CF-GKAT: Efficient Validation of Control-Flow Transformations

ANONYMOUS AUTHOR(S)

Guarded Kleene Algebra with Tests (GKAT) provides a sound and complete framework to reason about trace equivalence between simple imperative programs. However, there are still several notable limitations of GKAT: First, it is completely agnostic with respect to the meaning of primitives, to keep equivalence decidable. Second, GKAT excludes non-local control flow such as goto, break and return. To overcome these limitations, We introduce *control flow GKAT* (*CF-GKAT*), a system that allows reasoning about programs that include non-local control flow as well as hardcoded values. CF-GKAT is able to soundly and completely verify trace equivalence of a larger class of programs, while preserving the nearly linear efficiency of GKAT. This makes CF-GKAT suitable for the verification of control-flow manipulating procedures, such as decompilation and goto-elimination. To demonstrate CF-GKAT's abilities, we validated the output of several highly non-trivial program transformations, such as Böhm-Jacopini conversion, and Erosa and Hendren's goto-elimination procedure. CF-GKAT opens up the application of Kleene Algebra to a wider set of challenges, and provides an important verification tool that can be applied to the field of decompilation and control-flow transformation.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Logic and verification**; **Algebraic semantics**; • **Software and its engineering → Correctness**.

Additional Key Words and Phrases: Program equivalence, Kleene algebra, control flow recovery

ACM Reference Format:
Anonymous Author(s). 2024. CF-GKAT: Efficient Validation of Control-Flow Transformations. 1, 1 (October 2024), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

There are many notions of program equivalence, each with its own properties in terms of granularity and computational complexity. At one end of the spectrum, syntactic equality compares two programs solely based on its syntax tree. Although decidable, this technique will not equate "intuitively equivalent" programs like the following:

$$\texttt{if } t \texttt{ then } p \texttt{ else } q \qquad\qquad \texttt{if } \neg t \texttt{ then } q \texttt{ else } p$$

Conversely, input-output equivalence relates two programs if and only if they yield the same output when given the same input. This equivalence is very powerful, but also well-known to be undecidable.

Situated between these two extremes, *Guarded Kleene Algebra with Tests* [25, 32], or *GKAT* for short, reasons about *trace equivalence* between simple `while`-programs. By abstracting the meaning of the primitive tests and actions, it can focus on how tests determine which actions are performed. For example, the two programs above are equivalent in GKAT: they both execute p when t holds, and q when it does not. GKAT is also able to verify nontrivial equivalences like

$$\texttt{while } t \texttt{ do } p;\ \texttt{while } s \texttt{ do } \{\ q;\ \texttt{while } t \texttt{ do } p\ \} \equiv \texttt{while } t \vee s \texttt{ do } \{\ \texttt{if } t \texttt{ then } p \texttt{ else } q\ \}$$

ACM XXXX-XXXX/2024/10-ART
https://doi.org/XXXXXXX.XXXXXXX

Surprisingly, GKAT equivalence is decidable in nearly-linear time (assuming the set of test variables is fixed) [32], making it a reasonable compromise between complexity and granularity.

Nevertheless, the abstraction of GKAT comes at the price of not being able to verify some straightforward equivalences. First, as mentioned, GKAT disregards the meaning of primitive programs and tests. For instance, when given a program like

$$\texttt{if } y \neq 0 \texttt{ then } \{\ x := 42;\ p\ \} \texttt{ else } \{\ x := 42;\ q\ \}, \tag{1}$$

we can note that a change in the value of x does not have effect on whether $y \neq 0$. Hence, it should be possible to factor the assignment to x out of the branches, and obtain

$$x := 42;\ \texttt{if } y \neq 0 \texttt{ then } p \texttt{ else } q. \tag{2}$$

GKAT does not admit this equivalence, precisely because it is agnostic with respect to the meaning of primitive actions. However, moving to a setting that accounts for the semantics of actions is hard, because Turing completeness — and by extension, undecidability — lurks nearby [4, 19, 26].

Second, GKAT excludes non-local control-flow constructs like `goto`, `break`, and `return`. In a general imperative language, this does not limit expressivity, as these constructs can be recovered using variables [13] – indeed, the Böhm-Jacopini theorem says that every type of control flow can be written using a single `while`-loop [5]. However, lacking both variables and non-local control structures, GKAT is not able to express all control flow in real-world programs.

As a concrete example, consider the programs below. The control flow in Program 3 is based purely on labels and `goto`. Meanwhile, Program 4 is structured as a loop with the option to terminate early using `break`. These programs happen to be trace equivalent (i.e., they always execute the same actions in the same order) but represent behavior not expressible in plain GKAT [25, 31].

$$\texttt{label } \ell_0;\ \texttt{if } \neg t \texttt{ then goto } \ell_1;\ p;\ \texttt{if } t \texttt{ then goto } \ell_1;\ q;\ \texttt{goto } \ell_0;\ \texttt{label } \ell_1 \tag{3}$$

$$\texttt{while } t \texttt{ do } \{\ p;\ \texttt{if } \neg t \texttt{ then } q \texttt{ else break}\ \} \tag{4}$$

As it turns out, deciding equivalence between these complex programs is essential in verifying control-flow manipulation procedures. Specifically, consider the control flow structuring phase of a decompiler [9], which is tasked with converting conditional and unconditional jumps into more conventional control flow constructs as well as possible. Program 3 can be thought of as pseudo-assembly that models the input of this process, and Program 4 is a plausible outcome of decompilation. Thus, the control-flow structuring process is correct when Programs 3 and 4 are equivalent.

To overcome the limitations of GKAT, we propose control flow GKAT (CF-GKAT), an extension that is capable of equating some interesting programs. Concretely, we extend GKAT in two ways.

First, we add *indicator variables*, which can be assigned and tested against hardcoded values, and do not appear in other primitive actions and tests. For example, assignments like $x := 42$ are allowed, but assignments like $x := y + 1$ are not. This strikes a delicate balance, by being weak enough to exclude general computation (thus keeping equivalence decidable), yet strong enough to model the equivalence of Programs 1 and 2. Indicator variables are used in control-flow transformation algorithms [13, 36], and this addition empowers CF-GKAT to verify them. In this paper, we focus on one single global indicator variable for brevity; extending our approaches to multiple indicator variable should be straightforward.

Second, we extend GKAT with the non-local control-flow constructs `goto`, `break` and `return`. This poses a challenge, as the non-local nature of these commands prevents a compositional semantics — after all, the precise meaning of a statement like `break` depends on its context. To overcome this, we propose an intermediate *continuation semantics*. In this semantics, each trace is

tagged with a "continuation" that can accept (terminate normally), break, return, or go to a label. Then, the trace semantics of the program can be obtained by resolving these continuations.

We were able to design an automaton model for CF-GKAT, where every CF-GKAT expression can be converted into a CF-GKAT automaton while preserving the continuation semantics. Furthermore, CF-GKAT automata and continuation semantics can be lowered into GKAT automata and trace semantics respectively, while preserving their semantic correspondence. We are thus able to reduce the problem of deciding trace equivalence of programs into deciding bisimilarity of two GKAT automata, which is known to be efficient. Consequently, CF-GKAT can soundly and completely verify trace equivalence of a larger class of programs, while preserving the nearly-linear efficiency of GKAT. For instance, it can automatically verify that Programs 3 and 4 are equivalent, and also to their single-loop equivalent obtained via the Böhm-Jacopini theorem [5] (given below).

$$\begin{aligned} & x := 1;\ \texttt{while}\ x \neq 0\ \texttt{do}\ \{ \\ & \quad \texttt{if}\ x = 1 \wedge t\ \texttt{then}\ \{\ p;\ x := 2\ \} \\ & \quad \texttt{else if}\ x = 2 \wedge \neg t\ \texttt{then}\ \{\ q;\ x := 1\ \} \\ & \quad \texttt{else}\ x := 0\ \} \end{aligned} \tag{5}$$

To put this theory to work, we implemented an equivalence checker for CF-GKAT. This checker is able to validate highly non-trivial program transformations, such as the aforementioned Böhm-Jacopini conversion. We also implemented a front-end to the equivalence checker that can convert C code to CF-GKAT expressions by leveraging Clang's parser. The resulting tool is able to automatically validate the outcome of Erosa and Hendren's classic `goto` elimination procedure [13], as well as the control flow structuring pass of several decompilers.

