalysis>, Id,
            ↪  &Subst) -> bool {
            move |egraph, _, subst| {
                // println!("neq_0_check_match: {}", MioAnalysis::
                    ↪ extract_smallest_expr(egraph, subst[x]));
                if MioAnalysis::get_symbol(egraph, subst[x]).is_some() {
                    return true;
                }
                let pattern = "(!=␣?x␣0)";
                let pattern = pattern.parse::<Pattern<Mio>>().unwrap();
                egraph.rebuild();
                pattern.search_eclass(egraph, subst[x]).is_some()
            }
        }
        vec![
            rewrite!("domino-add";
                "(+␣?x␣?y)" => { AluApplier::new("arith-alu", "alu-add", vec!["?x"
                    ↪ , "?y"]) }
                if is_mapped("?x".parse().unwrap(), Some(false))
                if is_mapped("?y".parse().unwrap(), Some(false))),
            rewrite!("domino-sub";
                "(-␣?x␣?y)" => { AluApplier::new("arith-alu", "alu-sub", vec!["?x"
                    ↪ , "?y"]) }
                if is_mapped("?x".parse().unwrap(), Some(false))
                if is_mapped("?y".parse().unwrap(), Some(false))),
            rewrite!("domino-neq";
                "(!=␣?x␣?y)" => { AluApplier::new("rel-alu", "alu-neq", vec!["?x",
                    ↪  "?y"]) }
                if is_mapped("?x".parse().unwrap(), None)
                if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("domino-eq";
                "(=␣?x␣?y)" => { AluApplier::new("rel-alu", "alu-eq", vec!["?x", "
                    ↪ ?y"]) }
                if is_mapped("?x".parse().unwrap(), None)
                if is_mapped("?y".parse().unwrap(), None)),
            rewrite!("domino-geq";
```

```
34                  "(>=␣?x␣?y)" => { AluApplier::new("rel-alu", "alu-ge", vec!["?x",
                        ↪ "?y"]) }
35                  if is_mapped("?x".parse().unwrap(), None)
36                  if is_mapped("?y".parse().unwrap(), None)),
37              rewrite!("domino-lt";
38                  "(<␣?x␣?y)" => { AluApplier::new("rel-alu", "alu-lt", vec!["?x", "
                        ↪ ?y"]) }
39                  if is_mapped("?x".parse().unwrap(), None)
40                  if is_mapped("?y".parse().unwrap(), None)),
41              // rewrite!("domino-ite-0";
42              //     "(ite ?c ?x ?y)" => { AluApplier::new("arith-alu", "alu-ite",
                    ↪ vec!["?c", "?x", "?y"]) }
43              //     if neq_0_check_match("?c".parse().unwrap())),
44              rewrite!("domino-neq-0-or";
45                  "(lor␣?x␣?y)" => { AluApplier::new("rel-alu", "alu-or", vec!["?x",
                        ↪  "?y"]) }
46                  if neq_0_check_match("?x".parse().unwrap())
47                  if neq_0_check_match("?y".parse().unwrap())),
48              rewrite!("domino-neq-0-and";
49                  "(land␣?x␣?y)" => { AluApplier::new("rel-alu", "alu-and", vec!["?x
                        ↪ ", "?y"]) }
50                  if neq_0_check_match("?x".parse().unwrap())
51                  if neq_0_check_match("?y".parse().unwrap())),
52          ]
53      }
54  }
55
56  #[allow(dead_code)]
57  pub mod stateful {
58      fn check_relops(
59          v: Var,
60          operators: Vec<&'static str>,
61      ) -> impl Fn(&mut EGraph<Mio, MioAnalysis>, Id, &Subst) -> bool {
62          move |egraph, _, subst| {
63              let vid = subst[v];
64              assert_eq!(egraph[vid].nodes.len(), 1);
65              let operators = operators
66                  .iter()
67                  .map(|op| Mio::RelAluOps(op.parse().unwrap()))
68                  .collect::<Vec<Mio>>();
69              return operators.iter().any(|x| x.eq(&egraph[vid].nodes[0]));
70          }
71      }
72
73      fn check_arith_alu(v: Var) -> impl Fn(&mut EGraph<Mio, MioAnalysis>, Id, &
            ↪ Subst) -> bool {
74          move |egraph, _, subst| {
75              let add_pattern = "(arith-alu␣alu-add␣?x␣?y)".parse::<Pattern<Mio>>().
                    ↪ unwrap();
76              let sub_pattern = "(arith-alu␣alu-sub␣?x␣?y)".parse::<Pattern<Mio>>().
                    ↪ unwrap();
77              let vid = subst[v];
78              if MioAnalysis::has_leaf(egraph, vid) {
79                  // constant / variable is ok
80                  return true;
81              }
82              egraph.rebuild();
```

