Refinement of Path Expressions for Static Analysis

JOHN CYPHERT, University of Wisconsin, USA
JASON BRECK, University of Wisconsin, USA
ZACHARY KINCAID, Princeton University, USA
THOMAS REPS, University of Wisconsin, USA and GrammaTech, Inc., USA

Algebraic program analyses compute information about a program's behavior by first (a) computing a valid *path expression*—i.e., a regular expression that recognizes all feasible execution paths (and usually more)—and then (b) interpreting the path expression in a semantic algebra that defines the analysis. There are an infinite number of different regular expressions that qualify as valid path expressions, which raises the question "*Which one should we choose?*" While any choice yields a sound result, for many analyses the choice can have a drastic effect on the precision of the results obtained. This paper investigates the following two questions:
(1) *What does it mean for one valid path expression to be "better" than another?*
(2) *Can we compute a valid path expression that is "better," and if so, how?*
We show that it is not satisfactory to compare two path expressions E_1 and E_2 solely by means of the *languages that they generate*. Counter to one's intuition, it is possible for $L(E_2) \subsetneq L(E_1)$, yet for E_2 to produce a *less-precise* analysis result than E_1—and thus we would not want to perform the transformation $E_1 \rightarrow E_2$. However, the exclusion of paths so as to analyze a smaller language of paths is exactly the refinement criterion used by some prior methods.

In this paper, we develop an algorithm that takes as input a valid path expression E, and returns a valid path expression E' that is guaranteed to yield analysis results that are at least as good as those obtained using E. While the algorithm sometimes returns E itself, it typically does not: (i) we prove a *no-degradation result* for the algorithm's base case—for transforming a leaf loop (i.e., a most-deeply-nested loop); (ii) at a non-leaf loop L, the algorithm treats each loop L' in the body of L as an indivisible atom, and applies the leaf-loop algorithm to L; the no-degradation result carries over to (ii), as well. Our experiments show that the technique has a substantial impact: the loop-refinement algorithm allows the implementation of Compositional Recurrence Analysis to prove over 25% more assertions for a collection of challenging loop micro-benchmarks.

CCS Concepts: • **Theory of computation → Program analysis**; **Regular languages**; • **Software and its engineering → Automated static analysis**;

Additional Key Words and Phrases: Algebraic program analysis, control-flow refinement, abstract-interpretation precision

ACM Reference Format:
John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps. 2019. Refinement of Path Expressions for Static Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 45 (January 2019), 29 pages. https://doi.org/10.1145/3290358

Authors' addresses: John Cyphert, jcyphert@wisc.edu, University of Wisconsin, Madison, WI, USA; Jason Breck, jbreck@wisc.edu, University of Wisconsin, Madison, WI, USA; Zachary Kincaid, zkincaid@cs.princeton.edu, Princeton University, Princeton, NJ, USA; Thomas Reps, reps@cs.wisc.edu, University of Wisconsin, Madison, WI, USA, GrammaTech, Inc. Ithaca, NY, USA.

2475-1421/2019/1-ART45
https://doi.org/10.1145/3290358

1 INTRODUCTION

Tarjan [1981b] introduced the idea of using regular expressions as a kind of "most-general program-analysis method." This style of analysis is also sometimes referred to as *algebraic program analysis* [Farzan and Kincaid 2013; Kincaid et al. 2017]. Specific program-analysis problems are solved by first solving the *path-expression problem*: a program's control-flow graph (CFG) is considered to be a finite-state machine in which CFG nodes are states, and each edge is labeled by an alphabet symbol unique to that edge. Tarjan's path-expression method [Tarjan 1981a] creates for each node n a regular expression R_n whose language, $L(R_n)$, is the set of all paths from the CFG's start node to n. The "client" program-analysis problem is then solved by evaluating each regular expression R_n bottom-up, using an interpretation in which the regular-expression operators $+$, $\cdot$, and $*$—now treated as syntactic operators—are interpreted as some suitable (sound) operations, $\oplus$, $\otimes$, and $\circledast$, respectively, in the analysis domain.

The idea of applying transformations to the intermediate representation (IR) of a program as a way to improve the results of static analysis has a long history (see §7). For instance, many people have taken advantage of loop unrolling as a way to potentially improve analysis precision. In the nomenclature of algebraic program analysis, the question can be phrased as follows:[1]

> There are many different regular expressions that represent the same set of concrete-action sequences; are some of these regular expressions better than others? In other words, can the evaluation of one regular expression produce more precise analysis results, compared with a structurally different—but equivalent—regular expression? (1)

Not surprisingly, the answer is "Yes. The structure of the regular expression matters."

In this paper, we focus on a particular sub-problem related to loops: given a regular expression of the form $R = (r_1 + \ldots + r_n)^*$ and a set of forbidden (infeasible) subpaths, compute a new regular expression R' such that (a) the forbidden subpaths are eliminated from R', and (b) the precision of the analysis results obtained with R' are at least as good as those obtained with R.

Past work that has studied refinement has worked only with requirement (a). For that situation—studied in the context of iterative program analysis under the name *control-flow refinement* [Balakrishnan et al. 2009; Flores-Montoya and Hähnle 2014; Gulwani et al. 2009; Sharma et al. 2011]—the solution exploits the observation that the syntax of a loop (or the structure of the IR that represents the loop) may cause more paths through the loop to be considered than are actually possible. A set of infeasible paths can be characterized by a set of forbidden subwords; consequently, the language of paths *not* containing forbidden subwords is regular, and therefore recognized by a regular expression, say F. One can then obtain the analysis results for the loop by evaluating F instead of R.

Counter to one's intuition, however, the analysis of F can yield a *less-precise* summary of the loop than merely evaluating $(r_1 + \ldots + r_m)^*$. (See §2.2.) This conundrum suggests the following problem: find a regular expression R' that is *between* F and R—i.e., $L(F) \subseteq L(R') \subsetneq L(R)$—and which gives analysis results that are no worse than those obtained with R. In this paper, we give an algorithm for computing such an R'.

As just mentioned, it is unsatisfactory to compare regular expressions *solely* by means of the languages that they generate: the comparison must account for the fact that they are both *interpreted* in the abstract domain $\mathcal{A}$ in use. In particular, we wish to establish that path expression R' yields $\mathcal{A}$-analysis results that are at least as good as those obtained with R. Moreover, we wish to do so by some means other than (i) creating R', (ii) evaluating R' in $\mathcal{A}$, and (iii) comparing the result

[1] "Equivalent" is used in the sense that both expressions represent the same set of concrete-action sequences.

to the value of R evaluated in $\mathcal{A}$. Instead, we wish to reason about the relative precision of the $\mathcal{A}$-interpretations of R and R'. We address this issue by axiomatizing the properties of an abstract domain that influence the relative precision obtained using different regular expressions. Our name for the formalism that we introduce is "**pre-Kleene algebra**" (PKA) (§3). By assuming that abstract domain $\mathcal{A}$ satisfies the PKA axioms, we can use algebraic reasoning to prove properties of the transformation algorithm.

The second problem addressed in this paper is how to use our understanding of the simple case above in a program analyzer. There are two relevant problems: (1) how to generalize the results to arbitrary regular expressions, and (2) how to recognize infeasible sub-paths. We exploit the compositional nature of algebraic program analysis to provide answers to both questions. The result is a generic algorithm that *provably* does not degrade the precision of any algebraic program analysis meeting our conditions, and, as our experiments show, often improves the precision.

Our experiments show that the technique has a substantial impact: the loop-refinement algorithm allows the implementation of Compositional Recurrence Analysis (CRA) [Farzan and Kincaid 2015] to prove over 25% more assertions for a collection of challenging loop micro-benchmarks.

Contributions. Our work makes the following contributions:

- We prove a *no-degradation result* (Thm. 4.5) for the algorithm's base case, which transforms a leaf loop.
- At non-leaf loops, the algorithm applies the leaf-loop algorithm greedily, bottom-up (§5). We show that the non-degradation result carries over to the non-leaf loops, as well. Consequently, our bottom-up algorithm for refinement is guaranteed to yield analysis results that are at least as good as those obtained without refinement (Thm. 5.2).

  For an algebraic-analysis framework, the benefits of this approach are three-fold:
  1. The algorithm can be used to (often) improve the precision of *any* algebraic program analysis that uses an abstract domain that satisfies the PKA axioms.
  2. The refinement method can be incorporated in a *uniform* way. Refinement is invoked at the level of the pre-existing star operator, which only has to consider regular expressions of the form R^*, where R has already been analyzed. Moreover, checks for infeasible paths in R can be made in the abstract domain (e.g., using an SMT solver). The infeasible paths can be used to refine R^* to create an alternative expression R'. Then R' is evaluated in place of R^*, and the result is used in evaluating the parent expression.
  3. Thanks to the analysis method developed by Kincaid et al. [2017], the algorithm also applies to programs with recursive procedure calls.
- We have incorporated an implementation of our refinement algorithm into the algebraic-analysis tool described by Farzan and Kincaid [2015]. §6 presents empirical results that demonstrate the practical benefits of our method.

§2 motivates the problem of path-expression refinement by showing how the precision of an example analysis can be increased by soundly modifying the path-expression that is used. It also gives an example showing that naive refinement can lead to worse analysis results. §3 gives background on Kleene algebras, algebraic program analysis, and control-flow refinement, and introduces the formalism of pre-Kleene algebras. §7 discusses related work.

2 OVERVIEW

In this section, we present two motivating examples based on a Tarjan-style analysis. We consider an example abstract domain equipped with extend, combine, and iteration operators, denoted by $\otimes$, $\oplus$, and $\circledast$, respectively. To be concrete, we consider an abstract domain in which each element represents a transition relation: let $\boldsymbol{x}$ denote a finite set of program variables, and let each element

```
x = 0;  y = 50;
while (x < 100) {
    x = x + 1;
    if (x > 50)
        y = y + 1;
}
```

(a)

```
i = 0; j = 0;
while (*) {
    if (pos > 0){
        if (*){   //A
            i = i + 1; j = j + 1;
            pos = 1;
        } else{   //B
            i = i + 1; j = j + 1;
            pos = 0;
        }
    } else{   //C
        i = i + 1; j = j + 1;
        pos = 1;
    }
}
```

(b)

Fig. 1. (a) A multi-path loop that shows the benefits of refinement on static-analysis results. (b) A multi-path loop for which refinement can produce less-precise static-analysis results.

be a "two-vocabulary" formula $\phi(\boldsymbol{x}, \boldsymbol{x}')$ in the existential fragment of Presburger arithmetic over the variables $\boldsymbol{x}$, plus a set of primed copies $\boldsymbol{x}'$. Such a formula represents a relation between pre-states (over $\boldsymbol{x}$) and post-states (over $\boldsymbol{x}'$). The $\otimes$ operation is relational composition, and $\oplus$ is disjunction,

$$\phi \otimes \psi = \exists \boldsymbol{x}''.\phi(\boldsymbol{x}, \boldsymbol{x}'') \wedge \psi(\boldsymbol{x}'', \boldsymbol{x}') \qquad \phi \oplus \psi = \phi \vee \psi.$$

For the $\circledast$ operator, we consider a two-step process. To compute $\phi^{\circledast}$, first compute the best approximation of ϕ as an octagonal relation $w(\boldsymbol{x}, \boldsymbol{x}')$ using optimization modulo theories [Li et al. 2014; Sebastiani and Tomasi 2012]; then compute an existential Presburger formula representing the transitive closure of $w(\boldsymbol{x}, \boldsymbol{x}')$ using the algorithm of Bozga et al. [2009].

2.1 A Transformation that Leads to an Improved Result

Consider the program in Fig. 1(a). There are two paths through the loop's body. Let A denote the path in which the then branch is avoided, and B denote the path that follows the then branch. Let ϕ_A and ϕ_B be the transition formulas for paths A and B, respectively.

$$\phi_A \overset{\text{def}}{=} (x < 100 \wedge x' = x + 1 \wedge x' \leq 50 \wedge y' = y)$$
$$\phi_B \overset{\text{def}}{=} (x < 100 \wedge x' = x + 1 \wedge x' > 50 \wedge y' = y + 1)$$

The simplest path expression that represents the set of paths from just before the loop to just after the loop is $R \overset{\text{def}}{=} (A + B)^*$. To analyze the loop, we evaluate R with A and B replaced by ϕ_A and ϕ_B, respectively; $+$ replaced by $\oplus$; and $*$ replaced by $\circledast$: $(\phi_A \oplus \phi_B)^{\circledast} = (\phi_A \vee \phi_B)^{\circledast}$. To evaluate $(\phi_A \vee \phi_B)^{\circledast}$, we first abstract $\phi_A \vee \phi_B$ to create an over-approximating formula for the loop-body's transition relation. In particular, we create the most-precise over-approximation of $\phi_A \vee \phi_B$ that is in *conjunctive form*:

$$w_{\text{body}} = (x < 100 \wedge x' - x = 1 \wedge 1 \geq y' - y \wedge y' - y \geq 0).$$

In w_{body}, we have only an inequality that relates y and y'.

Even with the most-precise computation of the closure $w_{\text{body}}^{\circledast}$, the best property that can be deduced for the program's final state from $(x = 0 \wedge y = 50) \otimes w_{\text{body}}^{\circledast} \otimes (x \geq 100)$ is $x = 100 \wedge 100 \geq y \geq 50$. However, it is not difficult to see that $x = 100 \wedge y = 100$ always holds after the program executes.

The main issue is that $(A + B)^*$ describes a larger set of paths than the set of feasible execution paths. In particular, as observed by Sharma et al. [2011], once the predicate $x > 50$ is true on some iteration, it will continue to be true on all later iterations: path A will never execute after path B. One way that this property can be discovered is by showing that the formula

$$\phi_B \otimes \phi_A = (x < 100 \land x + 1 > 50 \land x + 1 < 100 \land x + 2 \leq 50 \land x' = x + 2 \land y' = y + 1)$$

is unsatisfiable, which can be established easily by an SMT solver. Consequently, we can refine the regular expression R to $R' \overset{\text{def}}{=} A^*B^*$, which is both (i) sound with respect to the program's semantics, and (ii) more closely represents the feasible execution paths of the program. In other words, the set of feasible paths of the program is contained in $L(A^*B^*)$, and $L(A^*B^*) \subsetneq L((A + B)^*)$.