Outline. The remainder of this article is organized as follows. In Section 2, we give an overview of CF-GKAT, including its syntax, continuation semantics, and trace semantics. In Section 3, we propose an automaton model for the continuation semantics, called CF-GKAT automata; we show how to translate a CF-GKAT expression to a CF-GKAT automaton, which in turn can be lowered to a GKAT automaton; ultimately, this gives rise to an algorithm for checking trace equivalence of CF-GKAT expressions. In Section 4, we report on an implementation of our algorithm, along with several experiments. We discuss related work in Section 5, and conclude in Section 6.

Our definition is full rigorous, and the proof of our main correctness result is formalized in Coq proof assistant [11]. Additionally, we also provide proof sketches to convey intuitions.

2 OVERVIEW

In this section, we introduce the language of CF-GKAT, and gradually develop its semantics. We begin by explaining the syntax of CF-GKAT; after that, we delve into the semantics of its tests. We then introduce the intermediate semantic model of *(labeled and indexed families of) guarded languages with continuations*, which is flattened into to a model based on *guarded languages*. Having defined these tools, we then conclude by giving a semantics to CF-GKAT programs in this model. Along the way, we will single out and explain some of the finer points using examples.

2.1 Syntax

The syntax of CF-GKAT consists of two levels, similar to GKAT. At the bottom level, there are *tests*; these are Boolean assertions that can occur as guards inside conditional statements, or within assertions that occur in the program text. To model them, we fix a finite set of primitive tests T, which represent uninterpreted expressions that may or may not hold. The full syntax is as follows.

$$\mathrm{BExp} \ni b, c ::= \texttt{false} \mid \texttt{true} \mid t \in T \mid x = i\ (i \in I) \mid b \vee c \mid b \wedge c \mid \neg b$$

Compared to GKAT, tests in CF-GKAT include the *indicator variable test* $x = i$ (highlighted in blue) for each *indicator value* i drawn from a finite but fixed set of possible indicator values I. As the notation suggests, this test holds when the indicator variable x currently has the value i.

The top level syntax of GKAT is built using a finite set of uninterpreted commands Σ (the *primitive actions*), as well as *assertions* of the form $\texttt{assert}\ b$, where $b \in \mathrm{BExp}$ is a test. Expressions are composed using sequencing, `if` statements, and `while` loops. CF-GKAT extends the base elements of the syntax with indicator variable assignments $x := i$ (for each $i \in I$), which changes the value of the indicator variable x to i. In addition, it adds the non-local control flow commands `break` and `return`, as well as $\texttt{goto}\ \ell$ and $\texttt{label}\ \ell$, where ℓ is taken from a fixed but finite set of labels L. The full syntax is given below (additions relative to GKAT highlighted in blue again).

$$\begin{aligned}\mathrm{Exp} \ni e, f ::= {} & \texttt{assert}\ b \mid p \in \Sigma \mid x := i\ (i \in I) \mid e; f \mid \texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ f \mid \\ & \texttt{while}\ b\ \texttt{do}\ e \mid \texttt{break} \mid \texttt{return} \mid \texttt{goto}\ \ell\ (\ell \in L) \mid \texttt{label}\ \ell\ (\ell \in L)\end{aligned}$$

A *valid program*, or *program* for short, is an expression without (1) duplicate labels, (2) `goto` commands with an undefined label, or (3) `break` statements that occur outside a loop. For the sake of simplicity, we assume that the reader does not require a more formal definition of this notion.

Example 2.1. Any GKAT expression is a valid program. Also, programs 3 to 5 from the introduction are all valid CF-GKAT programs. The following expressions are *not* valid programs:

$$\begin{aligned}\texttt{label}\ \ell; (\texttt{if}\ t\ \texttt{then}\ \texttt{label}\ \ell; p\ \texttt{else}\ q) &\qquad \text{(the label } \ell \text{ is defined twice)} \\ (\texttt{while}\ \texttt{true}\ \texttt{do}\ p); \texttt{goto}\ \ell &\qquad \text{(the label } \ell \text{ is undefined)} \\ \texttt{if}\ t\ \texttt{then}\ \texttt{break}\ \texttt{else}\ p &\qquad \text{(break appears outside a loop)}\end{aligned}$$

Remark. For soundness, it is important that the indicator variable x does not occur in any primitive test $t \in T$ or action $p \in \Sigma$. In other words, x is completely divorced from the other actions in the program, and may influence execution only by affecting flow control.

2.2 Boolean semantics

To assign a semantics to CF-GKAT expressions, we first need to talk about the semantics of tests. Intuitively, each test in a CF-GKAT expression symbolically denotes a set of execution contexts in which it is true. But how do we model an execution context? Because the primitive tests from T are uninterpreted, we represent them by simply listing the ones that are true; the unlisted tests are then assumed to be false. This coincides with the semantics of (G)KAT [24, 32]. As for the indicator tests, we include the current value of the indicator variable x in the execution context. This means that each execution context is of the form (i, α), for $i \in I$ and $\alpha \subseteq T$.

Putting these ideas together, we can calculate the set of execution contexts that satisfy a given test $b \in \mathrm{BExp}$ by induction. This set will be regarded as the semantics of b.

Definition 2.2. Let At denote 2^T, the set of *atoms* of (the free Boolean algebra generated by) T. We define the *Boolean semantics* function $[\![-]\!] : \mathrm{BExp} \to 2^{I \times \mathrm{At}}$ inductively, as follows.

$$\begin{aligned}[\![\texttt{false}]\!] &\triangleq \emptyset & [\![t]\!] &\triangleq \{(i, \alpha) \mid i \in I, t \in \alpha\} & [\![b \vee c]\!] &\triangleq [\![b]\!] \cup [\![c]\!] \\ [\![\texttt{true}]\!] &\triangleq I \times \mathrm{At} & [\![x = i]\!] &\triangleq \{(i, \alpha) \mid \alpha \in \mathrm{At}\} & [\![b \wedge c]\!] &\triangleq [\![b]\!] \cap [\![c]\!] \\ & & & & [\![\neg b]\!] &\triangleq I \times \mathrm{At} \setminus [\![b]\!]\end{aligned}$$

Example 2.3. Take $T = \{t_1, t_2\}$ and $I = \{1, 2, 3\}$; then we can calculate that

$$[\![(t_1 \vee \neg t_2) \wedge (x = 2)]\!] = \{(\{t_1, t_2\}, 2), (\{t_1\}, 2), (\emptyset, 2)\}$$

In other words, the test above holds in execution contexts where t_1 and t_2 are both true (first element) or both false (last element), and those where t_1 is true but t_2 is false (middle element). In contrast, $[\![x = 1 \wedge x = 3]\!] = \emptyset$, which is to say that this test does not hold in any execution context, because the indicator variable x cannot be both 1 and 3 at the same time.

2.3 Guarded language with continuations

We can now turn our attention to the semantics of CF-GKAT. Like (G)KAT, the semantics of CF-GKAT is given in terms of *guarded languages* [24, 32], which are best thought of as sets of symbolic traces of the program. These traces record the initial, intermediate and final machine states observed during execution, as well as the actions that occurred between those states. Because the value of the indicator variable matters only for control flow, we do not consider indicators to be part of the machine state; hence, machine states in a guarded word are drawn from At.

Definition 2.4. A *guarded word* is a sequence of the form $\alpha_1 p_1 \alpha_2 p_2 \cdots \alpha_{n-1} p_n \alpha_n$, where $\alpha_i \in \mathrm{At}$ and $p_i \in \Sigma$; that is to say, guarded words are elements of the regular language $\mathrm{At} \cdot (\Sigma \cdot \mathrm{At})^*$. We refer to sets of guarded words as *guarded languages*; the set of guarded languages is denoted $\mathcal{G}$.

Example 2.5. Let $T = \{t_1, t_2\}$ and $\Sigma = \{p_1, p_2\}$. Now the guarded word $\{t_1\}p_1\{t_2\}p_2\emptyset$ represents a program trace that starts out in a machine state where t_1 is true (but t_2 is not). The program then executes the action p_1, after which t_2 is true (but t_1 is not). Finally, the program goes on to execute the action p_2, and halts in a state where neither t_1 nor t_2 is true.