```
            if let Some(matches) = add_pattern
                .search_eclass(egraph, vid)
                .or(sub_pattern.search_eclass(egraph, vid))
            {
                // Check whether ?x and ?y are leaves
                let var_x = "?x".parse().unwrap();
                let var_y = "?y".parse().unwrap();
                return matches.substs.iter().any(|subst| {
                    MioAnalysis::has_leaf(egraph, subst[var_x])
                        && MioAnalysis::has_leaf(egraph, subst[var_y])
                });
            } else {
                return false;
            }
        }
    }

    pub fn bool_alu_rewrites() -> Vec<RW> {
        vec![
            rewrite!("domino-stateful-bool-alu-1";
                "(lnot (lor ?x ?y))" => "(rel-alu_alu-not (rel-alu_alu-or ?x ?y))"
                if constains_leaf("?x".parse().unwrap())
                if constains_leaf("?y".parse().unwrap())),
            rewrite!("domino-stateful-bool-alu-2";
                "(land (lnot ?x) ?y)" => "(rel-alu_alu-and (rel-alu_alu-not ?x) ?y
                    ↪ )"
                if constains_leaf("?x".parse().unwrap())
                if constains_leaf("?y".parse().unwrap())),
            rewrite!("domino-stateful-bool-alu-3";
                "(lnot ?x)" => "(rel-alu_alu-not ?x)"
                if constains_leaf("?x".parse().unwrap())),
            rewrite!("domino-stateful-bool-alu-and";
                "(land ?x ?y)" => "(rel-alu_alu-and ?x ?y)"
                if constains_leaf("?x".parse().unwrap())
                if constains_leaf("?y".parse().unwrap())),
            rewrite!("domino-stateful-bool-alu-or";
                "(lor ?x ?y)" => "(rel-alu_alu-or ?x ?y)"
                if constains_leaf("?x".parse().unwrap())
                if constains_leaf("?y".parse().unwrap())),
            rewrite!("domino-stateful-bool-alu-4";
                "(lnot (land ?x ?y))" => "(rel-alu_alu-not (rel-alu_alu-and ?x ?y)
                    ↪ )"
                if constains_leaf("?x".parse().unwrap())
                if constains_leaf("?y".parse().unwrap())),
            rewrite!("domino-stateful-bool-alu-5";
                "(lor (lnot ?x) ?y)" => "(rel-alu_alu-or (rel-alu_alu-not ?x) ?y)"
                if constains_leaf("?x".parse().unwrap())
                if constains_leaf("?y".parse().unwrap())),
        ]
    }

    pub fn global_or_0(v: Var) -> impl Fn(&mut EGraph<Mio, MioAnalysis>, Id, &
        ↪ Subst) -> bool {
        move |egraph, _, subst| {
            let vid = subst[v];
            if let Some(c) = MioAnalysis::get_constant(egraph, vid) {
                return c == Constant::Int(0) || c == Constant::Bool(false);
```