Now consider the evaluation of $(x = 0 \land y = 50) \otimes \phi_A^{\circledast} \otimes \phi_B^{\circledast}$. It happens that ϕ_A and ϕ_B are octagons already, so $\phi_A^{\circledast}$ and $\phi_B^{\circledast}$ can be computed via the methods of Bozga et al. [2009]. $(x = 0 \land y = 50) \otimes \phi_A^{\circledast}$ yields $(x = 50 \land y = 50)$ and $(x = 50 \land y = 50) \otimes \phi_B^{\circledast}$ becomes $(x = 100 \land y = 100)$. Thus the analysis of A^*B^* allows us to conclude $x = 100 \land y = 100$ after the loop. This example shows that a simple rewrite of the regular expression being analyzed can produce a more precise static-analysis result.

2.2 A Transformation that Leads to a Worse Result

Consider the program in Fig. 1(b). The loop body has three paths, which we call A, B, and C. The loop can be described by the regular expression $R_1 \overset{\text{def}}{=} (A + B + C)^*$. Let ϕ_A, ϕ_B, and ϕ_C be two-vocabulary formulas representing the transition relations for paths A, B, and C, respectively. We have

$$\phi_A \overset{\text{def}}{=} (pos > 0 \land i' = i + 1 \land j' = j + 1 \land pos' = 1)$$

$$\phi_B \overset{\text{def}}{=} (pos > 0 \land i' = i + 1 \land j' = j + 1 \land pos' = 0)$$

$$\phi_C \overset{\text{def}}{=} (pos \leq 0 \land i' = i + 1 \land j' = j + 1 \land pos' = 1)$$

Let us now evaluate R_1 in the abstract domain defined at the beginning of this section: $(\phi_A \oplus \phi_B \oplus \phi_C)^{\circledast}$. When we abstract the disjunction $\phi_A \lor \phi_B \lor \phi_C$ we obtain the formula w_{Body},

$$w_{Body} \overset{\text{def}}{=} (i' = i + 1 \land j' = j + 1 \land 0 \leq pos' \land pos' \leq 1).$$

The closure of this formula, $w_{Body}^{\circledast}$, along with the fact that $i = 0 \land j = 0$ before the loop, implies that $i = j$ holds after the loop.

Now consider a plausible alternative way of analyzing the program in Fig. 1(b). By inspecting the individual paths of the loop, we observe that $\phi_A \otimes \phi_C$, $\phi_B \otimes \phi_A$, $\phi_B \otimes \phi_B$, and $\phi_C \otimes \phi_C$ are all unsatisfiable. Using this observation, we refine the original regular expression R_1 to obtain a new regular expression, $R_2 \overset{\text{def}}{=} (\epsilon + C)(A + BC)^*(\epsilon + B)$ whose words never contain the subsequences $\ldots AC \ldots$, $\ldots BA \ldots$, $\ldots BB \ldots$, and $\ldots CC \ldots$. Note that $L(R_2) \subsetneq L(R_1)$, and that R_2 has fewer disjunctions that appear under a $*$ operator. For these reasons, we might expect that R_2 would produce a more-precise result than R_1. Counter-intuitively, R_2 produces a *less-precise* result.

To see why, consider the evaluation of R_2 in the abstract domain: $(1 \oplus \phi_C) \otimes (\phi_A \oplus \phi_B \otimes \phi_C)^{\circledast} \otimes (1 \oplus \phi_B)$. We first compute

$$\phi_{BC} \overset{\text{def}}{=} \phi_B \otimes \phi_C = (pos > 0 \land i' = i + 2 \land j' = j + 2 \land pos' = 1).$$

We then combine ϕ_A with ϕ_{BC} to obtain the formula:

$$\begin{aligned}\phi_{A+BC} \overset{\text{def}}{=} (i' = i + 1 \land j' = j + 1 \land pos \geq 0 \land pos' = 1) \lor \\ (i' = i + 2 \land j' = j + 2 \land pos \geq 0 \land pos' = 1).\end{aligned}$$

As a next step, we abstract ϕ_{A+BC} into the formula w':

$$w' \overset{\text{def}}{=} (i + 1 \leq i' \wedge i' \leq i + 2 \wedge j + 1 \leq j' \wedge j' \leq j + 2 \wedge pos' = 1).$$

Notice that w' does not imply that i and j increase by the same amount. The analysis continues by extending $(1 \oplus \phi_C)$ first with $(w')^{\circledast}$ and then with $(1 \oplus \phi_C)$. From the resulting formula, together with the fact that $i = 0 \wedge j = 0$ holds before the loop, after the loop one is able to establish only that $i \leq 2 * j$ and $j \leq 2 * i$. In particular, we can no longer draw the conclusion that $i = j$ holds, which the evaluation of the "cruder" regular expression R_1 succeeded in establishing.

This example demonstrates that an arbitrary—even plausible—refinement of a path expression can actually *degrade* analysis results, compared with using the original path expression. In general, degradation arises because of an interaction between the expressiveness of the abstract domain in use, and the refinement step. In the example above, the common increments to i and j in the three original disjuncts are lost because in $A + BC$, A involves one loop iteration, whereas BC involves two iterations. To retain the information that the increments to i and j are "synchronized," the formula w' that abstracts ϕ_{A+BC} would need to have either the subformula $j' - i' = j - i$ or the subformula $\exists k.i' = i + k \wedge j' = j + k$. The first subformula uses four variables, and each conjunct of the second subformula uses three variables; however, the octagon domain can only express conjunctions of two-variable inequalities.

2.3 Transformation in the Presence of Nested Loops

In the preceding examples, we applied transformations to regular expressions that do not contain nested stars (i.e., leaf loops); these expressions were derived from programs that do not contain nested loops. In general, however, we also want to analyze programs that do contain nested loops. We approach such problems by computing a regular expression that has nested stars. Beginning with an innermost occurrence of a star operator—i.e., a subexpression of the form R^*, where R is star-free)—we apply our transformation, compute a summary, and substitute the summary in place of the innermost star. When this process has been performed for all innermost occurrences of star in the body of a non-innermost occurrence of star, we can analyze the body of the latter occurrence as if the body were star-free. We repeat this process until we have summarized the entire procedure.

Example 2.1. Consider again the program in Fig. 1(a). As explained earlier in this section, the paths in the loop body can be refined from $R \overset{\text{def}}{=} (A + B)^*$ to $R' \overset{\text{def}}{=} A^*B^*$. Suppose that the loop had been nested inside an enclosing loop, and that the path expression for the loop-nest had the form $E \overset{\text{def}}{=} (\alpha((\beta + \gamma)\underbrace{(A + B)^*}_{R}(\delta + \nu))^*\mu)$. Then the refinement of R would create $E_1 \overset{\text{def}}{=} (\alpha \underbrace{((\beta + \gamma)\overbrace{A^*B^*}^{R'}(\delta + \nu))^*}_{S} \mu)$. The leaf-loop algorithm would then be applied to $S \overset{\text{def}}{=} ((\beta + \gamma)A^*B^*(\delta + \nu))^*$, with A^*B^* treated as an indivisible atom. This approach is tantamount to transforming S into "**let** $c = A^*B^*$ **in** $\underbrace{((\beta + \gamma)c(\delta + \nu))^*}_{T}$," and applying the leaf-loop algorithm to $T \overset{\text{def}}{=} ((\beta + \gamma)c(\delta + \nu))^*$. □

As shown in the earlier examples of this section, the leaf-loop algorithm requires us to have some way to check for and remove infeasible paths. The summarization of inner loops gives us a way to perform infeasibility checks even in the presence of nested loops, by using (the interpretations of) the summary values in place of starred subexpressions. For instance, in Ex. 2.1, we could multiply out $(\beta + \gamma)c(\delta + \nu)$ to obtain the sum of products $\beta c\delta + \beta c\nu + \gamma c\delta + \gamma c\nu$, and then test the abstract values of all words in $\{\beta c\delta, \beta c\nu, \gamma c\delta, \gamma c\nu\} \cdot \{\beta c\delta, \beta c\nu, \gamma c\delta, \gamma c\nu\}$ for pair-wise infeasibility.

3 BACKGROUND

3.1 Regular Languages and Algebra

Definition 3.1. The set of **regular expressions** over an alphabet Σ is defined by the following grammar:

$$regexp_\Sigma ::= 0 \mid 1 \mid a \in \Sigma \mid regexp_\Sigma + regexp_\Sigma \mid regexp_\Sigma \cdot regexp_\Sigma \mid regexp_\Sigma^*$$

(Because Σ will always be implicit from context, the subscript on *regexp* will henceforth be omitted.) For simplicity, we sometimes write "$a \cdot b$" as "ab." The **star height** of a regular expression $R \in regexp$ is the maximum nesting depth of a star in R.

A regular expression $R \in regexp$ denotes a **regular language**, $L(R)$, defined as follows:

$$L(R) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } R = 0 \\ \{\epsilon\} & \text{if } R = 1 \\ \{a\} & \text{if } R = a \in \Sigma \\ L(R_1) \cup L(R_2) & \text{if } R = R_1 + R_2 \\ L(R_1) \otimes L(R_2) & \text{if } R = R_1 \cdot R_2 \\ \bigcup_{i=0}^{\infty} L(R_1)^i & \text{if } R = (R_1)^* \end{cases}$$

where ϵ denotes the empty word (of length 0), $\otimes$ denotes language concatenation: $L_1 \otimes L_2 \overset{\text{def}}{=} \{xy \mid x \in L_1 \wedge y \in L_2\}$, and $L(R_1)^i$ denotes the i-fold concatenation of $L(R_1)$ with itself (e.g., $L(a^*) = \emptyset + \{a\} + \{aa\} + \{aaa\} + \ldots$).

We now give the axiomatizations of **Kleene algebras** and a family of strictly weaker structures, which we call **pre-Kleene algebras**.

Both of these structures have "combine," "extend," and "iteration" operators. The following definition for a Kleene algebra was given by Kozen [1994].

Definition 3.2. A **Kleene algebra (KA)**, $\langle A, +, \cdot, *, 0, 1\rangle$, is a set A (called the *carrier*) equipped with two binary operations $\cdot$ and $+$, a unary operation $*$, and distinguished elements 0 and 1, and such that the following conditions hold.

(1) (*Semiring*) The binary operators $\cdot$ and $+$ satisfy the following axioms, which define a semiring
  (a) (Associativity of $\cdot$ and $+$) $a + (b + c) = (a + b) + c$ and $a(bc) = (ab)c$, for all $a, b, c \in A$
  (b) (Commutativity of $+$) $a + b = b + a$, for all $a, b \in A$
  (c) (Distributivity) $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$, for all $a, b, c \in A$
  (d) (Identity elements) There exists an element $0 \in A$ such that, for all $a \in A$, $a + 0 = 0 + a = a$. There exists an element $1 \in A$ such that, for all $a \in A$, $a1 = 1a = a$
  (e) (Annihilation) $a0 = 0a = 0$, for all $a \in A$
(2) (*Idempotence*) It is also required that $+$ be idempotent: $a + a = a$, for all $a \in A$
(3) (*Iteration*) The above axioms imply that relation $\leq$ defined by $a \leq b \Leftrightarrow a + b = b$ is a partial order. Using this partial order, the $*$ operator satisfies the following axioms
  (a) $1 + a(a^*) \leq a^*$, for all $a \in A$
  (b) $1 + (a^*)a \leq a^*$, for all $a \in A$
  (c) If $a, x \in A$ with $ax \leq x$, then $a^*x \leq x$

(d) If $a, x \in A$ with $xa \leq x$, then $xa^* \leq x$

Two important examples of Kleene algebras are (1) Reg_Σ—the algebra of regular languages over an alphabet Σ—where the carrier is the set of regular languages, $+$ is union, $\cdot$ is concatenation, $*$ is Kleene-star, 0 is the empty language, and 1 is the singleton language containing the empty word; and (2) Rel_X—the algebra of relations over a set X—where the carrier is the set of binary relations on X, $+$ is union, $\cdot$ is relational composition, $*$ is reflexive transitive closure, 0 is the empty relation, and 1 is the identity relation.

Definition 3.3. A **pre-Kleene algebra (PKA)**, $\langle A, +, \cdot, *, 0, 1\rangle$, is a set A equipped with two binary operations $\cdot$ and $+$, a unary operation $*$, and distinguished elements 0 and 1, and such that the *Semiring* and *Idempotence* axioms hold, and additionally we have the following iteration axioms:
(1) *(Reflexivity)* $1 \leq a^*$, for all $a \in A$
(2) *(Extensivity)* $a \leq a^*$, for all $a \in A$
(3) *(Transitivity)* $a^* \cdot a^* = a^*$, for all $a \in A$
(4) *(Monotonicity)* if $a, b \in A$ with $a \leq b$ then $a^* \leq b^*$. Equivalently $a^* \leq (a+b)^*$, for all $a, b \in A$
(5) *(Unrolling)* $(a^n)^* \leq a^*$ for $n \in \mathbb{N}$ and $a \in A$, where $a^0 = 1$ and $a^n = \underbrace{a \cdot \ldots \cdot a}_{n \text{ times}}$

The abstract domain from §2 is an example of a pre-Kleene algebra. Two more examples follow. All three examples are pre-Kleene algebras, but none are Kleene algebras.

Example 3.4. Fix a set of variables $\mathbf{x}$ and a set of primed copies $\mathbf{x}'$. Define an algebraic structure $\mathsf{ACI} = \langle \mathcal{F}, \oplus, \otimes, \circledast, 0, 1\rangle$, inspired by the analysis of Ancourt et al. [2010], as follows:

- The carrier F consists of all formulas in the existential fragment of Presburger arithmetic over the symbols $\boldsymbol{x}'$ and $\boldsymbol{x}'$ (i.e., transition formulas), quotiented by logical equivalence.
- $\phi \otimes \psi \overset{\text{def}}{=} \exists \boldsymbol{x}''.\phi(\boldsymbol{x}, \boldsymbol{x}'') \land \psi(\boldsymbol{x}'', \boldsymbol{x}')$ is relational composition.
- $\phi \oplus \psi \overset{\text{def}}{=} \phi \lor \psi$ is disjunction.
- $0 \overset{\text{def}}{=} \mathit{false}$.
- $1 \overset{\text{def}}{=} \bigwedge_i x_i' = x_i$.
- $\phi^\circledast$ is defined as follows.
  - Let $pre(\phi)$ be the convex hull of the formula $(\exists \boldsymbol{x}'.\phi)$, which can be computed using the algorithm of Farzan and Kincaid [2015]. The formula $pre(\phi)$ is a precondition of the loop ϕ, and must hold on entry.
  - Similarly, let $post(\phi)$ be the convex hull of the formula $(\exists \boldsymbol{x}.\phi)$.
  - For each variable x_i, introduce a new variable δ_i, which we use to represent the change in x_i over the action of ϕ. Compute the convex hull of the formula $\exists \boldsymbol{x}, \boldsymbol{x}'.(\phi \land \bigwedge_i \delta_i = x_i' - x_i)$, and write it as $A\boldsymbol{\delta} \geq \boldsymbol{b}$. Let $\Delta(\phi)$ be the formula $\Delta(\phi) \overset{\text{def}}{=} A\boldsymbol{x}' \geq \boldsymbol{x} + k\boldsymbol{b}$
  - Finally, define $\phi^\circledast \overset{\text{def}}{=} \exists k. \big((k = 0 \land \bigwedge_i x_i' = x_i) \lor (k \geq 1 \land pre(\phi) \land post(\phi))) \land \Delta(\phi)\big)$.