Our semantics of a CF-GKAT expression will ultimately be a guarded language. To get there, however, we will need an intermediate *continuation semantics*, which can account for the indicator variables as well as the non-local flow control statements. The remainder of this subsection is dedicated to explaining the domain of continuation semantics, based on *guarded words with continuations*. Intuitively, these are guarded words equipped with a piece of information called a *continuation*, which contains information about how flow control continues after the program ends. This could, for instance, tell us that the execution will continue at a location marked by a label.

The possibility of including continuation information at the end of a trace allows us to define a semantics of CF-GKAT expressions inductively. This is especially necessary in the case of non-local control flow, because the label may occur in an entirely different part of the program whose traces have not yet been computed. Once the continuation semantics of a CF-GKAT program in terms of guarded languages with continuations is known, we can flatten it into a guarded language.

Definition 2.6. A *guarded word with continuation* is a pair $w \cdot c$, where w is a guarded word and c is a *continuation*, which can take on one of the following forms for $i \in I$ and $\ell \in L$:

$$\textbf{acc}\ i \qquad \textbf{brk}\ i \qquad \textbf{ret} \qquad \textbf{jmp}\ (\ell, i)$$

We write C for the set of all continuations. A set of guarded words with continuations is a *guarded language with continuations*; the set of guarded languages with continuations is written $\mathcal{C}$.

Intuitively, the different types of continuation may be interpreted as follows:

- The continuation **acc** i represents that the trace has successfully reached the end of this part of the program, with indicator value i. Execution can be picked up if the program is put in a larger context — e.g., if $w \cdot \textbf{acc}\ i$ is a trace of e, then it may be combined with a trace found when f is executed with indicator value i to compute the semantics of $e; f$.
- A continuation of the form **brk** i signals that the trace ends by halting the loop in which it occurs. Execution can resume only after this loop (with indicator value i). This kind of trace cannot be composed on the right, as is done for traces with accepting continuations, because we first need to enclose it in a loop to halt; it will then be converted into **acc** i.

- The continuation **ret** represents a trace that ends in the program halting completely. Traces of this kind will percolate upwards in the semantics, without changing their continuation. These are intended to model the `return` statement, which halts the program no matter how deeply it is nested. In this case, the indicator value does not matter any more.
- Finally, the continuation **jmp** (ℓ, i) is put on traces that will continue executing from label ℓ, with indicator value i. Like **brk** i and **ret**, these traces do not compose on the right, but unlike **brk** i this continuation does not change, as jump resolution happens only at the end, when the continuation semantics is known for the entire program.

Example 2.7. Let w be the guarded word from the previous example; the guarded word with continuation $w \cdot$ **jmp** $(\ell_1, 2)$ represents a partial program trace that takes the steps represented by w, and will continue executing at the label ℓ_1 with an indicator value of 2.

2.4 Indexed families and sequencing

The continuation semantics of a CF-GKAT expression takes a starting indicator value, and produces a guarded language with continuations representing the traces of that program when started with this indicator value. This semantics is modeled by the following.

Definition 2.8. An *indexed family* of guarded languages (with continuations), or "indexed family" for the sake of brevity, is function from I to guarded languages (with continuations). Typically, we use G and H to denote an indexed family. To lighten notation, we write G_i to denote $G(i)$.

Similar to guarded languages, indexed families can be composed in several ways. In particular, we are interested in the sequencing operation and the Kleene star operation of indexed families, because these will turn out to be useful when defining the continuation semantics of CF-GKAT.

When sequencing two families G and H, the traces in G_i with a continuation of the form **acc** j will be composed with traces in H_j; traces with different continuations are copied over in full, because they do not compose on the right. Formally, this operation is defined as follows.

Definition 2.9. Let $G, H : I \to \mathcal{C}$. We write $G \diamond H$ for the *sequencing* (or *concatenation*) operation of G and H, which is defined as the smallest family of guarded languages with continuations (in the pointwise order) satisfying the following rules for all $i, j \in I$ as well as all $\ell \in L$:

$$\frac{w\alpha \cdot \textbf{acc}\ j \in G_i \qquad \alpha x \cdot c \in H_j}{w\alpha x \cdot c \in (G \diamond H)_i} \qquad \frac{w \cdot c \in G_j \qquad c \in \{\textbf{brk}\ i, \textbf{acc}\ i, \textbf{jmp}\ (\ell, i)\}}{w \cdot c \in (G \diamond H)_j}$$

The first rule composes accepting traces in G with traces in H, picking up with the indicator value where the first trace left off. Note also that this rule requires the last atom in the trace on the left to match the first atom in the trace on the right, because we want the second trace to start from the machine state computed in the first trace. This mirrors the *coalesced product* used to define the sequential composition of guarded languages (without continuations) [24]. The last rule ensures that traces that encountered non-local control flow within G are preserved in $G \diamond H$.

Example 2.10. Let $I = \{1, 2\}$, and let G and H be indexed families given by:

$$G_1 = \{\alpha p \beta \cdot \textbf{brk}\ 1,\ \beta p \alpha \cdot \textbf{acc}\ 2\} \qquad G_2 = \{\alpha q \beta \cdot \textbf{acc}\ 1\}$$
$$H_1 = \{\gamma q \beta \cdot \textbf{ret}\} \qquad H_2 = \{\alpha r \beta \cdot \textbf{jmp}\ (\ell_1, 1)\}$$

Then we can compute that the sequencing $G \diamond H$ is the following indexed family:

$$(G \diamond H)_1 = \{\alpha p \beta \cdot \textbf{brk}\ 1,\ \beta p \alpha r \beta \cdot \textbf{jmp}\ (\ell_1, 1)\} \qquad (G \diamond H)_2 = \emptyset$$

Here, we find that $(G \diamond H)_1$ contains $\alpha p \beta \cdot$ **brk** 1 by the second rule, because G_1 does. Furthermore, the trace $\beta p \alpha \cdot$ **acc** 2 in G_1 is composed with $\alpha r \beta \cdot$ **jmp** $(\ell_1, 1)$ from H_2 to form $\beta p \alpha r \beta \cdot$ **jmp** $(\ell_1, 1)$ in

$(G \diamond H)_1$, by the first rule. The set $(G \diamond H)_2$ is empty, because despite the fact that $\alpha q \beta \cdot \mathbf{acc}\ 1 \cdot \mathbf{acc}\ 1 \in G_2$, there is no trace in H_2 that starts with β, and so neither rule can apply.

2.5 Continuation semantics

With the theory of indexed families in place, we can now define the continuation semantics $[\![e]\!]^\sharp$ of a CF-GKAT program e in terms of an indexed family. We start with the base cases.

Definition 2.11 (Continuation semantics, base). For all $i, j \in I$, we define the following sets:

$$[\![\texttt{assert}\ b]\!]^\sharp_i \triangleq \{\alpha \cdot \mathbf{acc}\ i \mid (i, \alpha) \in [\![b]\!]\} \qquad [\![\texttt{goto}\ \ell]\!]^\sharp_i \triangleq \{\alpha \cdot \mathbf{jmp}\ (\ell, i) \mid \alpha \in \mathrm{At}\}$$
$$[\![p]\!]^\sharp_i \triangleq \{\alpha p \beta \cdot \mathbf{acc}\ i \mid \alpha, \beta \in \mathrm{At}\} \qquad [\![\texttt{label}\ \ell]\!]^\sharp_i \triangleq \{\alpha \cdot \mathbf{acc}\ i \mid \alpha \in \mathrm{At}\}$$
$$[\![x := j]\!]^\sharp_i \triangleq \{\alpha \cdot \mathbf{acc}\ j \mid \alpha \in \mathrm{At}\} \qquad [\![\texttt{break}]\!]^\sharp_i \triangleq \{\alpha \cdot \mathbf{brk}\ i \mid \alpha \in \mathrm{At}\}$$
$$[\![\texttt{return}]\!]^\sharp_i \triangleq \{\alpha \cdot \mathbf{ret} \mid \alpha \in \mathrm{At}\}$$

Each of these base syntax elements yields a simple (finite) indexed family. For the constructs `return`, `goto`, and `break`, all traces terminate immediately in the corresponding continuation.