```
137             }
138             if let Some(sym) = MioAnalysis::get_symbol(egraph, vid) {
139                 return sym.starts_with("global.");
140             } else {
141                 return false;
142             }
143         }
144     }
145
146     pub fn non_global(v: Var) -> impl Fn(&mut EGraph<Mio, MioAnalysis>, Id, &Subst
        ↪ ) -> bool {
147         move |egraph, _, subst| {
148             let vid = subst[v];
149             return MioAnalysis::stateful_read_count(egraph, vid) == 0
150                 && !MioAnalysis::has_stateful_elaboration(egraph, vid);
151         }
152     }
153
154     pub fn same_if_is_var(
155         v1: Var,
156         v2: Var,
157     ) -> impl Fn(&mut EGraph<Mio, MioAnalysis>, Id, &Subst) -> bool {
158         move |egraph, _, subst| {
159             let vid1 = subst[v1];
160             let vid2 = subst[v2];
161             if let (Some(v1_sym), Some(v2_sym)) = (
162                 MioAnalysis::get_symbol(egraph, vid1),
163                 MioAnalysis::get_symbol(egraph, vid2),
164             ) {
165                 if v1_sym.starts_with("global.") && v2_sym.starts_with("global.")
                    ↪ {
166                     return v1_sym == v2_sym;
167                 } else {
168                     return true;
169                 }
170             } else {
171                 return true;
172             }
173         }
174     }
175
176     pub fn check_read_limit(
177         vars: Vec<&'static str>,
178         phv_field_limit: usize,
179         global_reg_limit: usize,
180     ) -> impl Fn(&mut EGraph<Mio, MioAnalysis>, Id, &Subst) -> bool {
181         move |egraph, _, subst| {
182             let mut local_read = HashSet::new();
183             let mut global_read = HashSet::new();
184             for v in vars.iter() {
185                 let vid = subst[v.parse().unwrap()];
186                 let read_set = MioAnalysis::read_set(egraph, vid);
187                 global_read.extend(read_set.iter().filter(|x| x.starts_with("
                    ↪ global.")));
188                 local_read.extend(read_set.iter().filter(|x| !x.starts_with("
                    ↪ global.")));
189             }
```

```
190             return local_read.len() <= phv_field_limit && global_read.len() <=
                    ↪ global_reg_limit;
191         }
192     }
193
194     pub fn if_else_raw() -> Vec<RW> {
195         // https://github.com/CaT-mindepth/minDepthCompiler/blob/weighted-grammar-
                ↪ parallel-final/src/grammars/stateful_domino/if_else_raw.sk
196         struct IfElseApplier {
197             op: Var,
198             r: Var,
199             rhs: Var,
200             x: Var,
201             y: Var,
202             z: Var,
203             w: Var,
204         }
205         impl Applier<Mio, MioAnalysis> for IfElseApplier {
206             fn apply_one(
207                 &self,
208                 egraph: &mut EGraph<Mio, MioAnalysis>,
209                 eclass: Id,
210                 subst: &Subst,
211                 searcher_ast: Option<&egg::PatternAst<Mio>>,
212                 rule_name: egg::Symbol,
213             ) -> Vec<Id> {
214                 let elaborations = MioAnalysis::elaborations(egraph, eclass);
215                 let evar = if elaborations.len() == 0 {
216                     "tmp".to_string()
217                 } else {
218                     elaborations.iter().next().unwrap().clone()
219                 };
220                 let pattern = format!(
221                     "(E␣?t␣?v␣(stateful-alu␣alu-ite␣{}
222 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(rel-alu␣{}␣{}␣{})
223 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(arith-alu␣alu-add␣{}␣{})
224 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(arith-alu␣alu-add␣{}␣{})))",
225                     evar, self.op, self.r, self.rhs, self.z, self.x, self.w, self.
                            ↪ y
226                 );
227                 return pattern.parse::<Pattern<Mio>>().unwrap().apply_one(
228                     egraph,
229                     eclass,
230                     subst,
231                     searcher_ast,
232                     rule_name,
233                 );
234             }
235         }
236         vec![rewrite!("domino-if-else-raw";
237                 "(E␣?t␣?v␣(ite
238 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(rel-alu␣?op␣?r␣?rhs)
239 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(+␣?z␣?x)
240 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(+␣?w␣?y)
241 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣))" => { IfElseApplier {
242                     op: "?op".parse().unwrap(),
243                     r: "?r".parse().unwrap(),
```

```
                    rhs: "?rhs".parse().unwrap(),
                    x: "?x".parse().unwrap(),
                    y: "?y".parse().unwrap(),
                    z: "?z".parse().unwrap(),
                    w: "?w".parse().unwrap(),
                } }
                if check_relops("?op".parse().unwrap(), vec!["alu-eq", "alu-neq",
                    ↪ "alu-gt", "alu-lt"])
                // if global_or_0("?r".parse().unwrap())
                // if global_or_0("?z".parse().unwrap())
                // if global_or_0("?w".parse().unwrap())
                if same_if_is_var("?v".parse().unwrap(), "?r".parse().unwrap())
                if same_if_is_var("?v".parse().unwrap(), "?z".parse().unwrap())
                if same_if_is_var("?v".parse().unwrap(), "?w".parse().unwrap())
                if none_global("?rhs".parse().unwrap())
                if constains_leaf("?rhs".parse().unwrap())
                if constains_leaf("?x".parse().unwrap())
                if constains_leaf("?y".parse().unwrap()))]
    }