Showing that ACI satisfies the semiring axioms, idempotence of +, and reflexivity, extensivity, and transitivity of $\circledast$ is straightforward. It is also not difficult to see that ACI *fails* to satisfy the KA axioms for iteration, because it fails to compute exact transitive closure.

The pre-Kleene algebra monotonicity and unrolling axioms are more subtle. First, observe that the natural order $\leq$ in ACI is entailment ($\phi \oplus \psi = \psi$ if and only if $\phi \models \psi$). So the monotonicity law is that if $\phi \models \psi$, then $\phi^\circledast \models \psi^\circledast$. This holds because the iteration operator is defined in terms of convex hulls of projections, and both projection and hull are monotone. To show that the unrolling axiom holds, let n be arbitrary. Clearly we have $pre(\phi^n) \models pre(\phi)$ and $post(\phi^n) \models post(\phi)$; it remains only to show that $B\boldsymbol{x}' \geq \boldsymbol{x} + \boldsymbol{c} \models A\boldsymbol{x}' \geq \boldsymbol{x} + n\boldsymbol{b}$, where $\Delta(\phi) = A\boldsymbol{x}' \geq \boldsymbol{x} + \boldsymbol{b}$ and $\Delta(\psi) = B\boldsymbol{x}' \geq \boldsymbol{x} + \boldsymbol{c}$.

Since $\phi \models A\boldsymbol{x}' \geq \boldsymbol{x} + \boldsymbol{b}$, we must have $\phi^n \models A\boldsymbol{x}' \geq \boldsymbol{x} + n\boldsymbol{b}$; since $\Delta(\phi^n)$ entails every inequation of the form $\boldsymbol{bx}' \geq \boldsymbol{bx} + \boldsymbol{c}$ that is entailed by ϕ^n, we have the result.

Example 3.5. Fix a set of variables $\boldsymbol{x}$ and a set of primed copies $\boldsymbol{x}'$. Let P be a finite set of predicate symbols in some theory $\mathcal{T}$ with a decidable existential fragment (e.g., the theory of bitvectors, arrays, etc.) over the variables $\boldsymbol{x}$. Define an algebraic structure $\mathsf{PA}_P = \langle \mathcal{F}, \oplus, \otimes, \circledast, 0, 1\rangle$, as follows:

- $\mathcal{F}$ is the set of existential $\mathcal{T}$-formulas over $\boldsymbol{x}$ and $\boldsymbol{x}'$
- $\oplus$, $\otimes$, 0, and 1 are as in Ex. 3.4
- $\phi^{\circledast} \overset{\text{def}}{=} \bigwedge\{p(\boldsymbol{x}) \Rightarrow p(\boldsymbol{x}') : p \in P : \phi \models p(\boldsymbol{x}) \Rightarrow p(\boldsymbol{x}')\}$

It is straightforward to check that PA_P forms a pre-Kleene algebra.

We are interested in determining which facts are true in Kleene algebras and pre-Kleene algebras. However, we are also interested in viewing expressions over $\cdot$, $+$, and $*$ as syntactic objects, because—as we see from the examples in §2—the structure of expressions matters for analysis. To make a notational distinction, we use

$$\mathcal{E} \models_{KA} R = R'$$

to denote that $R = R'$ holds in any Kleene algebra that satisfies each equation in a set $\mathcal{E}$. We use similar notation for pre-Kleene algebras. For example, for both Kleene algebras and pre-Kleene algebras, we have $a + (b + c) = (a + b) + c$. Because $a + (b + c)$ and $(a + b) + c$ are syntactically different, we no longer write $a + (b + c) = (a + b) + c$. Instead we write $\models_{KA} a + (b + c) = (a + b) + c$, and $\models_{PKA} a + (b + c) = (a + b) + c$. Note that $\models_{KA} R = R'$ if and only if $L(R) = L(R')$ [Kozen 1994].

The axioms of pre-Kleene algebras are implied by the axioms of Kleene algebras, so every Kleene algebra is also a pre-Kleene algebra; however, the converse is not true. For example,

$$1 + aa^* = a^* \tag{2}$$

is a property that holds in any Kleene algebra, but does not necessarily hold in a pre-Kleene algebra.

The motivation for introducing pre-Kleene algebras is to capture a broader class of program analyses, for which the axioms of Kleene algebras are too strong. In particular, the Kleene algebra iteration axioms imply that for any element a, a^* must be equal to the least fixed-point of the function $f_a(x) = 1 + ax$. For many abstract domains of interest, least fixed-points of such functions are not computable (and may not even exist). The weaker iteration axioms for pre-Kleene algebras allow us to study domains in which the $*$ operator is imprecise (i.e., a^* is a post fixed-point of f_a).

Naturally, one may ask if the iteration axioms for pre-Kleene algebras are *still* too strong. We note that there is general strategy (see Kincaid [2018]) for designing operators that over-approximate the transitive closure of transition formulas, which always results in an iteration operator satisfying the pre-Kleene algebra iteration axioms. Namely, if we compute $R^{\circledast}$ by (i) computing the best abstraction of R within some class of transition relations for which transitive closure is computable, and (ii) computing the exact transitive closure of that abstraction, then $R^{\circledast}$ satisfies the pre-Kleene algebra axioms.

The gap between Eqn. (2), which holds in KA, and the weaker property "$1 + aa^* \sqsupseteq a^*$," which holds in PKA, provides evidence that PKA captures a fundamental property of program analysis vis a vis program transformations: the analysis of an unrolled loop ("$1 + aa^*$") can give more precise results than the original loop ("a^*"). In contrast, the KA axioms do not allow algebraic reasoning about such possible improvements because they treat an unrolled loop ("$1 + aa^*$") as being equivalent to the original loop ("a^*").

The axiomatization of pre-Kleene algebras is relaxed enough to capture some interesting domains, such as the three introduced above, while being expressive enough to allow for non-trivial refinements.

3.2 Algebraic Program Analysis

We directly connect our interest in regular expressions and program analysis by considering static analyses in the style of Tarjan [1981b], sometimes referred to as *algebraic program analysis*. An algebraic analysis stands in contrast to the classic iterative style of program analysis; it takes an algebraic (rather than order-theoretic) approach to approximating repetitive behavior. The essential difference is that fixed-point computation is *external* to the abstract domain in the classic style and *internal* to the abstract domain in the algebraic approach. In other words, an algebraic analysis does not directly implement a fixed-point computation, instead it assumes that the domain is equipped with some internal method for approximating iteration. The need to supply an extra operation imposes an additional burden on the domain designer; on the other hand, the domain designer is then free to implement their own method for approximating iterative behavior, which could be some direct computation or some standard iterative fixed-point-finding process. By having an explicit iteration operator, the operators of an algebraic analysis have a direct correspondence with the three standard regular-expression operators. Therefore, a static-analysis task can be performed by reinterpreting regular expressions as domain transformers, and evaluating a set of path expressions for a program in a bottom-up manner.

Let Σ be some alphabet. Suppose that we have a structure with the signature $\mathsf{K} = \langle K, \oplus, \otimes, \circledast, 0, 1\rangle$, and a function $\mathsf{K}[\![\cdot]\!] : \Sigma \to K$. We extend $\mathsf{K}[\![\cdot]\!]$ in a syntax-directed way to interpret regular expressions over Σ in K:

$$\mathsf{K}[\![R_1 + R_2]\!] = \mathsf{K}[\![R_1]\!] \oplus \mathsf{K}[\![R_2]\!] \qquad \mathsf{K}[\![R_1 \cdot R_2]\!] = \mathsf{K}[\![R_1]\!] \otimes \mathsf{K}[\![R_2]\!] \qquad \mathsf{K}[\![R^*]\!] = (\mathsf{K}[\![R]\!])^{\circledast}$$
$$\mathsf{K}[\![0]\!] = 0 \qquad \mathsf{K}[\![1]\!] = 1$$

Note that we treat regular expressions as syntactic objects; $L(R_1) = L(R_2)$ does not necessarily imply that $\mathsf{K}[\![R_1]\!] = \mathsf{K}[\![R_2]\!]$.

Let $\mathsf{TR} = \langle TR, \cup, \circ, *, \emptyset, id\rangle$ be the *Kleene* algebra of transition relations. In this case, $\cup$ is relational union, $\circ$ is relational composition, and $*$ is reflexive transitive closure. The additive identity for TR is the empty relation $\emptyset$, and the multiplicative identity is the identity relation id.

We use a transition relation to represent the input/output relationship of each member of some alphabet of actions Σ. We assume TR has a corresponding semantic function, $\mathsf{TR}[\![\cdot]\!] : \Sigma \to TR$, which is extended to regular expressions over Σ in the manner described above. If we take Σ to be the set of program actions, then $\mathsf{TR}[\![R]\!]$ denotes the input/output relationship of the program paths described by R. In general, it is uncomputable to determine the exact value of $\mathsf{TR}[\![R]\!]$. Thus, we use an abstract domain D to approximate TR.

We consider an algebraic analysis to consist of an abstract domain $\mathsf{D} = \langle D, \oplus, \otimes, \circledast, 0, 1\rangle$, a semantic function $\mathsf{D}[\![\cdot]\!] : \Sigma \to D$, and a concretization function $\gamma : D \to TR$. We assume that there are effective procedures for the evaluation of the operators in D. The concretization function γ defines which abstract elements of D over-approximate which elements of TR. That is, $c \subseteq \gamma(a)$ indicates that $a \in D$ over-approximates $c \in TR$.

Definition 3.6. We consider a (sound) algebraic program analysis over an alphabet Σ to consist of a triple $(D, D[\![\cdot]\!], \gamma)$, where
(1) $\gamma(0) = \emptyset$
(2) $\gamma(1) \supseteq id$
(3) $\mathsf{TR}[\![A]\!] \subseteq \gamma(\mathsf{D}[\![A]\!])$, for all $A \in \Sigma$
(4) $\gamma(a_1) \cup \gamma(a_2) \subseteq \gamma(a_1 \oplus a_2)$; $\gamma(a_2) \circ \gamma(a_1) \subseteq \gamma(a_1 \otimes a_2)$; $\gamma(a)^* \subseteq \gamma(a^{\circledast})$ for all $a_1, a_2, a \in D$.

Defn. 3.6 gives a correspondence between the operators of TR and the operators of D, and allows a sound result to be computed in a bottom-up manner:

LEMMA 3.7. *Let $\langle D, D[\![\cdot]\!]\rangle$ be a sound interpretation over an alphabet Σ. Then for any $R \in regexp_{\Sigma}$, we have $TR[\![R]\!] \subseteq \gamma(D[\![R]\!])$.*

The example analysis in §2 gives an example of an algebraic analysis. For that analysis we have that D is the set of two-vocabulary transition formulas in Presburger arithmetic. The sequence operator, $\otimes$, is composition as defined in §2; the combine operator, $\oplus$, is logical or; and the iteration operator, $\circledast$, abstracts a formula to an octagon, and then directly computes the closure of the octagon.

Many standard program-analysis problems can be formulated as an algebraic analysis. In particular, any summary-based interprocedural analysis [Sharir and Pnueli 1981] can be cast as an algebraic analysis, including predicate abstraction [Ball and Rajamani 2001]; affine-relation analysis [Elder et al. 2014; King and Søndergaard 2010; Müller-Olm and Seidl 2004, 2007]; problems in the IFDS [Reps et al. 1995], IDE [Sagiv et al. 1996], and Weighted Pushdown System [Reps et al. 2005] frameworks; and polyhedral analysis [Cousot and Halbwachs 1978; Jeannet and Serwe 2004]. All of these examples are KAs, except for polyhedral analysis and one of the variants of affine-relation analysis [Elder et al. 2014; King and Søndergaard 2010], for which distributivity of $\cdot$ over $+$ fails to hold.

3.3 Rewriting Regular Expressions

Consider an algebraic analysis, with abstract domain D, where the goal is to approximate $\mathsf{TR}[\![R]\!]$ for some set of program paths, $L(R)$. As described in the previous section, an over-approximation can be obtained merely by evaluating R over D: $\mathsf{D}[\![R]\!]$. However, the structure of R may lead to an imprecise analysis result: perhaps there is another suitable expression R' that leads to a better analysis result. If such an R' exists, we would like to evaluate R' instead of R.

We must consider when it is "suitable" to analyze R' instead of R. Namely, if we are going to evaluate $\mathsf{D}[\![R']\!]$ instead of $\mathsf{D}[\![R]\!]$, we need to make sure that $\mathsf{D}[\![R']\!]$ is still a sound analysis.

Definition 3.8. A rewrite, $R \longrightarrow R'$, is *sound* if $\mathsf{TR}[\![R]\!] \subseteq \mathsf{TR}[\![R']\!]$.

This definition says that a rewrite is sound if the semantics of the new path-expression approximates the semantics of the old path-expression. This definition ensures that the analysis of the new regular expression still approximates the original task because, by Defn. 3.6, $\mathsf{TR}[\![R']\!] \subseteq \gamma(\mathsf{D}[\![R']\!])$. Thus, if $\mathsf{TR}[\![R]\!] \subseteq \mathsf{TR}[\![R']\!]$, then $\mathsf{TR}[\![R]\!] \subseteq \gamma(\mathsf{D}[\![R']\!])$. Ideally, we would like to have a regular expression that leads to as precise an analysis result as possible; the challenge would be to find a most-precise regular expression among the set of all sound rewrites—that is, "most precise" in terms of $\gamma(\mathsf{D}[\![R]\!])$ and $\subseteq$. Unfortunately, such an expression may not exist (and even if it does, it can only be computed under strong assumptions about the abstract domain). The main problem is that the set of considered rewrites is too large: there is an infinite space of possible rewrites, so how do we select one?