We inherit the semantics of assertions and primitive actions from (G)KAT [24, 32]. Assertions have traces that accept when their only atom satisfies the test. A primitive action p yields traces of the form $\alpha p \beta \cdot \mathbf{acc}\ i$ for all $\alpha, \beta \in \mathrm{At}$ to witness that p is uninterpreted: we could reach any other machine state by running p. The only information retained is the value of the indicator variable, because primitive actions cannot interact with indicators. In contrast with primitive actions, an assignment like $x := j$ has traces that accept immediately, without changing the machine state; however, each trace ends with the indicator value j — regardless of the initial indicator value i.

Finally, labels are encoded as no-operations, which makes them neutral for sequencing operator, i.e., we have $[\![\texttt{label}\ \ell]\!]^\sharp \diamond G = G = [\![\texttt{label}\ \ell]\!]^\sharp \diamond G$ for all indexed families G. This is because labels serve only as potential starting points of execution; we will leverage them in the next subsection.

We now turn our attention to the program composition operators. These are generalizations of the guarded language semantics of GKAT [32]. First of all, the $\texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ f$ filters out traces in the semantics of the e that satisfy the guard b, as well as the traces in f that invalidate it.

Definition 2.12 (Continuation semantics, branching). Let $e, f \in \mathrm{Exp}$. We define $[\![\texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ f]\!]^\sharp$ as the least indexed family that satisfies the following rules for all $i \in I$:

$$\frac{\alpha \in [\![b]\!] \qquad \alpha w \cdot c \in [\![e]\!]^\sharp_i}{\alpha w \cdot c \in [\![\texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ f]\!]^\sharp_i} \qquad \frac{\alpha \notin [\![b]\!] \qquad \alpha w \cdot c \in [\![f]\!]^\sharp_i}{\alpha w \cdot c \in [\![\texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ f]\!]^\sharp_i}$$

The semantics of the sequencing operator is easy: it just composes the semantics of the operands with the sequencing operator we have for indexed families. For loops, some more care is needed because traces can be iterated, and we need to account for early termination.

Definition 2.13 (Continuation semantics, sequencing and loops). Let $e, f \in \mathrm{Exp}$. We define $[\![e; f]\!]^\sharp \triangleq [\![e]\!]^\sharp \diamond [\![f]\!]^\sharp$. Also, for all $b \in \mathrm{BExp}$, we define $[\![\texttt{while}\ b\ \texttt{do}\ e]\!]^\sharp$ as the least indexed family satisfying:

$$\frac{i \in I \qquad \alpha \notin [\![b]\!]}{\alpha \cdot \mathbf{acc}\ i \in [\![\texttt{while}\ b\ \texttt{do}\ e]\!]^\sharp_i} \qquad \frac{\alpha \in [\![b]\!] \qquad \alpha w \cdot c \in ([\![e]\!]^\sharp \diamond [\![\texttt{while}\ b\ \texttt{do}\ e]\!]^\sharp)_i}{\alpha w \cdot \lfloor c \rfloor \in [\![\texttt{while}\ b\ \texttt{do}\ e]\!]^\sharp_i}$$

The operation $\lfloor - \rfloor$ in the last rule is defined by $\lfloor c \rfloor = \mathbf{acc}\ i$ when $c = \mathbf{brk}\ i$, and $\lfloor c \rfloor = c$ otherwise.

The first rule accounts for traces that halt immediately because the loop guard is false. The second rule allows prepending traces from the loop body that satisfy the guard. Because of the

way sequencing works, body traces that end in **brk** i may occur; the second rule converts their continuations to **acc** i, signaling that the loop has been exited and normal control flow can resume.

The semantics we have so far defines the traces of a program starting from the beginning. However, a CF-GKAT program can be started from any label. To obtain these traces for a given label ℓ, we must descend into the program until we encounter the corresponding label statement. For the base cases, this is relatively simple to accomplish: just check if we start at the label.

Definition 2.14 (Continuation semantics starting from a label, base). Let $e \in \mathrm{Exp}$. For each $\ell \in L$, we define the following guarded languages with continuations:

$$\begin{aligned} [\![\texttt{assert}\ b]\!]^\ell_i &\triangleq \emptyset & [\![\texttt{goto}\ \ell']\!]^\ell_i &\triangleq \emptyset \\ [\![p]\!]^\ell_i &\triangleq \emptyset & [\![\texttt{label}\ \ell']\!]^\ell_i &\triangleq \{\alpha \cdot \mathbf{acc}\ i \mid \alpha \in \mathrm{At},\ \ell = \ell'\} \\ [\![x := j]\!]^\ell_i &\triangleq \emptyset & [\![\texttt{break}]\!]^\ell_i &\triangleq \emptyset \\ [\![\texttt{return}]\!]^\ell_i &\triangleq \emptyset \end{aligned}$$

Note how none of these cases has a trace, except the one for $[\![\texttt{label}\ \ell']\!]^\ell_i$ when $\ell' = \ell$, which accepts immediately. With these cases covered, we can then treat the inductive step.

Definition 2.15 (Continuation semantics starting from a label, sequencing and branching). Let $e, f \in \mathrm{Exp}$, $b \in \mathrm{BExp}$ and $\ell \in L$. We define the following indexed families to cover the traces of CF-GKAT programs starting from the label ℓ when composed using branching or sequencing:

$$[\![\texttt{if}\ b\ \texttt{then}\ e\ \texttt{else}\ f]\!]^\ell_i \triangleq [\![e]\!]^\ell_i \cup [\![f]\!]^\ell_i \qquad [\![e; f]\!]^\ell_i \triangleq ([\![e]\!]^\ell \diamond [\![f]\!]^\sharp)_i \cup [\![f]\!]^\ell_i$$

For branching, the semantics starting from ℓ disregards the guard and descends into the operands. The sequencing case is more interesting: here, we still need to account for the traces that start from the beginning of f after executing a trace in e starting from the label ℓ.

The only remaining case to cover is the loop. In this case, if we start execution from a label somewhere in the body, we may need to start the loop again after completing the loop body. On the other hand, early termination in the loop body still needs to be turned into an accepting trace.

Definition 2.16 (Continuation semantics starting from a label, loops). Let $e \in \mathrm{Exp}$ and $b \in \mathrm{BExp}$. We define the indexed family $[\![\texttt{while}\ b\ \texttt{do}\ e]\!]^\ell$ below, where $\lfloor - \rfloor$ is as in Definition 2.13:

$$[\![\texttt{while}\ b\ \texttt{do}\ e]\!]^\ell_i = \{w \cdot \lfloor c \rfloor \mid w \cdot c \in ([\![e]\!]^\ell \diamond [\![\texttt{while}\ b\ \texttt{do}\ e]\!]^\sharp)_i\}$$

2.6 Guarded language semantics

The continuation semantics of a CF-GKAT program e in terms of indexed families $[\![e]\!]^\sharp$ uses continuations to record how a trace ends. In particular, some traces may end with the continuation of the form **jmp** (ℓ, i), signaling that computation needs to continue from the label ℓ. The semantics $[\![e]\!]^\ell$ provides the necessary information to resolve such jump continuations: we can resume **jmp** (ℓ, i) by concatenating with the traces in $[\![e]\!]^\ell_i$. We will end this section by doing just that.

To formalize our approach, we need a way to refer to the continuation CF-GKAT semantics of a program as a whole, i.e., for all indicator values, starting from either the beginning or some label.

Definition 2.17. A *labeled family of guarded languages (with continuations)*, or *labeled family* for short, is a function G from $L + \sharp$ to indexed families of guarded languages (with continuations), e.g., $G : L + \sharp \to I \to \mathcal{C}$. We often use superscripts to denote the value at a given label $\sharp$, writing L^ℓ for $L(\ell)$. Note that under this convention, L^ℓ is an indexed family, which means that we may further unravel by writing L^ℓ_i to obtain the guarded language with continuations $L(\ell, i)$.

Crucially, we can retrofit the continuation semantics $[\![e]\!]$ as a labeled family; after all, $[\![e]\!]^\sharp$ is an indexed family, and so is $[\![e]\!]^\ell$ for each $\ell \in L$. We will thus treat $[\![e]\!]$ as such from this point on.