    pub fn nested_ifs() -> Vec<RW> {
        // https://github.com/CaT-mindepth/minDepthCompiler/blob/weighted-grammar-
            ↪ parallel-final/src/grammars/stateful_domino/nested_ifs.sk
        struct NestedIfsApplier;
        impl Applier<Mio, MioAnalysis> for NestedIfsApplier {
            fn apply_one(
                &self,
                egraph: &mut EGraph<Mio, MioAnalysis>,
                eclass: Id,
                subst: &Subst,
                searcher_ast: Option<&egg::PatternAst<Mio>>,
                rule_name: egg::Symbol,
            ) -> Vec<Id> {
                // check conditions
                let stateless_read_violation =
                    |v: Id| MioAnalysis::stateless_read_count(egraph, v) > 1;
                let c1 = subst["?c1".parse().unwrap()];
                let c2 = subst["?c2".parse().unwrap()];
                let c3 = subst["?c3".parse().unwrap()];
                if MioAnalysis::stateful_read_count(egraph, c1)
                    + MioAnalysis::stateful_read_count(egraph, c2)
                    + MioAnalysis::stateful_read_count(egraph, c3)
                    > 1
                {
                    return vec![];
                }
                let b1 = subst["?b1".parse().unwrap()];
                let b2 = subst["?b2".parse().unwrap()];
                let b3 = subst["?b3".parse().unwrap()];
                let b4 = subst["?b4".parse().unwrap()];
                if MioAnalysis::stateful_read_count(egraph, b1)
                    + MioAnalysis::stateful_read_count(egraph, b2)
                    + MioAnalysis::stateful_read_count(egraph, b3)
                    + MioAnalysis::stateful_read_count(egraph, b4)
                    > 1
                {
                    return vec![];
```

```
                }
                if [b1, b2, b3, b4, c1, c2, c3]
                    .into_iter()
                    .any(stateless_read_violation)
                {
                    return vec![];
                }
                let elaborations = MioAnalysis::elaborations(egraph, eclass);
                let evar = if elaborations.len() == 0 {
                    "tmp".to_string()
                } else {
                    format!("{}", elaborations.iter().next().unwrap().clone())
                };
                let pattern = format!(
                    "(E ?t ?v (stateful-alu alu-ite {}
                        ?c1
                        (stateful-alu alu-ite tmp
                            ?c2
                            ?b1
                            ?b2
                        )
                        (stateful-alu alu-ite tmp
                            ?c3
                            ?b3
                            ?b4
                        )
                    ))",
                    evar,
                );
                return pattern.parse::<Pattern<Mio>>().unwrap().apply_one(
                    egraph,
                    eclass,
                    subst,
                    searcher_ast,
                    rule_name,
                );
            }
        }
        vec![rewrite!("domino-stateful-nested-ifs";
                "(E ?t ?v (ite
                    ?c1
                    (ite
                        ?c2
                        ?b1
                        ?b2
                    )
                    (ite
                        ?c3
                        ?b3
                        ?b4
                    )
                ))" => { NestedIfsApplier {} }
            if is_n_depth_mapped("?b1".parse().unwrap(), 1, None)
            if is_n_depth_mapped("?b2".parse().unwrap(), 1, None)
            if is_n_depth_mapped("?b3".parse().unwrap(), 1, None)
            if is_n_depth_mapped("?b4".parse().unwrap(), 1, None)
            if is_n_depth_mapped("?c1".parse().unwrap(), 1, None)
```