One approach for selecting a sound rewrite is based on removing infeasible words from R. Suppose that we have discovered some set of infeasible words Inf. That is, for every $w \in Inf$ we have $\mathsf{TR}[\![w]\!] = \emptyset$. Consider creating the following regular expression: $R_{inf} = \sum_{w \in Inf} w$. We can now create the rewrite $R \longrightarrow R'$, by computing a regular expression R' with $L(R') = L(R) \cap L(R_{inf})^c$. The rewrite is sound, because $L(R) = L(R') \cup L(R_{inf})$ implies $\mathsf{TR}[\![R]\!] = \mathsf{TR}[\![R']\!] \cup \mathsf{TR}[\![R_{inf}]\!] = \mathsf{TR}[\![R']\!]$. Written another way, R' is a sound rewrite because $\mathcal{E} \models_{KA} R' = R$, where $\mathcal{E}$ is the set of equations $\mathcal{E} = \{w = \emptyset \mid w \in Inf\}$. We call this type of rewrite a *refinement*.

Definition 3.9. Let $\mathcal{E}$ be some set of equations denoting infeasible paths. A rewrite, $R \longrightarrow R'$, is a *refinement* if $\mathcal{E} \models_{KA} R' = R$ and $L(R') \subseteq L(R)$.

Refinements provide a natural method for producing alternative analysis tasks, which in practice often give better analysis results. However, as shown in the analysis of Fig. 1(b) an arbitrary refinement can actually produce *worse* results. One could imagine rectifying this issue by creating an analysis framework that refines a given analysis task R to a series of refined regular expressions $R_1, \ldots, R_k$, using methods in the style of previous work on control-flow refinement [Balakrishnan et al. 2009; Flores-Montoya and Hähnle 2014; Gulwani et al. 2009; Sharma et al. 2011]. The analysis would then analyze each expression R_i, as well as the original expression R, and then combine the results. We do not find this approach satisfactory, because:

(1) It blows up the number of analysis tasks.
(2) We do not assume our domains have a "meet" operator, so there might not be a meaningful method to combine the results.
(3) Arbitrary refinements do not give us any analytical understanding of what makes one regular expression better than alternative expressions—for the purpose of analysis.

Unfortunately, without any structure on γ or domain D, we cannot understand the analysis precision obtained using a given regular expression, just from the regular expression itself. We remedy this situation by requiring D to have some algebraic properties that turn out to give some traction on the analysis-precision question.

We assume that each abstract domain D satisfies the axioms of a **pre-Kleene algebra** (PKA).

In essence, we use the concept of a PKA to axiomatize the precision properties of abstract domains. Under our domain assumption, we have

$$\text{if } \models_{PKA} R \leq R' \text{ then } \mathsf{D}[\![R]\!] \leq \mathsf{D}[\![R']\!].$$

We can now use the order $\leq$ of a pre-Kleene algebra to compare the relative analysis precision of regular expressions.

Theorem 3.10. *Let Σ be an alphabet, and let $R, R' \in regexp_\Sigma$ with $\models_{PKA} R' \leq R$. Then for any algebraic program analysis $(D, D[\![\cdot]\!], \gamma)$ over Σ, we have $\gamma(D[\![R']\!]) \subseteq \gamma(D[\![R]\!])$; i.e., γ is monotone with respect to $\leq_{PKA}$.*

Proof. $\models_{PKA} R' \leq R$ is equivalent to saying $\models_{PKA} R' + R = R$, which means $\mathsf{D}[\![R']\!] \oplus \mathsf{D}[\![R]\!] = \mathsf{D}[\![R]\!]$. By Defn. 3.6 $\gamma(\mathsf{D}[\![R']\!]) \cup \gamma(\mathsf{D}[\![R]\!]) \subseteq \gamma(\mathsf{D}[\![R']\!] \oplus \mathsf{D}[\![R]\!]) = \gamma(\mathsf{D}[\![R]\!])$. Therefore, $\gamma(\mathsf{D}[\![R']\!]) \subseteq \gamma(\mathsf{D}[\![R']\!]) \cup \gamma(\mathsf{D}[\![R]\!]) \subseteq \gamma(\mathsf{D}[\![R]\!])$. □

Thm. 3.10 gives us the analytical understanding we have been looking for. Thm. 3.10 says that if $\models_{PKA} R' \leq R$ holds, then the analysis of R' will be at least as precise as—and possibly more precise than—the analysis of R.

In this paper, we use $\leq_{PKA}$ to drive the design of a procedure that, given an initial analysis task R and a set of infeasible paths $\mathcal{E}$, refines R to R' with (i) $\mathcal{E} \models_{KA} R' = R$ and $L(R') \subseteq L(R)$ (refinement, Defn. 3.9) and (ii) $\models_{PKA} R' \leq R$ (potentially precision-improving, Thm. 3.10). (3)

By using a refinement procedure that satisfies the specification above, we can analyze R' instead of R: as long as our domain is a pre-Kleene algebra, we are guaranteed that R' will give no worse results, compared with R.

Note that a procedure that satisfies the above specification is still useful for a domain that is not a pre-Kleene algebra: such a refinement method will produce a *sound* analysis task R' from R whenever $\mathcal{E} \models_{KA} R' = R$, for some infeasible paths $\mathcal{E}$. However, if the domain is not a pre-Kleene algebra, then $\models_{PKA} R' \leq R$ does not imply that $\mathsf{D}[\![R']\!] \leq \mathsf{D}[\![R]\!]$, and the conclusion of Thm. 3.10 does not hold. Even though we lose the guarantee that precision does not degrade with domains that are not pre-Kleene algebras, in practice we find that our refinement procedure improves analysis results (see §6).

4 REFINING INNER LOOPS

In this section, we address the problem of refining an innermost-loop based on infeasible paths: we are given an expression R^* where R is star-free; we would like to refine R^* to R' without having R' decrease analysis precision. We use the order of a pre-Kleene algebra to say when a refinement does not decrease precision. That is, we want a method that refines R^* while still having $\models_{PKA} R' \leq R^*$, because by Thm. 3.10 we know the analysis of R' will be at least as good as the analysis of R^*.

Given R^*, we first take R to a different form. Using the distributivity axioms of a pre-Kleene algebra, we rewrite R into the form $a_1 + a_2 + \ldots + a_k$, for some set of words $A \overset{\text{def}}{=} \{a_1, a_2, \ldots, a_k\}$. Due to the axioms of the operators + and $\cdot$ in a pre-Kleene algebra, $\models_{PKA} R^* = (a_1 + \ldots + a_k)^*$; i.e., taking R^* to $(a_1 + \ldots + a_k)^*$ does not change analysis precision. Once R^* is in this form, it becomes easy to determine an expression of infeasible word sequences, R_{inf}. Because we have $\gamma(0) = \emptyset$, if $\mathsf{D}[\![a_{i,1} \cdot \ldots \cdot a_{i,n}]\!] = 0$ then $\mathsf{TR}[\![a_{i_1} \cdot \ldots \cdot a_{i_n}]\!] = \emptyset$. For instance, we can test all sequences of a small number of elements $\{a_i\} \subseteq A$.

We now consider the problem of refining the loop $(a_1 + \ldots + a_k)^*$ into another expression R' by removing infeasible sequences in $L(R_{inf})$, while maintaining that $\models_{PKA} R' \leq R^*$.

For the remainder of this section, we will consider $\{a_1, a_2, \ldots a_n\}$ to be the alphabet under consideration; we are interested in words over $\{a_1, a_2, \ldots a_n\}$, where each a_i is considered to have no "internal" structure.

4.1 Form for Refinements

Before we present the algorithm for refinement, we first ask the following question. We are going to refine $(a_1 + \ldots + a_k)^*$ to R' with $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. What are the properties that R' must satisfy?

Definition 4.1. We say that a regular expression R over some alphabet S satisfies the *equal-unrolling* property if R has star-height 0 or 1 and for every starred sub-expression $(R_i)^*$ of R, we have that every pair of words w_1 and w_2 in $L(R_i)$, $|w_1| = |w_2|$.

In other words, every word in $L(R_i)$ must have the same length. An alternative way to express Defn. 4.1 is as follows: R satisfies the equal-unrolling property if R can be written in the following form

$$R = \sum_i R'_{i,1}(R_{i,1})^* \cdot \ldots \cdot R'_{i,l}(R_{i,l})^* \tag{4}$$

where each $R'_{i,j}$ is a $*$-free expression over the alphabet S and $R_{i,j} = \sum_{w \in S'} w$, $S' \subseteq (S)^{n_{i,j}}$ for some $n_{i,j}$, and $(S)^n$ denotes the set $\{a_{i,1} \cdot \cdots \cdot a_{i,n} | a_{i,j} \in S\}$.

Theorem 4.2. *$\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$ if and only if R' satisfies the* equal-unrolling *property.*

Proof. ($\Leftarrow$) Suppose that R' satisfies the equal-unrolling property. Then R' can be written in the form shown in Eqn. (4). Consider some $R'_{i,j}$. Since $R'_{i,j}$ is $*$-free, then there exists a longest word for $R'_{i,j}$. Let N be the length of this longest word. By the definition of $\leq$ in a pre-Kleene algebra, we must have

$$\models_{PKA} R'_{i,j} \leq \sum_{i=0}^{N}(a_1 + \ldots + a_k)^i \leq \sum_{i=0}^{N}((a_1 + \ldots + a_k)^i)^* \leq \sum_{i=0}^{N}(a_1 + \ldots + a_k)^* \leq (a_1 + \ldots + a_k)^*$$

With the last three steps due to extensivity (2), unrolling (5), and idempotence (2) respectively. Now consider some $R_{i,j} = \sum_{w \in S'} w$, where $S' \subseteq S^{n_{i,j}}$ for some $n_{i,j}$. By the definition of $\leq$ in a pre-Kleene algebra, $\models_{PKA} R_{i,j} \leq (a_1 + \ldots + a_k)^{n_{i,j}}$. Thus, due to monotonicity of $*$ (4) and unrolling (5), $\models_{PKA} (R_{i,j})^* \leq ((a_1 + \ldots + a_k)^{n_{i,j}})^* \leq (a_1 + \ldots + a_k)^*$. Because we have $\models_{PKA} R'_{i,j} \leq (a_1 + \ldots + a_k)^*$

and $\models_{PKA} (R_{i,j})^* \leq (a_1 + \ldots + a_k)^*$ for every i and j, we can conclude $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$ due to transitivity (3) and idempotence (2).

($\Rightarrow$) [sketch] Suppose that the property $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$ holds. By the completeness of first-order logic, there must exist a proof of this fact, denoted by $\vdash_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. The axioms for the proof system are the axioms of pre-Kleene algebra (in equational form). Other relevant inference rules are the usual ones for first-order logic with equality, including transitivity, reflexivity, and symmetry rules and functional consistency (e.g., $\vdash_{PKA} E_1 = E_1'$ and $\vdash_{PKA} E_2 = E_2'$ then $\vdash_{PKA} E_1 + E_2 = E_1' + E_2'$).

We now prove, using induction on the height of the proof tree for the judgement $\vdash_{PKA} E = E'$, that E satisfies the equal-unrolling property iff E' does, and E and E' have the same star-height. To prove the base case, we consider the pre-Kleene axioms in equational form. Then we simply check that if the left-hand side of each equality axiom satisfies the equal-unrolling property then the right-hand side does as well. For example, due to extensivity (2) we have $\vdash_{PKA} a^* + a = a^*$ and both the left-hand and right-hand sides of the equality either satisfy the equal-unrolling property or both do not, and both sides have the same star-height. Checking the other axioms follows a similar pattern.

Now for the inductive step. The induction hypothesis says that for every sub-proof-tree with root $\vdash_{PKA} F = F'$ in the proof $\vdash_{PKA} E = E'$, if F satisfies the equal-unrolling property iff F' does, and F and F' have the same star-height. Consider the last step of the proof $\vdash_{PKA} E = E'$. Consider the inference rule

$$\frac{\vdash_{PKA} E_1 = E_1' \qquad \vdash_{PKA} E_2 = E_2'}{\vdash_{PKA} E_1 + E_2 = E_1' + E_2'}$$

Now suppose $E_1 + E_2$ satisfies the equal-unrolling property. Then E_1 and E_2 must also satisfy it as well. By the induction hypothesis, it must be the case that E_1' and E_2' also satisfy the equal-unrolling property. Therefore, $E_1' + E_2'$ must also satisfy the equal-unrolling property. Also, E_1 has the same star-height as E_1' and E_2 has the same star-height as E_2'. Therefore, the star-height of $E_1 + E_2$ must be the same as the star-height fo $E_1' + E_2'$. The reasoning for the $\cdot$ inference rule is similar. Finally, suppose that the last step of $\vdash_{PKA} E = E'$ was of the form $\vdash_{PKA} (E_1)^* = (E_1')^*$ under the assumption $\vdash_{PKA} E_1 = E_1'$. Suppose that $(E_1)^*$ satisfies the equal-unrolling property. Because the equal unrolling property only applies to expressions of star-height 0 or 1, E_1 must be star-free and all word lengths be equal. By the induction hypothesis E_1' must also be star-free. It then becomes the case that words in $L(E_1')$ must have the same length. Thus $(E_1')^*$ has star-height 1 and satisfies the equal-unrolling property. What we have shown is that if we have $\vdash_{PKA} E = E'$ and E satisfies the equal-unrolling property then so must E', and E has the same star-height as E'. Our initial assumption of the ($\Rightarrow$) direction was that $\vdash_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. Equivalently, $\vdash_{PKA} (a_1 + \ldots + a_k)^* = (a_1 + \ldots + a_k)^* + R'$. $(a_1 + \ldots + a_k)^*$ satisfies the equal-unrolling property and has star-height 1. Therefore, R' must have star-height 0 or 1 and also satisfy the equal-unrolling property. □

Essentially, Thm. 4.2 says that for *any* R'' with $\models_{PKA} R'' \leq (a_1 + \ldots + a_k)^*$, R'' can be rewritten into a sum-of-products form R', and R' must satisfy two important properties. One, R' must have a star-height no greater than 1. Two, for all the $*$-expressions $R_{i,j}^*$ in R', the bodies of the expressions must be a sum of words over $\{a_1, \ldots, a_k\}$, where each word has the same length. As an example, using Thm. 4.2 we can directly conclude that $\models_{PKA} A^*B^* \leq (A + B)^*$, because A^*B^* has the form shown in Eqn. (4). Also, Thm. 4.2 shows that $\not\models_{PKA} (\epsilon + C)(A + BC)^*(\epsilon + B) \leq (A + B + C)^*$ because A and BC come from unrolling $(A + B + C)$, but they come from unrollings of different lengths.