To resolve the jumps in a labeled family of guarded languages with continuations, we resolve the traces ending in **jmp** (ℓ, i) by looking up the traces that originate from label ℓ with indicator value i. We also remove the continuations **acc** i and **ret**, because those come with traces that either reached the end of the program, or encountered a `return` statement respectively. Continuations of the form **brk** i should not occur at the top level when computing the semantics of a program, so we can ignore them. The result is a labeled family of guarded languages (without continuations).

Definition 2.18. Let $G : L + \sharp \to I \to \mathcal{C}$ be a labeled family of guarded languages with continuations. We write $L\!\downarrow$ for the (point-wise) least labeled family of guarded languages $G\!\downarrow$ such that the following rules are satisfied for all $k \in L + \sharp$, $\ell \in L$, and $i, j \in I$:

$$\frac{w \cdot \mathbf{acc}\ i \in G_i^k}{w \in G\!\downarrow_i^k} \qquad \frac{w \cdot \mathbf{ret} \in G_i^k}{w \in G\!\downarrow_i^k} \qquad \frac{w\alpha \cdot \mathbf{jmp}\ (\ell, j) \in G_i^k \qquad \alpha x \in G\!\downarrow_j^\ell}{w\alpha x \in G\!\downarrow_i^k}$$

The first two rules take care of flattening guarded words with continuations that in acceptance, while the third rule strings together guarded words continuations that jump to a different label.

Example 2.19. Let G be the labeled family of guarded languages with continuations defined by

$$\begin{aligned} G_1^\sharp &= \{\alpha \cdot \mathbf{jmp}\ (\ell, 1)\} & G_2^\sharp &= \emptyset \\ G_1^\ell &= \{\alpha p \alpha \cdot \mathbf{jmp}\ (\ell', 1), \beta \cdot \mathbf{acc}\ 1\} & G_2^\ell &= \{\alpha \cdot \mathbf{jmp}\ (\ell', 2)\} \\ G_1^{\ell'} &= \{\alpha q \alpha \cdot \mathbf{jmp}\ (\ell, 1), \alpha r \beta \cdot \mathbf{jmp}\ (\ell, 1)\} & G_2^{\ell'} &= \{\alpha \cdot \mathbf{jmp}\ (\ell, 1)\} \end{aligned}$$

Now $G\!\downarrow_1^\sharp$ contains, among other things, the guarded word $\alpha p \alpha q \alpha p \alpha r \beta$.

Note furthermore that $G\!\downarrow_2^\ell$ is empty, despite G_2^ℓ containing a guarded word with a continuation that has a mutual jump with another guarded word with continuation in $G_2^{\ell'}$, as these can never be concatenated into one guarded word with continuation of the form **acc** i or **ret**.

In total, we can then obtain the semantics of a CF-GKAT term as $[\![e]\!]\!\downarrow$, in the form of a labeled family of guarded languages. This concludes our discussion of the semantics of CF-GKAT.

3 DECISION PROCEDURE

To decide whether two CF-GKAT expressions denote the same guarded language, we propose CF-GKAT automata. We will show that every CF-GKAT expression can be converted to a CF-GKAT automaton with the same continuation semantics, by induction on the expression *à la Thompson [34]*. However, because CF-GKAT automata implement the continuation semantics, directly comparing them for language equivalence will not be sufficient. For instance, the following programs exhibit identical trace semantics, but differ in their continuation semantics:

$$x := 1 \qquad\qquad \texttt{assert true}.$$

The first program sets the indicator variable to 1, and the second program simply does nothing. These programs have the same trace semantics, because the trace semantics does not care about the final value of the indicator variable. Yet, their continuation semantics are different: guarded words in $[\![x := 1]\!]_i^\sharp$ are always paired with the continuation **acc** 1 regardless of i, but $[\![\texttt{assert true}]\!]_i^\sharp$ preserves the starting indicator by outputting the continuation **acc** i. Furthermore, equivalence checking needs to account for jump resolution, by looking up the sequel of a trace ending in a continuation of the form **jmp** (i, ℓ) in the traces starting from label ℓ with indicator value i.

Thus, our decision procedure does not rely equivalence checking of CF-GKAT automata; instead, we *lower* the CF-GKAT automata into GKAT automata with the right semantics. This allows us to reuse the nearly-linear decision procedure for GKAT automata equivalences [32]. Finally, the soundness and completeness of our decision procedure follows from the correctness of Thompson's construction (Theorem 3.19), the correctness of the lowering (Theorem 3.13), and finally the correctness of GKAT automata equivalence checking [32].

3.1 GKAT automata

To establish our decision procedure, we will first recap GKAT automata and their trace semantics.

Definition 3.1 (GKAT automata [25, 32]). A *GKAT automaton* $A \triangleq \langle S, \delta, \hat{s} \rangle$ consists of a set of *states* S, a *transition function* $\delta : S \to \mathrm{At} \to \bot + \top + \Sigma \times S$, and a *start state* $\hat{s} \in S$.

Intuitively, given a state s and an atom α accounting for the truth value of each primitive test, a GKAT automaton will either *reject* the input, represented by $\delta(s, \alpha) = \bot$; *accept* the input, represented by $\delta(s, \alpha) = \top$; or to *transition* to a new state in S after executing an action from Σ, represented by $\delta(s, \alpha) \in \Sigma \times S$. A GKAT automaton induces a guarded language, by tracing all the possible execution paths reaching an accepting transition.

Definition 3.2. Given a GKAT automaton $A \triangleq \langle S, \delta, \hat{s} \rangle$, we define $[\![-]\!]_A : S \to \mathcal{G}$ as the (pointwise) smallest function satisfying the following rules for all $s \in S$ and $\alpha \in \mathrm{At}$:

$$\frac{\delta(s, \alpha) = \top}{\alpha \in [\![s]\!]_A} \qquad \frac{\delta(s, \alpha) = (p, s') \qquad w \in [\![s']\!]_A}{\alpha p w \in [\![s]\!]_A}$$

Finally, we define the guarded language semantics of A by setting $[\![A]\!] = [\![\hat{s}]\!]_A$.

The trace equivalence of finite GKAT automata is decidable, which we record as follows.

Theorem 3.3 (Decidability for GKAT [32]). *Given two finite GKAT automata A_0 and A_1, it is decidable whether they represent the same guarded language, i.e., whether $[\![A_0]\!] = [\![A_1]\!]$. The algorithm to do this has a complexity that is nearly-linear*[1] *in the total number of states.*

3.2 CF-GKAT automata

To leverage the efficient decision algorithm for GKAT automata, we will need to convert each CF-GKAT expression e and a starting indicator value i into a GKAT automaton that recognizes the guarded language $[\![e]\!]\downarrow_i$. As discussed, this process is separated into two steps, where we use CF-GKAT automata as an intermediate between CF-GKAT expressions and GKAT automata. Like GKAT automata, CF-GKAT automata need a transition structure. As it turns out having a separate type for this transition structure will turn out to be useful, so we isolate it as follows.

Definition 3.4 (CF-GKAT dynamics). Given a set S, we write $G(S)$ for the set of possible *CF-GKAT dynamics* on S, which is given by the following function type, where C is as in Definition 2.6:

$$G(S) \triangleq I \times \mathrm{At} \to \bot + C + \Sigma \times X \times I.$$

Intuitively, the elements of $G(S)$ represent outgoing transitions of a single (initial) state or label in a CF-GKAT automaton over a state set S. Given the current indicator value $i \in I$ and an atom α accounting for the truth value of each primitive test, a dynamics $\rho \in G(S)$ may either:

- *reject* the input, represented by $\rho(i, \alpha) = \bot$;
- offer a *continuation*, represented by $\rho(i, \alpha) \in C$; or

[1] $\mathcal{O}(\hat{\alpha}(n))$, where $\hat{\alpha}$ is the inverse Ackermann function; c.f. [33].

- execute a primitive action in Σ and set a new indicator value from I while transitioning to a new state in S, represented by $\rho(i, \alpha) \in \Sigma \times X \times I$.

Then the definition of CF-GKAT automaton is similar to GKAT automaton, except we will need a function $\lambda : L \to G(S)$ where $\lambda(\ell)$ provides a dynamics representing the "entry point" of label ℓ.

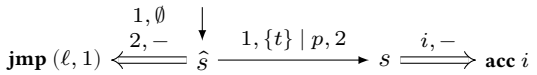
Definition 3.5. A *CF-GKAT automaton* $A \triangleq \langle S, \delta, \hat{s}, \lambda \rangle$ consists of a set of *states* S, a *transition function* $\delta : S \to G(S)$, a *start state* $\hat{s} \in S$, and a *jump map* $\lambda : L \to G(S)$.