```
356             if is_n_depth_mapped("?c2".parse().unwrap(), 1, None)
357             if is_n_depth_mapped("?c3".parse().unwrap(), 1, None)
358         )]
359     }
360
361     pub fn pred_raw() -> Vec<RW> {
362         struct PredRawApplier {
363             op: Var,
364             r: Var,
365             rhs: Var,
366             r1: Var,
367             x: Var,
368             g: Var,
369         }
370         impl Applier<Mio, MioAnalysis> for PredRawApplier {
371             fn apply_one(
372                 &self,
373                 egraph: &mut EGraph<Mio, MioAnalysis>,
374                 eclass: Id,
375                 subst: &Subst,
376                 searcher_ast: Option<&egg::PatternAst<Mio>>,
377                 rule_name: egg::Symbol,
378             ) -> Vec<Id> {
379                 let elaborations = MioAnalysis::elaborations(egraph, eclass);
380                 let evar = if elaborations.len() == 0 {
381                     "tmp".to_string()
382                 } else {
383                     elaborations.iter().next().unwrap().clone()
384                 };
385                 let pattern = format!(
386                     "(E ?t ?v (stateful-alu alu-ite {}
387 (rel-alu {} {} {})
388 (arith-alu alu-add {} {})
389 ?g
390 ))",
391                     evar, self.op, self.r, self.rhs, self.r1, self.x
392                 );
393                 return pattern.parse::<Pattern<Mio>>().unwrap().apply_one(
394                     egraph,
395                     eclass,
396                     subst,
397                     searcher_ast,
398                     rule_name,
399                 );
400             }
401         }
402         vec![rewrite!("domino-stateful-pred-raw";
403             "(E ?t ?v (ite
404 (rel-alu ?op ?r ?rhs)
405 (+ ?r1 ?x)
406 ?g
407 ))" => { PredRawApplier {
408                 op: "?op".parse().unwrap(),
409                 r: "?r".parse().unwrap(),
410                 rhs: "?rhs".parse().unwrap(),
411                 r1: "?r1".parse().unwrap(),
412                 x: "?x".parse().unwrap(),
```

```
413                 g: "?g".parse().unwrap(),
414             } }
415             if check_relops("?op".parse().unwrap(), vec!["alu-eq", "alu-neq", "alu
                    ↪ -gt", "alu-lt"])
416             if global_or_0("?r".parse().unwrap())
417             if global_or_0("?r1".parse().unwrap())
418             if global_or_0("?g".parse().unwrap())
419             if same_if_is_var("?v".parse().unwrap(), "?r".parse().unwrap())
420             if same_if_is_var("?v".parse().unwrap(), "?r1".parse().unwrap())
421             if same_if_is_var("?v".parse().unwrap(), "?g".parse().unwrap())
422             if constains_leaf("?rhs".parse().unwrap())
423             if constains_leaf("?x".parse().unwrap()))]
424     }
425
426     pub fn stateful_ite_simpl() -> Vec<RW> {
427         struct SAluIteSimplApplier {
428             comp: Var,
429         }
430         impl Applier<Mio, MioAnalysis> for SAluIteSimplApplier {
431             fn apply_one(
432                 &self,
433                 egraph: &mut EGraph<Mio, MioAnalysis>,
434                 eclass: Id,
435                 subst: &Subst,
436                 _searcher_ast: Option<&egg::PatternAst<Mio>>,
437                 _rule_name: egg::Symbol,
438             ) -> Vec<Id> {
439                 let comp_id = subst[self.comp];
440                 if let Ok((_op_name, args)) = MioAnalysis::decompose_alu_ops(
                        ↪ egraph, comp_id) {
441                     let vid = subst["?v".parse().unwrap()];
442                     if let Ok(op_id) = MioAnalysis::get_alu_op_id(egraph, comp_id)
                            ↪ {
443                         let salu_id = egraph.add(Mio::SAlu(
444                             vec![op_id, vid]
445                                 .into_iter()
446                                 .chain(args.into_iter())
447                                 .collect(),
448                         ));
449                         egraph.union(eclass, salu_id);
450                         return vec![salu_id];
451                     } else {
452                         vec![]
453                     }
454                 } else {
455                     vec![]
456                 }
457             }
458         }
459         vec![
460             rewrite!("stateful-ite-simpl"; "(stateful-alu␣alu-ite␣?v␣true␣?t1␣?t2)
                    ↪ "
461             => {
462                 SAluIteSimplApplier {
463                     comp: "?t1".parse().unwrap()
464                 }
465             }),
```

```
            rewrite!("stateful-ite-same-branch-simpl";
            "(stateful-alu␣alu-ite␣?v␣?c␣?t1␣?t1)" => {
                SAluIteSimplApplier {
                    comp: "?t1".parse().unwrap()
                }
            }),
        ]
    }
}
```