4.2 Refinement Graphs

Consider the problem of refining the loop $(a_1 + \ldots + a_k)^*$ to another expression R' by removing some infeasible sequences denoted by an expression R_{inf}, while maintaining that $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. Note that it is possible to generate an expression R'' that removes all the infeasible sequences from R_{inf} by finding an R'' such that $L(R'') = L((a_1 + \ldots + a_k)^*) \cap L(R_{inf})^c$. However it might be the case that neither R'' nor any other equivalent expression comes with any analysis-precision guarantees.

Definition 4.3. Suppose that we have an expression $(a_1 + \ldots + a_k)^*$ and an expression R_{inf} denoting infeasible words. Let $G = (V, E, \Lambda)$ be a directed graph with vertices V, edges E, and a labeling function $\Lambda : V \longrightarrow \Sigma$. Let $Paths(G)$ be the set of paths of G. A path is a sequence of vertices $v_1 v_2 \cdots v_n$ where for each $i < n$, $(v_i, v_{i+1}) \in E$. We define $L_{Paths}(G) = \{\Lambda(v_1) \cdots \Lambda(v_n) \mid v_1 \cdots v_n \in Paths(G)\}$. We say that G is a *refinement graph* with respect to R_{inf} if $L_{Paths}(G) = L((a_1 + \ldots + a_k)^*) \cap L(R_{inf})^c$.

Given the appropriate refinement graph, our problem now becomes one of finding an expression R' with $L_{Paths}(G) \subseteq L(R') \subseteq L((a_1 + \ldots + a_k)^*)$ and $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. Because of Thm. 4.2, we only need to consider regular expressions of the form shown in Eqn. (4) to ensure $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. Note that being careful about the $*$'s in R' is the important piece to make sure R' has the form given in Eqn. (4).

Consequently, we now consider the cyclic portions of G, and determine how to represent these cyclic behaviors using an expression R' with a star-height no greater than 1 and an equal number of unrollings under a star. Because the cyclic behavior of G is captured by its strongly-connected components, we can focus on a strongly-connected components S.

Theorem 4.4. *Let S be a non-trivial strongly-connected component. If*
(1) The cycles in S have the same length
(2) There is a vertex common to all the cycles in S
then

There exists an expression of the form shown in Eqn. (4) whose language exactly matches $L_{Paths}(S)$. $(\star)$

Proof. This proof is constructive. Let $\{C_1, \ldots, C_N\}$ be the set of simple cycles of S. Because we have $(1) \wedge (2)$, $|C_i| = n$ for all i, and there exists a vertex, e, common to all the cycles.

Now rotate all the cycles C_i to C_i', where each C_i' starts with e. Let $\langle v_{i,1}, \ldots, v_{i,n} \rangle$ be the sequence of vertices for a cycle C_i'. After rotation, $v_{i,1} = v_{j,1} = e$ for each i, j. Let $a_i = \Lambda(v_i)$ for each i. Let $w_i = a_{i,1} \cdot \ldots \cdot a_{i,n}$. Consider the expression $(\sum_{i=1}^{N} w_i)^*$. Since all the cycles C_i' start and end with the same vertex, we have $L((\sum_{i=1}^{N} w_i)^*) \subseteq L_{Paths}(S)$.

However, there is a difference between $L((\sum_{i=1}^{N} w_i)^*)$ and $L_{Paths}(S)$. For example, suppose that we have one cycle in S with length 2. That is, $C_1' = \langle v_{1,1}, v_{1,2} \rangle$. $a_{1,1}a_{1,2}a_{1,1} \in L_{Paths}(S)$, but $a_{1,1}a_{1,2}a_{1,1} \notin L((a_{1,1}a_{1,2})^*)$. We have captured the cyclic behavior of S with $(\sum_{i=1}^{N} w_i)^*$; however, we need to consider how a path can enter a cycle and how it can leave a cycle.

We can solve this issue by creating a new graph S', where S' has a vertex v_{star} with label $(\sum_{i=1}^{N} w_i)^*$, and creating *head* and *tail* chains in S'. These chains act as an extension of the cyclic behavior of the component, showing how control can enter the cycle and how control can leave the cycle. These heads and tails can be created by creating a head and tail vertex for every vertex in the component, and then chaining the head vertices together according to the paths in the cycle; and similarly for the tail vertices. In short, S' contains the vertex v_{star} that captures the cyclic behavior, as well as head chains going into v_{star}, and tail chains coming from v_{star}.

S' is a DAG, for which $L_{Paths}(S') = L_{Paths}(S)$. Furthermore, since S' is a DAG, an expression R' of the form shown in Eqn. (4) such that $L(R') = L_{Paths}(S')$ can be constructed by applying an existing path-expression algorithm. □

Algorithm 1: SafeRefinement(G)

```
   Data: $G = (V, E, \Lambda)$ is a refinement graph
   /* Create the Extended Condensation DAG $Cond$ */
1  $SCCs \leftarrow$ strongly connected components of $G$;
2  $Head \leftarrow$ Empty Map; $Tail \leftarrow$ Empty Map;
3  $Cond = (V_c, E_c, \Lambda_c) \leftarrow$ Empty graph;  /* Initialize the Extended Condensation DAG */
4  for each SCC of SCCs do
5      $(V_{comp}, E_{comp}, \Lambda_{comp}) \leftarrow ProcccessSCC(SCC, Head, Tail, \Lambda)$;
6      $V_c \leftarrow V_c \cup V_{comp}$; $E_c \leftarrow E_c \cup E_{comp}$; $\Lambda_c \cup \Lambda_{comp}$;
7  for $(u, w) \in E$ do                          /* Connect the sccs together */
8      if $u$ and $w$ are not in the same scc then
9          $E_c \leftarrow E_c \cup (Tail(u), Head(w))$;
10 Add entry and exit vertices to $Cond$ with an edge from entry to every other vertex in $Cond$
     and an edge from every other vertex in $Cond$ to exit;
11 Compute an expression for all paths from entry to exit in $Cond$ using the labeling function
     $\Lambda_c$
```

4.3 Refinement Procedure

In §4.2, we saw that it is possible to exactly capture the language of a refinement graph G with an expression R' of the form shown in Eqn. (4), when for each strongly-connected components S, all the simple cycles of S are the same length and share a common vertex. We use this understanding to develop Alg. 1, which takes in a refinement graph and determines such an R'.

The basic process of Alg. 1 is to identify cyclic behavior, to capture cyclic behavior using one $*$, and then to connect that cyclic behavior together in a new graph. This process is similar to graph condensation. In graph condensation, one vertex is created for each strongly connected component, which can be identified via the algorithm of Tarjan [1972]; then, if there was an edge in the original graph between any vertices in two strongly connected components c_1 and c_2, then an edge is added between the vertices in the condensed graph that represent c_1 and c_2. The resulting condensed graph is a directed acyclic graph (DAG). The main conceptual difference with the standard algorithm and Alg. 1 is that strongly connected components are not necessarily condensed to a single vertex in Alg. 1. However, the basic structure of (i) process strongly connected components, and then (ii) connect the result remains. Because our algorithm follows this process, we call the graph it produces the *extended condensation DAG.*

To construct the extended condensation DAG, Alg. 1 calls Alg. 2 to process each strongly connected component. Alg. 2 performs a few tasks. At the very least, Alg. 2 works to create an over-approximating regular expression that captures the cyclic behavior of the component using a regular expression with $*$-height at most 1. The first few lines of Alg. 2 check to see if the incoming component is trivial or not. If a strongly connected component has a single vertex, then the branch at line 2 is taken. In this case, the algorithm will either return a new vertex with associated labeling a_1, or a_1^*.

If the incoming strongly connected component has more than one vertex, all the simple cycles of the component are found. This task can be done via the algorithm of Johnson [Johnson 1975]. Then a check on the cycles is made. If the cycles have different lengths, or if they do not share a common vertex, then algorithm returns a new vertex with associated labeling $(\sum a_i)^*$, where the a_i's are the labels associated with the vertices of the component.

Algorithm 2: ProcessSCC(SCC, $Head$, $Tail$, Λ_G)

```
   Data: $SCC$ a strongly connected component containing vertices $v_1, \dots, v_n$
1  $(V, E, \Lambda) \leftarrow$ Empty refinement graph;
2  if $SCC$ has a single vertex $v_1$ then
3    if $(v_1, v_1) \notin SCC$ then
4      $v' \leftarrow$ New vertex; $\Lambda(v') \leftarrow \Lambda_G(v_1)$;
5      $Head(v_1) \leftarrow (v')$, $Tail(v_1) \leftarrow (v')$;
6      return $(\{v'\}, \emptyset, \Lambda)$
7    $v' \leftarrow$ New vertex; $\Lambda(v') = (\Lambda(v_1))^*$;
8    $Head(v_1) \leftarrow (v')$, $Tail(v_1) \leftarrow (v')$;
9    return $(\{v'\}, \emptyset, \Lambda)$
10 $Cycles \leftarrow$ all simple cycles of $SCC$ including self-loops;
11 if the cycles of $SCC$ have different lengths or there is no vertex common to all cycles in $Cycles$
     then
12   $v \leftarrow$ New vertex; $\Lambda(v) \leftarrow (\sum_{i=1,\dots,n} \Lambda_G(v_i))^*$;
13   $Head(v_i) \leftarrow (v)$, $Tail(v_i) \leftarrow (v)$, for $i \in 1, \dots, n$;
14   return $(\{v\}, \emptyset, \Lambda)$
15 $e \leftarrow$ a vertex that is common to all the cycles in $Cycles$;
16 Permute all the cycles in $Cycles$ to start with $e$;
17 $v_{star} \leftarrow$ New vertex; $\Lambda(v_{star}) \leftarrow (\sum_{\langle v_{i,1},\dots,v_{i,m}\rangle \in Cycles} (\Lambda_G(v_{i,1}) \cdot \dots \cdot \Lambda_G(v_{i,m})))^*$;
18 $Head(e) \leftarrow v_{star}$; $V \leftarrow V \cup \{v_{star}\}$;
19 for $i = 1, \dots, n$ do                    /* Create head and tail vertices */
20   $t_i \leftarrow$ New vertex; $Tail(v_i) \leftarrow t_i$; $\Lambda(t_i) \leftarrow \Lambda_G(v_i)$;
21   $V \leftarrow V \cup \{t_i\}$;
22   if $v_i \neq e$ then
23     $h_i \leftarrow$ New vertex; $Head(v_i) \leftarrow h_i$; $\Lambda(h_i) \leftarrow \Lambda_G(v_i)$;
24     $V \leftarrow V \cup \{h_i\}$;
25 for $\langle v_{i,1}, \dots, v_{i,m}\rangle \in Cycles$ do     /* Chain together head and tail vertices */
26   for $k = 2, \dots, m-1$ do
27     $E \leftarrow E \cup \{(h_{i,k}, h_{i,k+1})\}$;
28   for $k = 1, \dots, m-1$ do
29     $E \leftarrow E \cup \{(t_{i,k}, t_{i,k+1})\}$;
30   $E \leftarrow E \cup \{(h_{i,m}, v_{star})\} \cup \{(v_{star}, t_{i,1})\}$;
31 return $(V, E, \Lambda)$
```

If the condition at line 11 is false, then, by Thm. 4.4, it is possible to capture the cyclic behavior of the component without adding any additional paths. Alg. 2 follows the construction from Thm. 4.4 in this case.

After the strongly connected components have been processed by Alg. 2, Alg. 1 hooks together the resulting DAGs returned by Alg. 2 into the extended condensation DAG. Because every vertex is in some strongly connected component, perhaps a trivial one, Alg. 2 will consider every vertex in the refinement graph. Therefore, every vertex of the refinement graph will have an entry in both *Head* and *Tail*. With this in mind, Alg. 1 hooks together the DAGs returned by Alg. 2 by looking at each edge (v_i, v_j) in the refinement graph, and checking to see if v_i and v_j belong to different strongly connected components. If v_i and v_j are associated with different components, then Alg. 1 adds the edge $(Tail(v_i), Head(v_j))$ to the extended condensation DAG. Finally, Alg. 1 adds an *entry* and *exit* vertex to the extended condensation DAG, and adds the edges $(entry, u)$ and $(u, exit)$ for every vertex u. The algorithm finishes by determining an expression for all paths from *entry* to *exit*, using the labeling function of the extended condensation DAG. This task can be accomplished by moving the label of each vertex to its incoming edges, and using a traditional path-expression algorithm to capture all paths from entry to exit. The resulting path expression will contain all the paths $L_{Paths}(G)$, where G is the original refinement graph, while satisfying the form of Thm. 4.2.

We now consider the complexity of Alg. 1. If we measure the complexity of Alg. 1 in terms of the size of graph G, the running time is dominated by the time for finding simple cycles in Alg. 2. The best known algorithm for this problem is Johnson's Algorithm, which has complexity $O((|V| + |E|)(c + 1))$, where c is the number of simple cycles in the graph. Johnson notes that c can be exponential in $|V|$. However, we have found in practice that refinement graphs tend to be fairly simple (see §6).