The transition map δ assigns every state a dynamics, while the jump map λ assigns a dynamics to each label, indicating how to resume computation when a jump continuation is reached.

Example 3.6 (A simple CF-GKAT automaton). Let $I = \{1, 2\}$ and consider the following program:

$$\texttt{if}\ b \wedge (x = 1)\ \texttt{then}\ \{x := 2; \texttt{label}\ \ell;\ p\}\ \texttt{else}\ \{x := 1;\ \texttt{goto}\ \ell\}.$$

We can construct the following automaton with the same continuation semantics from state $\hat{s}$:



Here, $\hat{s} \xrightarrow{1,\{t\}|p,2} s$ means that $\delta(\hat{s}, 1, \{t\}) = (p, s, 2)$ and $s \overset{i,-}{\Longrightarrow} \mathbf{acc}\ i$ means that $\delta(s, i, \alpha) = \mathbf{acc}\ i$ for $\alpha \in \mathrm{At}$ and $i \in I$. The behavior of this automaton, when starting from $\hat{s}$, is as follows.

- If the input atom is $\{t\}$ — that is, the test t is true — and the indicator is 1, we transition to the state s, while setting the indicator to 2 and executing p (and skipping over label ℓ). Upon reaching s, the automaton accepts unconditionally, preserving the indicator.
- Otherwise, if the input atom is $\emptyset$ — that is, the test t is false — or the indicator value is anything other than 1, the automaton will simply offer the continuation $\mathbf{jmp}\ (\ell, 1)$, thereby telling execution to continue from label ℓ with indicator value 1.

As we can see, the behavior of $\hat{s}$ indeed matches the behavior of the program when executing from the start. On the other hand, the entry dynamics for ℓ can be defined as follows:

$$\forall i \in I, \alpha \in \mathrm{At}, \lambda(\ell, i, \alpha) \triangleq (p, s, i).$$

To put the above definition into words: when jumping to the label ℓ, we will reach the state s while executing p. Thus, the behavior of $\lambda(\ell)$ matches the behavior of the program when starting from ℓ.

Remark. We could have opted to instrument CF-GKAT automata with a *state* for each label ℓ, rather than a dynamics. This would simplify their definition, at the cost of complicating the Thompson construction (Section 3.4). An additional advantage of this approach is that we avoid creating unreachable states when lowering CF-GKAT automata to GKAT automata (Section 3.3).

To formalize the intuition of "behaviors", as seen in the previous example, we will assign a continuation semantics to each CF-GKAT automaton. It is convenient to first assign a semantics to each dynamics $\rho \in G(S)$ in a CF-GKAT automaton $A \triangleq \langle S, \delta, \hat{s}, \lambda \rangle$, instead of every state.

Definition 3.7 (continuation semantics). Given an automaton $A \triangleq \langle S, \delta, \hat{s}, \lambda \rangle$, the continuation semantics of each dynamics $\rho \in G(S)$ is an indexed family $[\![\rho]\!]_A : I \to \mathcal{C}$, defined as the (pointwise) smallest set satisfying the following rules for $i, j \in I$, $\alpha \in \mathrm{At}$ and $c \in C$:

$$\frac{\rho(i,\alpha) = \mathbf{acc}\ j}{\alpha \cdot \mathbf{acc}\ j \in ([\![\rho]\!]_A)_i} \qquad \frac{\rho(i,\alpha) = \mathbf{brk}\ j}{\alpha \cdot \mathbf{brk}\ j \in ([\![\rho]\!]_A)_i} \qquad \frac{\rho(i,\alpha) = \mathbf{ret}}{\alpha \cdot \mathbf{ret} \in ([\![\rho]\!]_A)_i}$$

$$\frac{\rho(i,\alpha) = \mathbf{jmp}\ (\ell, j)}{\alpha \cdot \mathbf{jmp}\ (\ell, j) \in ([\![\rho]\!]_A)_i} \qquad \frac{\rho(i,\alpha) = (p, s, j) \qquad w \cdot c \in ([\![\delta(s)]\!]_A)_j}{\alpha p w \cdot c \in ([\![\rho]\!]_A)_i}$$

Similar to the continuation semantics of expressions, the continuation semantics of automata are also labeled families of guarded languages with continuations: the semantics from the start $[\![A]\!]^\sharp$ is defined by the dynamics of the start state, and that of a label $\ell \in L$ is defined by the jump map:

$$[\![A]\!]^\sharp = [\![\delta(\hat{s})]\!]_A, \qquad [\![A]\!]^\ell = [\![\lambda(\ell)]\!]_A \text{ for } \ell \in L.$$

Example 3.8. Returning to the CF-GKAT automaton from Example 3.6, we find that

$$[\![A]\!]^\sharp_1 = \{\emptyset \cdot \mathbf{jmp}\ (\ell, 1), \{t\}p\emptyset \cdot \mathbf{acc}\ 2, \{t\}p\{t\} \cdot \mathbf{acc}\ 2\}$$

$$[\![A]\!]^\sharp_2 = \{\emptyset \cdot \mathbf{jmp}\ (\ell, 1), \{t\} \cdot \mathbf{jmp}\ (\ell, 1)\} \qquad [\![A]\!]^\ell_1 = [\![A]\!]^\ell_2 = \{\alpha p \beta \cdot \mathbf{acc}\ i : \alpha, \beta \in \mathrm{At}\}$$

3.3 Lowering CF-GKAT automata to GKAT automata

The process to lower a CF-GKAT automaton $\langle S, \delta, \hat{s}, \lambda \rangle$ into a GKAT automaton consists of two different components. We first "embed" the indicator values into the state set; the new state set then becomes $S \times I$. After that, we resolve all the continuations in transitions using the jump map.

The second step requires some care. When $\delta(s, i, \alpha) = \mathbf{jmp}\ (\ell, j)$, the α-transition leaving the state (s, i) will take the behavior of ℓ starting from indicator j and atom α, that is $\lambda(\ell, j, \alpha)$. This by itself is alright, but we also need to account for subprograms such as `label` l; $x := k$; `goto` ℓ', which entails that $\lambda(\ell, j, \alpha)$ points to a different label by returning $\mathbf{jmp}\ (\ell', k)$. In turn, $\lambda(\ell', k, \alpha)$ may also yield another jump, et cetera. To resolve these chained jumps, we will need to iterate the jump map, and terminate when either the result is no longer a jump, or an infinite loop is detected.

Definition 3.9 (iteration lifting). Given a function $h : X \to X + \bot + E$, where X is a finite set and $\bot + E$ specifies the "exit results", we can iterate h until some exit value is reached, or a loop is detected. Formally, we define $\mathrm{iter}(h)$ as the least function satisfying for all $x \in X$ that

$$\mathrm{iter}(h)(x) = \begin{cases} \mathrm{iter}(h)(h(x)) & h(x) \in X \\ h(x) & h(x) \in E \end{cases}$$

in the directed-complete partial order (DCPO) on functions from X to $X + \bot + E$ where $f \leq g$ when for all $x \in X$, we have that $f(x) = \bot$ or $f(x) = g(x)$. This makes $\mathrm{iter}(h)$ the least fixed point of the Scott-continuous function on this DCPO that sends f to

$$x \mapsto \begin{cases} f(h(x)) & h(x) \in X \\ h(x) & h(x) \in E + \bot \end{cases}$$

By Kleene's fixed point theorem this uniquely defines $\mathrm{iter}(h)$.

Remark. When X is finite, we can directly compute $\mathrm{iter}(h)$ by iterating h. More precisely, we can define a helper function $\mathrm{iter}'(h) : 2^X \to X \to \bot + E$ with an additional parameter, as follows:

$$\mathrm{iter}'(h)(M)(x) \triangleq \begin{cases} \bot & \text{if } x \in M \\ h(x) & \text{if } x \notin M \text{ and } h(x) \in \bot + E\ ; \\ \mathrm{iter}'(h)(M \cup \{x\})(h(x)) & \text{if } x \notin M \text{ and } h(x) \in X \end{cases}$$

It is then not too hard to show that $\mathrm{iter}(h) = \mathrm{iter}'(h)(\emptyset)$. Intuitively, M keeps track of explored values in X; if an input has already been explored, then iter will return $\bot$. On the other hand, if $h(x)$ falls into the exit set $\bot + E$, then $\mathrm{iter}(h)$ will exit and return $h(x)$. Finally, if $h(x)$ falls into X, then $\mathrm{iter}(h)$ will continue to iterate with $h(x)$ as input, and mark x as explored. Note that $\mathrm{iter}(h)$ is total because X is finite, so elements in X cannot be explored twice.

For the sake of clarity, when defining a function h to iterate, we will write **cont** for the injection $X \to X + \bot + E$, so that **cont**(x) indicates iteration continues with the value x; and **exit** for the injection $\bot + E \to X + \bot + E$, so that **exit**$(e)$ indicates iteration stops with the value e.

For jump resolution, iteration will continue when the result of λ is a jump, but exit otherwise.

Definition 3.10 (Jump resolution). Let S be a finite set, and let $\lambda : L \to G(S)$ be a jump function. We define the resolved jump map $\lambda{\downarrow} : L \to G(S)$, as follows:

$$\lambda{\downarrow} = \text{iter}\left((\ell, i, \alpha) \mapsto \begin{cases} \textbf{cont}(\ell', i', \alpha) & \lambda(\ell, i, \alpha) = \textbf{jmp}\ (\ell', i') \\ \textbf{exit}(\lambda(\ell, i, \alpha)) & \text{otherwise} \end{cases}\right)$$

Because of the way iter works, this means that a cycle of **jmp**-continuations (without actions in between) resolves to $\bot$. Indeed, GKAT treats unproductive infinite iterations in the `while` loops with the same strategy [32], identifying non-terminating behavior with behavior that rejects explicitly.

Note that $\lambda{\downarrow}$ resolves all internal jumps, i.e., $\lambda{\downarrow}\ (\ell, i, \alpha)$ is never a jump continuation, because the iteration map used to define $\lambda{\downarrow}$ keeps iterating on values of this form. We can then use the resolved jump map to tie together the loose ends indicated by jump continuations. In the process, we can reroute continuations of the form **brk** i to rejection, as these should not occur at the top level of a well-formed program (internal **brk**-continuations are handled in the next section).

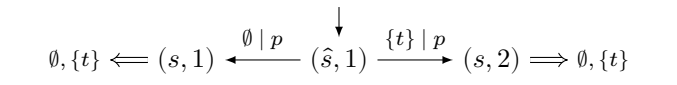
Definition 3.11 (Lowering). Given a CF-GKAT automaton $A \triangleq \langle S, \delta, \hat{s}, \lambda\rangle$ and $i \in I$, we define the GKAT automaton $A{\downarrow_i} \triangleq \langle S \times I, \delta{\downarrow}, (\hat{s}, i)\rangle$, where $\delta{\downarrow}$ is defined in two steps:

$$\delta'((s,i),\alpha) \triangleq \begin{cases} \lambda{\downarrow}\ (\ell, i, \alpha) & \delta(s,i,\alpha) = \textbf{jmp}\ (\ell, i) \\ \delta(s,i,\alpha) & \text{otherwise} \end{cases}$$

$$\delta{\downarrow}\ ((s,i),\alpha) \triangleq \begin{cases} \bot & \delta'(s,i,\alpha) = \textbf{brk}\ j \\ \top & \delta'(s,i,\alpha) = \textbf{ret} \text{ or } \delta'(s,i,\alpha) = \textbf{acc}\ j \\ \delta(s,i,\alpha) & \text{otherwise} \end{cases}$$

Note that $\delta{\downarrow}$ is well-defined: for all $s \in S$, $i \in I$ and $\alpha \in \text{At}$, it holds that $\delta{\downarrow}\ ((s,i),\alpha) \in \bot + \top + \Sigma \times (S \times I)$, as expected for a GKAT automaton on state set $S \times I$.

Example 3.12. If A is the automaton from Example 3.6, then $A{\downarrow_1}$ can be drawn as follows:



Here, we use similar graphical conventions as before; for instance, $(\hat{s}, 1) \xrightarrow{\{t\}|p} (s, 2)$ means that $\delta{\downarrow}\ ((\hat{s}, 1), \{t\}) = (p, (s, 2))$, and $(s, 2) \implies \emptyset$ means that $\delta{\downarrow}\ ((s, 2), \emptyset) = \top$.

In this lowered automaton, the jump continuation originally reached from $\hat{s}$ for indicator value 1 and t false (i.e., atom $\emptyset$) has been resolved via the jump map λ to continue in state s. This automaton also holds two copies of the state s, one for each possible indicator value; in this case, these states happen to have the same behavior, but in general they may be different.

Having defined our lowering operation, we can state its correctness as follows.

Theorem 3.13 (Correctness of lowering). *Let $A \triangleq \langle S, \delta, \hat{s}, \lambda\rangle$ be a CF-GKAT automaton. The translation from CF-GKAT automata to GKAT automata commutes with the semantic jump resolution operator, in the sense that for $i \in I$, it holds that $[\![A{\downarrow_i}]\!] = [\![A]\!]{\downarrow_i^\sharp}$.*

Proof sketch. More generally, we can prove that for any state s of A, it holds that $[\![(s,i)]\!]_{A{\downarrow_i}} = G{\downarrow_i^\sharp}$, in which G is the labeled family given by $G^\sharp = [\![\delta(s)]\!]_A$ and $G^\ell = [\![\lambda(\ell)]\!]_A$. This property, which implies the main claim, can be proved by induction on the length of guarded words. □

3.4 Converting expressions to CF-GKAT automata

The final piece of our puzzle is to convert CF-GKAT expressions to CF-GKAT automata. To accomplish this, we generalize a construction proposed for GKAT, which turns a GKAT expression into a GKAT automaton in a trace-equivalent manner [32]. This construction, which was inspired by Thompson's construction to obtain a non-deterministic finite automaton from a regular expression [34], proceeds by induction on the structure of the expression. In contrast to the original, howvever, the Thompson construction for GKAT produces a GKAT automaton with a *start dynamics*, also called an *initial pseudostate* [32], instead of an explicit start state. We adopt this shift in presentation to efficiently compose automata, avoiding the silent transitions in the original [34].

Definition 3.14. A CF-GKAT automaton with start dynamics $A \triangleq \langle S, \delta, \iota, \lambda \rangle$ consists of S, δ and λ as in a CF-GKAT automaton, in addition to a start dynamics $\iota \in G(S)$. The labeled family defined by a CF-GKAT automaton with start dynamics, also denoted $[\![A]\!]$, is simply given by $[\![A]\!]^\sharp = [\![\iota]\!]_A$ and $[\![A]\!]^\ell = [\![\lambda(\ell)]\!]_A$, where the right-hand sides are defined as for plain CF-GKAT automata.

It should be clear that CF-GKAT automata with start dynamics can easily be converted to plain CF-GKAT automata by adding a start state $\hat{s}$ that takes the behavior of the start dynamics ι:

$$\langle S, \delta, \iota, \lambda \rangle \mapsto \langle S + \hat{s}, \delta_\iota, \hat{s}, \lambda \rangle, \qquad \text{where} \qquad \delta_\iota(s, i, \alpha) \triangleq \begin{cases} \iota(i, \alpha) & \text{if } s = \hat{s} \\ \delta(s, i, \alpha) & \text{if } s \neq \hat{s} \end{cases} \tag{6}$$

Our construction turns a CF-GKAT expression e into a CF-GKAT automaton with start dynamics, which we call the *Thompson automaton for* e. The following paragraphs describe the construction and intuition behind each case in the construction. We make the simplifying assumption that each ℓ appears at most once in a subexpression of the form `label` ℓ. Moreover, we write ! for the unique function ! : $\emptyset \to X$ (for any set X), and in inductive cases we denote the Thompson automata for e_1 and e_2 by $A_1 \triangleq \langle S_1, \delta_1, \iota_1, \lambda_1 \rangle$ and $A_2 \triangleq \langle S_2, \delta_2, \iota_2, \lambda_2 \rangle$ respectively. Finally, to compress our notation, we will use the compact syntax of GKAT [32] for `if`-statements and `while`-loops:

$$e_1 +_b e_2 \triangleq \texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2, \qquad\qquad e_1^{(b)} \triangleq \texttt{while } b \texttt{ do } e_1.$$