We now demonstrate Alg. 1 by analyzing in detail the loop in Fig. 1(a), with an algebraic analysis that uses our refinement algorithm at the evaluation of a iteration operator. We consider the same abstract domain described at the beginning of §2. We denote the statements and conditions of the program, as well as their semantics in the abstract domain as follows:

$$\begin{array}{rcl@{\qquad}rcl}
s_1 &:=& x = 0 & \mathsf{D}[\![s_1]\!] = \phi_{s_1} &:=& x' = 0 \land y' = y \\
s_2 &:=& y = 50 & \mathsf{D}[\![s_2]\!] = \phi_{s_2} &:=& y' = 50 \land x' = x \\
c_{1,t} &:=& [x < 100] & \mathsf{D}[\![c_{1,t}]\!] = \phi_{c_{1,t}} &:=& x < 100 \land x' = x \land y' = y \\
c_{1,f} &:=& [x \geq 100] & \mathsf{D}[\![c_{1,f}]\!] = \phi_{c_{1,f}} &:=& x \geq 100 \land x' = x \land y' = y \\
s_3 &:=& x = x + 1 & \mathsf{D}[\![s_3]\!] = \phi_{s_3} &:=& x' = x + 1 \land y' = y \\
c_{2,t} &:=& [x > 50] & \mathsf{D}[\![c_{2,t}]\!] = \phi_{c_{2,t}} &:=& x > 50 \land x' = x \land y' = y \\
c_{2,f} &:=& [x \leq 50] & \mathsf{D}[\![c_{2,f}]\!] = \phi_{c_{2,f}} &:=& x \leq 50 \land x' = x \land y' = y \\
s_4 &:=& y = y + 1 & \mathsf{D}[\![s_4]\!] = \phi_{s_4} &:=& y' = y + 1 \land x' = x
\end{array}$$

Suppose that the path-expression that described the set of paths of the procedure is

$$s_1 \cdot s_2 \cdot (c_{1,t} \cdot s_3 \cdot (c_{2,t} \cdot s_4 + c_{2,f}))^* \cdot c_{1,f}.$$

The task of the analysis is then to evaluate $\mathsf{D}[\![s_1 \cdot s_2 \cdot (c_{1,t} \cdot s_3 \cdot (c_{2,t} \cdot s_4 + c_{2,f}))^* \cdot c_{1,f}]\!]$. Consider refining the loop $(c_{1,t} \cdot s_3 \cdot (c_{2,t} \cdot s_4 + c_{2,f}))^*$ using algorithm Alg. 1. First, we take the loop to the form $(a_1 + \ldots + a_k)^*$. We can do this by distributing $\cdot$ through $+$ to achieve the following:

$$(c_{1,t} \cdot s_3 \cdot c_{2,t} \cdot s_4 + c_{1,t} \cdot s_3 \cdot c_{2,f})^*$$

Associate A with $c_{1,t} \cdot s_3 \cdot c_{2,t} \cdot s_4$ and B with $c_{1,t} \cdot s_3 \cdot c_{2,f}$. We then have the goal of refining $(A + B)^*$. We construct a refinement graph for this problem, by computing pair-wise feasibility of the summands of $(A + B)^*$. We find that $\mathsf{D}[\![BA]\!] = 0$, but $\mathsf{D}[\![AA]\!], \mathsf{D}[\![AB]\!], \mathsf{D}[\![BB]\!] \neq 0$. Thus, we construct the refinement graph depicted in Fig. 2(a), where the labeling of the vertices is shown.

Call the vertex with associated label A z_1, and the vertex with associated label B z_2.

The first phase of the algorithm is to build an extended condensation graph (DAG). First, as with traditional graph condensation, the strongly connected components are identified. For the graph in Fig. 2(a) there are two strongly connected components: vertex z_1, with a self-loop; and vertex z_2, also with a self-loop. Thus, Alg. 1 will call Alg. 2 on both of these components.

Consider the call on Alg. 2 with the strongly connected component that consists of vertex z_1. In this case, the strongly connected component has a single vertex. Therefore, in Alg. 2 the first then branch is taken. Alg. 2 then checks to see if z_1 has a self-loop. In the case of the graph in Fig. 2(a), this property is true. Therefore, a new vertex is created and associated with a label A^*. Call this

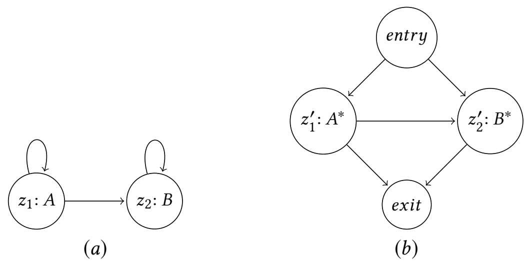

Fig. 2. (a) A refinement graph for the loop from Fig. 1(a), based on pair-wise feasibility. (b) The extended condensation DAG created by Alg. 1 for the graph from (a).

new vertex z_1'. Alg. 2 adds the mapping from z_1 to z_1' in both $Head$ and $Tail$, and then returns the graph containing just z_1'. The same process occurs with z_2.

Control is then returned to Alg. 1. The extended condensation DAG $Cond$ just contains vertices z_1' and z_2' at this point. The final for-loop in Alg. 1 then adds the edge (z_1', z_2') to the graph, because (z_1, z_2) was in the original graph. After these vertices have been connected, the extended condensation DAG has been almost completely constructed. All that remains is to add *entry* and *exit* vertices, with an edge from *entry* to all other vertices, and an edge from all vertices to *exit*. The point of the *entry* and *exit* vertices is to indicate that, in the original expression, it is possible for the loop to start and end on any summand. The resulting DAG is shown in Fig. 2(b). Finally, the algorithm creates a refinement by computing an expression for all paths from *entry* to *exit* over the labels of $Cond$. For our example, the resulting expression is $A^* + B^* + A^*B^*$. It can easily be shown that $\models_{PKA} A^* + B^* + A^*B^* = A^*B^*$, which was the refined expression given in §2.[2]

We now give an example to demonstrate some of the more complicated aspects of Alg. 2. Suppose that we have as input the refinement graph shown in Fig. 3(a). Consider calling Alg. 2 with the non-trivial strongly connected component that consists of the vertices z_1, z_2, and z_3 with respective associated labels a, b, and c. This strongly connected component has more than one vertex, so the first branch is not taken. The algorithm then computes all the simple cycles of the strongly connected component, which can be accomplished by Johnson's algorithm [Johnson 1975]. The simple cycles of this component are $\langle z_1, z_2 \rangle$ and $\langle z_2, z_3 \rangle$. Both of these cycles have length 2, and share the common vertex z_2, which means that this cyclic behavior can be captured by a single $*$, without adding any infeasible paths. That is, the branch at line 11 is not taken. The algorithm then permutes the cycles so that they start with the same vertex. Thus, after line 16, the cycles in $Cycles$ are $\langle z_2, z_1 \rangle$ and $\langle z_2, z_3 \rangle$. The algorithm captures the cyclic behavior of the component by creating a new vertex v_{star} with associated labeling $(ba + bc)^*$. Thus, we have captured the cyclic behavior of this component with the labeling of v_{star}. The remaining problem is that the regular expression $(ba + bc)^*$ says that each path must start with b; however, $a \in L_{Paths}(G)$ but $a \notin L((ba + bc)^*)$. Thus, we have to create heads and tails for this component to indicate how control can get to the cyclic part, and how control can leave the cyclic part. For z_1 and z_3, (i) the head and tail vertices z_1^h and z_1^t are created and given labels that are the same as z_1's label; and (ii) z_3^h and z_3^t are created and given labels that are the same as z_3's label. For z_2, we only create a tail vertex z_2^t, because starting the component with labeling b is captured by the labeling of v_{star}. Alg. 2 also populates the $Head$ and

[2]Actually, the easiest way to obtain A^*B^* is to recognize that the edge from *entry* to z_2' and the edge from z_1' to *exit* are extraneous. Our actual implementation removes such extraneous edges, and thus the extended condensation DAG would just be a chain. However, for this presentation, we do not give the details on how to remove such edges, because precision is not affected in either case.

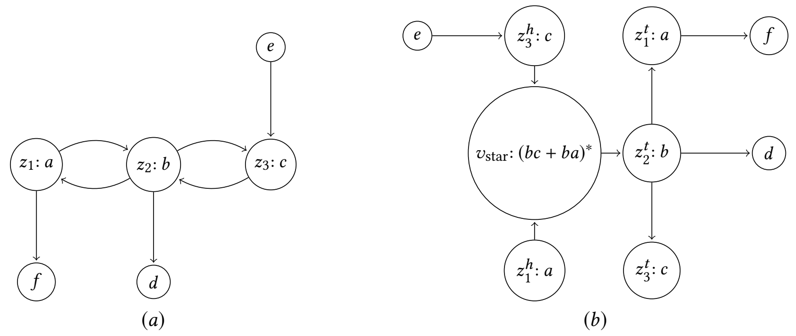

Fig. 3. (a) An example refinement graph. (b) The extended condensation DAG without *entry* and *exit* vertices.

$Tail$ maps as follows:

$$\begin{aligned} Head(z_1) &= z_1^h & Head(z_2) &= v_{star} & Head(z_3) &= z_3^h \\ Tail(z_1) &= z_1^t & Tail(z_2) &= z_2^t & Tail(z_3) &= z_3^t \end{aligned}$$

The algorithm returns control back to Alg. 1. Alg. 1 also processes the other (trivial) strongly connected components of the refinement graph. Once Alg. 1 has processed all the strongly connected components, the next task is to connect the components together. This task is accomplished by connecting a tail vertex to a head vertex whenever such an edge exists in the original graph. For this example, in the original graph there is an edge between the vertex with labeling e and the vertex with labeling c. Consequently, Alg. 1 adds an edge in the extended condensation DAG between the tail vertex with labeling e and the head vertex with labeling c. Alg. 1 repeats this action for all of the edges in the original refinement graph. Thus, excluding *entry* and *exit* vertices, we are left with the extended condensation DAG depicted in Fig. 3(b). To finish this example, Alg. 1 adds *entry* and *exit* vertices, and then computes a regular expression for all paths from *entry* to *exit* over the labels of the resulting graph.

In §2, we observed in the example of the analysis of the program in Fig. 1(b) that a plausible refinement lead to a worse analysis result compared with just evaluating a regular expression that reflected the syntax of the program. We originally had the path expression $(A + B + C)^*$. We now show how our algorithm avoids producing the refinement discussed in §2. Instead, Alg. 1 returns the original path expression $(A + B + C)^*$.

Based on the infeasible paths noted in §2, we would obtain the refinement graph depicted in Fig. 4(a). The only strongly connected component of this graph is the whole graph. Thus, Alg. 1 would call Alg. 2 with the graph depicted in Fig. 4(a). This graph is not a trivial strongly connected component, so Alg. 2 would compute all the simple cycles of the strongly connected component, which are (in terms of labelings) $\langle A \rangle$, $\langle B, C \rangle$, and $\langle A, B, C \rangle$. Note that these cycles do not share a common length. Consequently, the then branch at line 11 is taken, and a new vertex is created with the labeling $(A + B + C)^*$. Control returns to Alg. 1, and the extended condensation DAG depicted in Fig. 4(b) is created. Thus, the final expression created for this example is exactly the same expression it was given as input, namely, $(A + B + C)^*$.

Theorem 4.5. *Let $\mathcal{E}$ be some set of infeasible paths and let R_{inf} be a regular expression such that $L(R_{inf}) = \mathcal{E}$. Now let G be a refinement graph with respect to R_{inf} and $(a_1 + \ldots + a_n)^*$. Alg. 1 will produce a regular expression R' with*
(1) $\mathcal{E} \models_{KA} R' = (a_1 + \ldots + a_n)^*$ and $L(R') \subseteq L(a_1 + \ldots + a_n)^*$
(2) $\models_{PKA} R' \leq (a_1 + \ldots + a_n)^*$

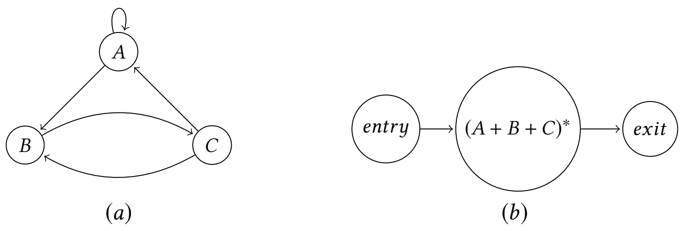

Fig. 4. (a) A refinement graph for the loop from Fig. 1(b), based on pair-wise feasibility. (b) The extended condensation DAG created by Alg. 1 for the graph from (a).

Proof. As for (2), stars are only created in Alg. 2. Every star that is created in Alg. 2 is either of the form $(\sum_{\langle v_{i_1},\ldots,v_{i_m}\rangle \in Cycles}(\Lambda_G(v_{i,1}) \cdot \ldots \cdot \Lambda_G(v_{i,m})))^*$ or $(\sum_{i=1,\ldots,n} \Lambda_G(v_i))^*$. Both of these satisfy the form of interest, and are the only star expressions in R'. Thus, Alg. 1 always produces an expression of the form show in Eqn. (4). That is, Alg. 1 returns an expression R', where R' has star-height no greater than 1, and every summand under a star in R' has the same length. By Thm. 4.2, this property is enough to conclude that $\models_{PKA} R' \leq (a_1 + \ldots + a_n)^*$ holds.

In regards to (1), $\mathcal{E} \models_{KA} R' = (a_1 + \ldots + a_n)^*$ holds because the only difference between R' and $(a_1 + \ldots + a_n)^*$ is the (possible) removal of some of the infeasible paths in $\mathcal{E}$. As for $L(R') \subseteq L(a_1 + \ldots + a_n)^*$, $L((a_1 + \ldots + a_n)^*)$ contains all words over the alphabet $\{a_1 + \ldots + a_n\}$. Alg. 1 creates a regular expression R' over the same alphabet, so trivially $L(R') \subseteq L(a_1 + \ldots + a_n)^*$. □

Thm. 4.5 shows that Alg. 1 meets the goal (3) described in §3.3 for an expression R^* where R is star-free. That is, Alg. 1 is a procedure that can be used to refine a most inner loop, and provides analysis precision guarantees for PKA domains.

5 PUTTING REFINEMENT TO WORK IN A PROGRAM ANALYZER

Refining Arbitrary Regular Expressions. In §4, we gave an approach for refining regular expressions of the restricted form R^*, where R is star-free, to another regular expression R'. We showed in Thm. 4.5 that $L(R') \subseteq L(R^*)$ and $\models_{PKA} R' \leq R^*$. In this section, we show how such a method can be incorporated to rewrite an arbitrary regular expression E to another expression E' with $L(E') \subseteq L(E)$ and $\models_{PKA} E' \leq E$.

The method works bottom-up. Suppose that we have an expression R^* where R is *not* star-free. Alg. 3 first takes R to a "sum-of-products" form. Then Alg. 3 calls itself recursively to refine each of the summands, A_i, of the transformed R to obtain refined expressions A_i'. For each A_i', Alg. 3 associates a new label a_i. Alg. 3 determines infeasible sequences of a_i's by analyzing sequences of A_i''s. Alg. 3 then uses the methods from §4 to refine the expression $(a_1 + \cdots + a_n)^*$ based on the detected infeasible sequences. Finally, Alg. 3 returns the resulting refined expression with each a_i replaced by A_i'. As to the complexity of Alg. 3, we note that line 8 can cause an exponential blow-up in the size of the regular expression when the body of a loop is transformed to a sum-of-products form. In other words, $(A_1 + \cdots + A_n)$ can be exponentially larger than R.

We now prove that Alg. 3 refines an arbitrary expression E and gives back an expression E' with $\models_{PKA} E' \leq E$. First, we give a lemma about pre-Kleene algebras.

Lemma 5.1. *Suppose that* $\models_{PKA} a_1 \leq b_1$ *and* $\models_{PKA} a_2 \leq b_2$ *hold. Then* $\models_{PKA} a_1 + a_2 \leq b_1 + b_2$ *and* $\models_{PKA} a_1 a_2 \leq b_1 b_2$ *hold.*

Algorithm 3: GeneralRefinement(E)

```
Data: $E$ is a regular expression over an alphabet $\Sigma$
if $E$ is a label in $\Sigma$ then
  return $E$
if $E$=$R_1 + R_2$ then
  return GeneralRefinement($R_1$) + GeneralRefinement($R_2$)
if $E$=$R_1 \cdot R_2$ then
  return GeneralRefinement($R_1$)·GeneralRefinement($R_2$)
if $E = R^*$ then
  $(A_1 + \ldots + A_n) \leftarrow$ distribute $\cdot$ through + in $R$ ;
  Associate a new alphabet symbol $a_i$ with each summand $A_i$;
  $A_i' \leftarrow$GeneralRefinement($A_i$);
  $\mathsf{D}[\![a_i]\!] \leftarrow \mathsf{D}[\![A_i']\!]$;
  Identify some set $Seq$ of sequences of $a_i$'s;   /* Candidate infeasible sequences */
  for all sequences $a_{i,1} \cdot \ldots \cdot a_{i,n}$ of $Seq$ do
    if $D[\![a_{i,1} \cdot \ldots \cdot a_{i,n}]\!] = 0$ then
      $Inf \leftarrow Inf \cup a_{i,1} \cdot \ldots \cdot a_{i,n}$
  Build a refinement graph $G$ out of $Inf$ and $a_i$'s;
  $E' \leftarrow$ SafeRefinement($G$);
  return Replace all occurrences of $a_i$ in $E'$ with $A_i'$
```

Proof. $\models_{PKA} a_1 \leq b_1$ means $\models_{PKA} a_1 + b_1 = b_1$. Similar for a_2 and b_2.

$$\models_{PKA} a_1 + a_2 \leq a_1 + a_2 + b_1 + b_2 = b_1 + b_2$$
$$\models_{PKA} a_1 a_2 \leq a_1 a_2 + a_1 b_2 + a_2 b_1 + a_2 b_2 = (a_1 + b_1)(a_2 + b_2) = b_1 b_2$$

□

Theorem 5.2. *Suppose that E is a regular expression, and $E' =$ GeneralRefinement(E). Then*
(1) $L(E') \subseteq L(E)$
(2) $\models_{PKA} E' \leq E$

Proof. The proof is by structural induction on E.

The base case is when E is just a label, in which case the conditions hold trivially.

For the recursive cases, let E'_{sub} = GeneralRefinement(E_{sub}) for each sub-expression E_{sub} of E. The induction hypothesis says that for each sub-expression E_{sub} of E, $L(E'_{sub}) \subseteq L(E_{sub})$ and $\models_{PKA} E'_{sub} \leq E_{sub}$

(1) The cases for $E = R_1 + R_2$ and $E = R_1 \cdot R_2$ follow directly from Lem. 5.1.
(2) If $E = R^*$, Alg. 1 first converts R to have the form $A_1 + \ldots + A_n$, and then refines each A_i into A_i', calling Alg. 3 recursively. It then uses Alg. 1 to build a refined regular expression E' using the labels $\{a_i\}$.

    Note that by Thm. 4.5, $L(E') \subseteq L((a_1 + \ldots + a_n)^*)$ and $\models_{PKA} E' \leq (a_1 + \ldots + a_n)^*$. By Lem. 5.1, as well as the monotonicity axiom of star for a pre-Kleene algebra (Defn. 3.3, item (4)), if we replace each occurrence of a_i in E' with A_i to obtain E_A, we can conclude $L(E_A) \subseteq L((A_1 + \ldots + A_n)^*)$ and $\models_{PKA} E_A \leq (A_1 + \ldots + A_n)^* = E$. Furthermore, because of the induction hypothesis, we have $L(A_i') \subseteq L(A_i)$ and $\models_{PKA} A_i' \leq A_i$. Consequently, if we replace each a_i in E' with A_i' to obtain E_A', we can conclude $L(E_A') \subseteq L(E_A) \subseteq L((A_1 + \ldots + A_n)^*)$ and $\models_{PKA} E_A' \leq E_A \leq (A_1 + \ldots + A_n)^* = E$. The algorithm returns E_A'.

□

Thm. 5.2 shows that for the most part Alg. 3 fits the description of a refinement procedure that meets the goal (3) described in §3.3 for *any* arbitrary expression. The only difference is that Alg. 3 does not refine R based on some *externally* given set of infeasible paths $\mathcal{E}$. Instead Alg. 3 refines sub-expressions based on an *internal* strategy for detecting infeasible path sequences (lines 12–15). The benefits of this design are described below.

Recognizing Infeasible Sub-Paths. The second issue that must be addressed to put our refinement technique to work in a program analyzer is that of recognizing infeasible sub-paths. Lines 12–15 of Alg. 3 constitute a mechanism for identifying a set of infeasible sub-paths, which could be used with different policies. There are a few interesting points to observe about this mechanism.

- It exploits *compositionality*: the values in the sequences $a_{i,1} \cdot \ldots \cdot a_{i,n}$ used in the test $\mathrm{D}[\![a_{i,1} \cdot \ldots \cdot a_{i,n}]\!] = 0$ involve summary values computed for subterms of R (where E has the form R^*). The mechanism applies to non-leaf loops because a summary, in the form of an abstract-domain value, will have been computed for each more-deeply-nested loop contained within R.
- Because the test $\mathrm{D}[\![a_{i,1} \cdot \ldots \cdot a_{i,n}]\!] = 0$ merely involves evaluation in the abstract domain, the test is decidable.
- Because the test is performed using the *same* abstract domain employed everywhere else in the analyzer, there should be a good "impedance match" with the rest of the analyzer. That is, the test will only cause a sub-path to be excluded if the abstract domain has enough fidelity to observe the properties that cause the sub-path to be infeasible.

These properties contrast with the method Sharma et al. [2011] use to identify splitter predicates: their method also works bottom-up, but for non-leaf loops they need to rely on a *separate* method for identifying loop invariants of inner loops.

In our experiments, we used a simple policy of checking all pairs of summands when R is put in sum-of-products form, as discussed in the paragraph just after Ex. 2.1.

6 EXPERIMENTS

Our experiments were designed to answer the questions posed below.

The algorithm given in §4 refines a regular expression R into a refinement R' for which it is guaranteed that the results obtained with R' are *no worse than* those obtained with R. However, in practice, we would also like a refinement to provide *better* answers. One measure of success is whether employing the refinement algorithm improves some "down-stream task" that uses the analysis results.

> EXPERIMENTAL QUESTION 1: Does our refinement algorithm allow an analysis to prove more assertions in practice?

In practice, we often find that the most expensive part of an analysis is the evaluation of the $*$ operators in a regular expression. The refinement procedure works bottom-up, repeatedly taking a regular expression of the form R^*, where R is star-free, and producing another expression R'. At each level, there is only one star operator to evaluate; however, there is no *a priori* bound on the number of star operators in R'. Thus, it could be the case that the refinement procedure increases precision, at the cost of substantially increased analysis time.

> EXPERIMENTAL QUESTION 2: Do our refinements greatly increase analysis time in practice?

We would also like to understand how well our approach performs compared to alternative methods for static program analysis. It is possible that our techniques increase analysis precision in practice, but not enough to make algebraic program analyses competitive with other analysis techniques.

Experimental Question 3: How does the performance of an algebraic program analyzer that uses our refinement techniques compare to that of state-of-the-art model checkers?

6.1 Experimental Setup

We implemented Alg. 3 as an extension of the implementation of Compositional Recurrence Analysis (CRA) [Farzan and Kincaid 2015]. At each star, our implementation checks the feasibility of pairs of labels a_i, a_j to see whether $a_i a_j$ or $a_j a_i$ are infeasible action sequences. We tested the algorithm using two different abstract domains, which we call KCBR [Kincaid et al. 2018] and ACI [Ancourt et al. 2010] (Ex. 3.4). The KCBR domain is more expressive than the ACI domain, but does not satisfy (all of) the axioms of a pre-Kleene algebra (Defn. 3.3). Thus, there is no longer a guarantee that the results from analyzing the refined regular expression are no worse than the results from analyzing the original expression. However, because our refinements are always sound in the sense of Defn. 3.8, the resulting analysis will still be a (sound) over-approximation. The ACI domain always satisfies the axioms, so the refinement algorithm is guaranteed to give analysis results that are at least as good as those obtained without using refinement.

To answer the experimental questions, we ran our implementation(s), as well as the software model checkers Ultimate Automizer [Heizmann et al. 2013] version 0.1.23 and SeaHorn [Gurfinkel et al. 2015] version 0.1.0, on several suites of micro-benchmark programs containing only true assertions. In Tab. 1, we report for each suite (i) the number of programs for which the analyzer was able to prove all assertions, (ii) the total analysis time, and (iii) the number of timeout and out-of-memory exceptions. Timings (with a timeout limit of 300 seconds) were taken on a virtual machine (using Oracle VirtualBox) with 8GB of RAM, with a guest OS of Ubuntu 14.04, host OS of CentOS 6.9, and a 3.2 GHz quad-core Intel Core i5-4570 host CPU. The programs come from the following sources:

- 35 programs are from a suite used to test the resource-bound-analysis tool C4B [Carbonneaux et al. 2015].
- 46 programs are from a suite used to test the invariant-generating tool HOLA [Dillig et al. 2013].
- 96 programs are from the *Integers and Control Flow—Loops* subcategory of SV-COMP16 [SV-COMP16 2016].
- 47 programs are benchmarks containing multi-path loops that we created to test the analyzer.

Answers to the three experimental questions are given in §6.2, §6.3, and §6.4.

6.2 EQ1: Does Refinement Allow More Assertions to be Proven?

In short, the answer is, “Yes, the refinement algorithm allows the analyzer to prove more assertions in practice. Overall, refinement helped both KCBR and ACI prove over 25% more assertions than without refinement.”

There was one instance where the additional overhead of refinement increased the analyzer’s memory usage enough that the analysis could not complete successfully. This example was in the *loops* suite, and is the reason that, for both the KCBR and ACI domains, the number of assertions proved decreased for that suite when using refinement. For all other suites, the refinement algorithm allowed both the KCBR and ACI domains to prove strictly more assertions than when the analysis was performed without refinement.

Table 1. The results of the assertion-checking experiments. Column 2 shows the total number of programs in each benchmark suite. Columns 3-20 show analysis results under six different conditions: with abstract domain KCBR or ACI, each with or without using the refinement algorithm, and using the state-of-the-art model checkers Ultimate Automizer and SeaHorn. For each configuration, the left column indicates the total running time (in seconds), the middle column indicates the number of programs in which all assertions were proven by the analyzer, and the right column is a pair T/M where T is the number of timeouts and M is the number of out-of-memory exceptions. In each row, the smallest running time and the greatest number of assertions proved are shown in boldface.

Benchmark Suite	Total	KCBR			KCBR + refinement			ACI			ACI + refinement			UAutomizer			SeaHorn		
	#P	Time	#P	#E	Time	#P	#E	Time	#P	#E	Time	#P	#E	Time	#P	#E	Time	#P	#E
C4B	35	37.7	19	0/0	50.4	28	0/0	**36.0**	19	0/0	49.5	28	0/0	4118.7	24	11/0	1861.2	**29**	6/0
HOLA	46	62.0	33	0/0	90.1	**42**	0/0	**59.6**	30	0/0	86.5	37	0/0	2467.3	36	7/0	1212.8	38	4/0
SV/loop-accel	19	13.2	13	0/0	17.6	**15**	0/0	**12.6**	12	0/0	17.4	13	0/0	2554.0	11	7/0	907.3	**15**	3/0
SV/loop-invgen	19	32.1	18	0/0	56.6	**19**	0/0	**31.5**	18	0/0	53.2	**19**	0/0	2221.2	12	7/0	474.1	17	1/0
SV/loop-lit	15	12.0	10	0/0	16.1	**13**	0/0	**11.4**	10	0/0	15.8	12	0/0	940.1	**13**	2/0	302.9	**13**	1/0
SV/loop-new	8	6.2	6	0/0	8.1	**7**	0/0	6.1	4	0/0	**6.0**	5	0/0	1469.3	4	4/0	301.8	6	1/0
SV/loops	33	45.7	22	0/0	68.4	21	0/1	**44.3**	22	0/0	68.9	21	0/1	2275.2	**26**	6/0	910.4	**26**	3/0
misc	47	39.4	20	0/0	70.3	36	0/0	38.2	19	0/0	68.0	33	0/0	1909.4	**43**	4/0	**8.7**	**43**	0/0
Total	222	248.3	141	0/0	377.6	181	0/1	**239.7**	134	0/0	365.3	168	0/1	17955.5	169	48/0	5979.2	**187**	19/0

Consider the programs shown in Fig. 5. These programs each contain one assertion. When using either the KCBR or the ACI domain, neither assertion can be proven by the analyzer without using refinement, but both assertions can be proven with refinement.

The program shown in Fig. 5(a) is *leapsum2.c* from the *misc* suite. The loop body in this program has two paths; we will use the label A for the path that takes the then-branch of the conditional statement, and the label B for the path that takes the else-branch. When we check the feasibility of path sequences, we discover that an iteration taking path A can only be followed an iteration taking path B, and vice versa. As a result, we can refine a regular expression of the form $(A + B)^*$ to one of the form $(\epsilon + B)(A \cdot B)^*(\epsilon + A)$. The refined star body $A \cdot B$ can be described by the transition formula $x' = x + 2 \wedge y' = y + 2 \wedge z' = 1 \wedge t' = t + 1$. This formula implies that the difference between x and y after the star $(A \cdot B)^*$ is the same as before the star, which is the key step in proving that the assertion holds. In contrast, without refinement, the analyzer produces a summary for the loop by summarizing its two paths and joining them with a disjunction; the resulting summary is imprecise and does not imply that the difference between x and y remains bounded between 1 and -1.

The program shown in Fig. 5(b) is *maxequals_linear_2.c* from the *misc* suite. This program illustrates a less-obvious application of refinement. There are two paths A and B through the loop body, depending on whether x is assigned the left (A) or the right (B) subexpression of the *max* macro, i.e., depending on whether or not $x > 75 - 10 * t$ holds. The subexpression $75 - 10 * t$ always decreases from one iteration to the next, so any execution of the loop proceeds in two phases: a first phase of at most one iteration in which B occurs, and a second phase in which only A occurs. Therefore, when refinement is used, the analyzer can conclude that x will ultimately have the value 55, which is the value of $75 - 10 * t$ on the first iteration. This property suffices to prove the assertion. Without refinement, the analyzer fails to find any upper bound on x that is implied by the loop body's transition formula, and therefore the analyzer cannot prove the assertion.

6.3 EQ2: Does Refinement Greatly Increase Analysis Time?

In short, the answer is, "The refinement algorithm caused an increase in overall analysis time of about 50%."

```
void main(int N) {
  int x = 1; int y = 2; int z = 1;
  for(int t = 0; t < N; t++) {
    if (z > 0) {
        x = x + 2;
        z = -1;
    } else {
        y = y + 2;
        z = 1;
    }
  }
  assert(x + z == y);
}
```

(a) misc/leapsum2.c

```
#define max(A,B) ((A > B) ? A : B)
int x;
void loop() {
  for(int t = 2; t <= 10; t++) {
    x = max(x, 75 - 10 * t);
  }
}
void main() {
  x = 0;
  loop();
  assert(x == 55);
}
```

(b) misc/maxequals_linear_2.c

Fig. 5. Two examples of programs having assertions that the analyzer can prove only when using the refinement algorithm.

Because both the KCBR and ACI domains are based on transition formulas, the infeasibility check on line 14 of Alg. 3 results in a call to an SMT solver. Experimentally, we found that these feasibility checks are often the main contributor to the increased analysis time when using refinement. As a corollary, we find that while in principle Alg. 1 can be exponential time for certain complicated refinement graphs, in our experience Alg. 1 does not significantly contribute to increased analysis time.

6.4 EQ3: How Does CRA With Refinement Compare with State-of-the-Art Model Checkers?

In terms of assertions proved, we found experimentally that equipping the KCBR and ACI domains with refinement made CRA—with each domain—competitive with Ultimate Automizer and SeaHorn. Restricting the comparison to KCBR versus SeaHorn, we see that refinement allows KCBR to at least tie SeaHorn, in terms of assertions proved, for the *HOLA*, *loop-accel*, and *loop-lit* suites, whereas KCBR without refinement is well behind SeaHorn for these suites. Furthermore, refinement allows KCBR to overtake SeaHorn for the loop-new category and increase its lead in the loop-invgen category.

In terms of total analysis time, CRA with the KCBR and ACI domains (either with or without refinement) completes the test suite in a fraction of the time taken by Ultimate Automizer or SeaHorn. The slowest variant of CRA—based on the KCBR domain with refinement—is more than forty-seven times faster than Ultimate Automizer and fifteen times faster than SeaHorn. However, these speedup ratios are due, in part, to the fact that CRA with KCBR or ACI never timed out on any example, whereas Ultimate Automizer and SeaHorn, which are both based on abstraction refinement, timed out on many examples. Thus, the total times given in Tab. 1 are highly dependent on the chosen timeout value, here 300 seconds. Nevertheless, the figures of >47x and >15x improvement are valid in the sense that we optimistically credit Ultimate Automizer and SeaHorn as having completed in 300 seconds for the examples on which they time out.

If we exclude programs for which Ultimate Automizer timed out, then Ultimate Automizer took 20 seconds on average to analyze a program. Similarly, SeaHorn took 1.4 seconds on average for programs on which it did not time out. These numbers should be compared to 1.08 and 1.7 seconds per program for the fastest and slowest variants of CRA. However, there were example programs for which Ultimate Automizer and SeaHorn took much longer than average. Excluding timeouts, the maximum time that each tool took to analyze a program was 10.1, 12.0, 9.9, 11.5, 261.7, and 168.9

seconds for KCBR, KCBR+refinement, ACI, ACI+refinement, Ultimate Automizer, and SeaHorn, respectively.

7 RELATED WORK

Cousot and Cousot [2002] developed a general theory of semantically justified program transformation based on abstract interpretation. They view a syntactic program as an abstraction of its semantics, and a syntactic transformation as a (conceptual) decompilation of an associated semantic transformation. They develop associated correctness conditions for when a syntactic transformation is an over-approximation of a semantic transformation. Their methodology aims to provide a conceptual framework for proving that, compared to the original program, each program produced by some transformation algorithm has some desired property in the concrete semantics. A similar approach could be used in our context, but we have the slightly different goal of being able to show that, for each transformation $R \longrightarrow R'$, certain desirable properties hold for both the concrete semantics of R and R' and the abstract semantics of R and R'.

Our work represents a framework that can be instantiated with different abstract domains. It comes equipped with a transformation algorithm taking, e.g., R to R', for which (i) the concrete semantics of R' is sound with respect to the concrete semantics of R, and (ii) the abstract semantics of R' yields a value that is sometimes better—and never worse—than the abstract semantics of R. The framework uses two related algebras, Kleene algebra and pre-Kleene algebra, to characterize the concrete and abstract properties on which the transformation algorithm relies. We are then able to use algebraic reasoning to prove properties of the transformation algorithm. In a similar vein, Kot and Kozen [2005] consider an axiomatization that is weaker than Kleene algebra, which they use in an algorithm to compute the closure of a matrix with respect to a cutset of the control-flow graph. This weaker axiomatization is incomparable to the pre-Kleene axiomatization given in this paper. The axiomatization used in Kot and Kozen [2005] does not assume full distributivity laws, which we do. However, they assume the ascending-chain condition, which allows them to consider a more restrictive $*$ than the ones considered in this paper. Kozen [2003] also uses an axiomatization based on Kleene algebra, called Kleene algebra with tests (KAT) in a static-analysis context. This work differs from ours in that Kozen [2003] uses a complete equational theory and careful manual reasoning, while we use an approximate abstract domain and automatic reasoning. At a high level, Kozen [2003] annotates a program's path expression, say R, with a (security) policy to obtain an R'. Then, to prove that R satisfies the policy, Kozen [2003] uses a special theorem prover to show, using our notation, $\mathcal{E} \models_{KAT} R \leq R'$, for some manually chosen set $\mathcal{E}$. We, on the other hand, automatically determine a set of infeasible paths $\mathcal{E}$, and refine our original expression E to an expression E' with $\mathcal{E} \models_{PKA} E' \leq E$.

The idea of applying transformations to a program's IR as a way to improve the results of static analysis has a long history. In some work, transformations are *explicit*, such as the abstraction-refinement method used in SLAM [Ball and Rajamani 2001], approaches based on isolating hot paths [Ammons and Larus 1998; Fisher 1981; Melski 2002], and techniques for rewriting loops [Balakrishnan et al. 2009; Flores-Montoya and Hähnle 2014; Gulwani et al. 2009; Sharma et al. 2011]. In other work, transformations are *implicit*: extra information—typically information about the execution context—is used to *label values* that arise during the course of an analysis. This approach is tantamount to splitting lazily the elements of the IR, where each duplicated IR element is then associated with a single analysis value. The latter idea appears in numerous places, going back at least to the work of Holley and Rosen [1981] on "Qualified data flow problems." Other instances of the idea include the call-strings approach to interprocedural dataflow analysis [Sharir and Pnueli 1981], weighted pushdown systems [Bouajjani et al. 2003; Reps et al. 2005], and trace partitioning [Rival and Mauborgne 2007]. It has been used in such systems as ESP [Das et al. 2002], Archer [Xie et al. 2003], and Saturn [Dillig et al. 2008], and in analysis libraries such as Moped [Schwoon [n. d.]] and WALi [Kidd et al. 2007].

Our work performs an explicit rewrite of the IR used by the analyzer (namely, a regular expression). It was directly inspired by experience with an implementation of the algorithm of Sharma et al. [2011]. While that algorithm works well for some examples, it inspired us to investigate whether an algorithm loop-transformation could provide an "improvement guarantee." While our algorithm does not guarantee *improved* analysis results, it does guarantee *not to produce worse (or incomparable) results* (see Thm. 5.2).

There is a huge literature on loop transformations for the purpose of optimizing a program's execution time: transformations are typically performed on innermost loops, or loop nests. However, that work is more a *client* of an analysis, and focused on reducing the execution time, whereas the focus of our work is on improving analysis precision.

ACKNOWLEDGMENTS

This work was supported in part by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; by ONR under grant N00014-17-1-2889; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from WARF. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

G. Ammons and J.R. Larus. 1998. Improving Data-flow Analysis with Path Profiles. In *PLDI*.

C. Ancourt, F. Coelho, and F. Irigoin. 2010. A Modular Static Analysis Approach to Affine Loop Invariants Detection. *Electron. Notes Theor. Comput. Sci.* 267, 1 (Oct. 2010), 3–16.

G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, and A. Gupta. 2009. Refining the Control Structure of Loops using Static Analysis. In *EMSOFT*.

T. Ball and S.K. Rajamani. 2001. Bebop: A Path-sensitive Interprocedural Dataflow Engine. In *PASTE*.

A. Bouajjani, J. Esparza, and T. Touili. 2003. A Generic Approach to the Static Analysis of Concurrent Programs with Procedures. In *Princ. of Prog. Lang.* 62–73.

M. Bozga, C. Gîrlea, and R. Iosif. 2009. Iterating Octagons. In *TACAS*.

Q. Carbonneaux, J. Hoffmann, and Z. Shao. 2015. Compositional Certified Resource Bounds. In *PLDI*.

P. Cousot and R. Cousot. 2002. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *POPL*.

P. Cousot and N. Halbwachs. 1978. Automatic Discovery of Linear Constraints Among Variables of a Program. In *POPL*.

M. Das, S. Lerner, and M. Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Prog. Lang. Design and Impl.* ACM Press, New York, NY, 57–68.

I. Dillig, T. Dillig, and A. Aiken. 2008. Sound, Complete and Scalable Path-Sensitive Analysis. In *PLDI*.

I. Dillig, T. Dillig, B. Li, and K. McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *OOPSLA*.

M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. 2014. Abstract Domains of Affine Relations. *TOPLAS.* 36, 4 (Jan. 2014).

A. Farzan and Z. Kincaid. 2013. An Algebraic Framework for Compositional Program Analysis. *CoRR (arXiv)* (2013).

A. Farzan and Z. Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*.

J.A. Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Computers* C-30, 7 (1981), 478–490.

A. Flores-Montoya and R. Hähnle. 2014. Resource analysis of complex programs with cost equations. In *APLAS*.

S. Gulwani, S. Jain, and E. Koskinen. 2009. Control-flow Refinement and Progress Invariants for Bound Analysis. In *PLDI*.

A. Gurfinkel, T. Kahsai, A. Komuravelli, and J.A. Navas. 2015. The SeaHorn Verification Framework. In *CAV*.

M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. 2013. Ultimate Automizer with SMTInterpol (Competition Contribution). In *TACAS*.

L.H. Holley and B.K. Rosen. 1981. Qualified Data Flow Problems. *Trans. on Softw. Eng.* 7, 1 (1981), 60–78.

B. Jeannet and W. Serwe. 2004. Abstracting Call-Stacks for Interprocedural Verification of Imperative Programs. In *AMAST*.

D. Johnson. 1975. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* (1975).

N. Kidd, A. Lal, and T. Reps. 2007. WALi: The Weighted Automaton Library. http://www.cs.wisc.edu/wpis/wpds/download.php

Z. Kincaid. 2018. Numerical Invariants via Abstract Machines. In *SAS*.

Z. Kincaid, J. Breck, A. Forouhi Boroujeni, and T. Reps. 2017. Compositional Recurrence Analysis Revisited. In *PLDI*.

Z. Kincaid, J. Cyphert, J. Breck, and T. Reps. 2018. Non-Linear Reasoning for Invariant Synthesis. *PACMPL* 2(POPL) (2018), 54:1–54:33.

A. King and H. Søndergaard. 2010. Automatic Abstraction for Congruences. In *VMCAI*.

L. Kot and D. Kozen. 2005. Kleene Algebra and Bytecode Verification. *Electr. Notes Theor. Comp. Sci.* 141, 1 (2005).

D. Kozen. 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. In *IEEE Sym. on Logic in Comp. Sci.*

D. Kozen. 2003. *Kleene Algebra with Tests and the Static Analysis of Programs.* TR 2003-1915. Dept. of Comp. Sci., Cornell Univ., Ithaca, NY.

Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic Optimization with SMT Solvers. In *Princ. of Prog. Lang.* 607–618.

D.G. Melski. 2002. *Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation.* Ph.D. Dissertation. Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1435.

M. Müller-Olm and H. Seidl. 2004. Precise Interprocedural Analysis through Linear Algebra. In *POPL*.

M. Müller-Olm and H. Seidl. 2007. Analysis of Modular Arithmetic. *TOPLAS.* 29, 5 (2007).

T. Reps, S. Horwitz, and M. Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. 49–61.

T. Reps, S. Schwoon, S. Jha, and D. Melski. 2005. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. *SCP* 58 (2005).

X. Rival and L. Mauborgne. 2007. The Trace Partitioning Abstract Domain. *TOPLAS.* 29, 5 (2007).

M. Sagiv, T. Reps, and S. Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comp. Sci.* 167 (1996), 131–170.

S. Schwoon. [n. d.]. Moped System. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.

Roberto Sebastiani and Silvia Tomasi. 2012. Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ Cost Functions. In *IJCAR*. 484–498.

M. Sharir and A. Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

R. Sharma, I. Dillig, T. Dillig, and A. Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *CAV*.

SVCOMP16 2016. 5th Int. Competition on Software Verification (SV-COMP16). https://sv-comp.sosy-lab.org/2016/

R. Tarjan. 1972. Depth-first Search and Linear Graph Algorithms. *SIAM J. Comput.* (1972).

R.E. Tarjan. 1981a. Fast Algorithms for Solving Path Problems. *J. ACM* 28, 3 (1981), 594–614.

R.E. Tarjan. 1981b. A Unified Approach to Path Problems. *J. ACM* 28, 3 (1981), 577–593.

Y. Xie, A. Chou, and D. Engler. 2003. ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors. In *ESEC/FSE*.