Converting `break`, `return`, `goto` ℓ, *and indicator assignment:* recall that the semantics of `break`, `return`, `goto` ℓ, and indicator assignments simply emit the corresponding continuations. Thus, the non-trivial part of these Thompson automata are the start dynamics ι, which output said continuations. Because the start dynamics does not need to transition anywhere, there are no further states; because these programs also do not contain any `label` primitives, there is no need to assign a dynamics for them either. We thus construct the following automata:

$$\begin{array}{llll}
S_{\texttt{break}} \triangleq \emptyset & \delta_{\texttt{break}} \triangleq ! & \iota_{\texttt{break}}(i, \alpha) \triangleq \mathbf{brk}\ i & \lambda_{\texttt{break}}(\ell, i, \alpha) \triangleq \bot \\
S_{\texttt{return}} \triangleq \emptyset & \delta_{\texttt{return}} \triangleq ! & \iota_{\texttt{return}}(i, \alpha) \triangleq \mathbf{ret} & \lambda_{\texttt{return}}(\ell, i, \alpha) \triangleq \bot \\
S_{\texttt{goto}\ \ell} \triangleq \emptyset & \delta_{\texttt{goto}\ \ell} \triangleq ! & \iota_{\texttt{goto}\ \ell}(i, \alpha) \triangleq \mathbf{jmp}\ (\ell, i) & \lambda_{\texttt{goto}\ \ell}(\ell, i, \alpha) \triangleq \bot \\
S_{x:=i} \triangleq \emptyset & \delta_{x:=i} \triangleq ! & \iota_{x:=i}(i, \alpha) \triangleq \mathbf{acc}\ i & \lambda_{x:=i}(\ell, i, \alpha) \triangleq \bot
\end{array}$$

Converting tests and primitive actions: the conversions of primitive tests and primitive actions largely inherit the Thompson construction for GKAT. The Thompson automaton for tests `assert` b contains only a start dynamics, which accepts the input indicator-atom pairs if and only if they satisfy b. The Thompson's automata for primitive actions p contains a start dynamics that always

executing the action p before transitioning to the unique state, which accepts unconditionally.

$$S_{\texttt{assert}\ b} \triangleq \emptyset \qquad S_p \triangleq \{*\}$$
$$\delta_{\texttt{assert}\ b} \triangleq ! \qquad \delta_p(s,i,\alpha) \triangleq \mathbf{acc}\ i$$
$$\iota_{\texttt{assert}\ b}(i,\alpha) \triangleq \begin{cases} \mathbf{acc}\ i & (i,\alpha) \in [\![b]\!] \\ \bot & (i,\alpha) \notin [\![b]\!] \end{cases} \qquad \iota_p(i,\alpha) \triangleq (p,*,i)$$
$$\lambda_{\texttt{assert}\ b}(\ell,i,\alpha) \triangleq \bot \qquad \lambda_p(\ell,i,\alpha) \triangleq \bot$$

Converting labels: recall that `label` ℓ' is a does not affect the semantics from the start of the program, i.e., should be the same as the sequential identity `assert true`. However, the behavior of `label` ℓ' and `assert true` diverges when we consider the jump map: if $\ell = \ell'$, then executing `label` ℓ' starting from ℓ lets us reach the end of the program, whereas `assert true` does not contain any label. We thus end up with the following Thompson automaton for `label` ℓ':

$$S_{\texttt{label}\ \ell'} \triangleq \emptyset \quad \delta_{\texttt{label}\ \ell'} \triangleq ! \quad \iota_{\texttt{label}\ \ell'}(i,\alpha) \triangleq \mathbf{acc}\ i \quad \lambda_{\texttt{label}\ \ell'}(\ell,i,\alpha) \triangleq \begin{cases} \mathbf{acc}\ i & \ell = \ell' \\ \bot & \ell \neq \ell' \end{cases}$$

Converting `if` *statements:* the Thompson automaton for `if` b `then` e_1 `else` e_2 is also similar to that of GKAT: if the input indicator-atom pair satisfies (resp. falsifies) b, the start ι will enter the Thompson automaton of e_1 (resp. e_2) by taking on the behavior of ι_1 (resp. ι_2). The jump map λ assigns the entry point for label ℓ based on where ℓ appears: namely if ℓ appears in e_1, then ℓ will take its entry point in A_1; and similarly, if ℓ appears in e_2, ℓ will take its entry point in A_2.

$$S_{e_1 +_b e_2} \triangleq S_1 + S_2$$
$$\lambda_{e_1 +_b e_2}(\ell) \triangleq \begin{cases} \lambda_1(\ell) & (\texttt{label}\ \ell) \text{ appears in } e_1 \\ \lambda_2(\ell) & (\texttt{label}\ \ell) \text{ appears in } e_2 \\ \bot & \text{otherwise} \end{cases}$$
$$\delta_{e_1 +_b e_2}(s) \triangleq \begin{cases} \delta_1(s) & \text{if } s \in S_1 \\ \delta_2(s) & \text{if } s \in S_2 \end{cases}$$
$$\iota_{e_1 +_b e_2}(i,\alpha) \triangleq \begin{cases} \iota_1(i,\alpha) & (i,\alpha) \in [\![b]\!] \\ \iota_2(i,\alpha) & (i,\alpha) \notin [\![b]\!] \end{cases}$$

Converting Sequencing: Sequencing of automata can be defined by *uniform continuations* [32], which combine two dynamics $h_1, h_2 \in G(S)$ into a new dynamics $h_1[h_2]$. The latter acts like h_1 in almost all cases, except when h_1 accepts — then it will take on the behavior of h_2. In other words, $h_1[h_2]$ connects all the accepting transition of h_1 to h_2. Uniform continuation is typically used to compose two automata or add self-loops to an automaton; it can be formally defined as follows.

Definition 3.15 (Uniform Continuation). Let S be a set and given two dynamic $h_1, h_2 \in G(S)$, their *uniform continuation* is the dynamic $h_1[h_2] \in G(S)$, defined as follows:

$$h_1[h_2](i,\alpha) \triangleq \begin{cases} h_2(i',\alpha) & \text{if } h_1(i,\alpha) = \mathbf{acc}\ i' \\ h_1(i,\alpha) & \text{otherwise} \end{cases}$$

To construct the Thompson automaton for $e_1; e_2$, we will simply connect all the accepting transitions in A_1 to A_2 by applying uniform continuations on start dynamics ι_1, transitions δ_1, and jump map λ_1, while preserving the dynamics in A_2 — with the proviso that, if ℓ accepts in A_1, then

computation continues in A_2 via a uniform extension with ι_2. Formally, this works out as follows:

$$S_{e_1;e_2} \triangleq S_1 + S_2 \qquad \delta_{e_1;e_2}(s) \triangleq \begin{cases} \delta_1(s)[\iota_2] & \text{if } s \in S_1 \\ \delta_2(s) & \text{if } s \in S_2 \end{cases}$$

$$\iota_{e_1;e_2} \triangleq \iota_1[\iota_2] \qquad \lambda_{e_1;e_2}(\ell, i, \alpha) \triangleq \begin{cases} \lambda_1(\ell)[\iota_2] & (\texttt{label}\ \ell) \text{ appears in } e_1 \\ \lambda_2(\ell) & (\texttt{label}\ \ell) \text{ appears in } e_2 \\ \bot & \text{otherwise} \end{cases}$$

Converting `while` *loops:* Like GKAT automata, CF-GKAT automata require every transition between states to execute a primitive action. This presents a unique challenge in defining the start dynamics for while loops. Namely, a loop may not immediately encounter a primitive action on its first iteration, but this may trigger a change in indicator value that causes a primitive action to be executed in the *next* iteration, or the one after that, et cetera. This is reminiscent of jump resolution, where we also have to chase through some indirection to arrive at the right transition.

As a concrete example of this phenomenon, consider the following CF-GKAT program:

$$\begin{array}{l} \texttt{while true do } \{ \\ \quad \texttt{if } x = 0 \texttt{ then } x := 1 \\ \quad \texttt{else if } x = 1 \texttt{ then break} \\ \quad \texttt{else } \{ \texttt{ assert true } \} \ \} \end{array} \tag{7}$$

If this program starts with the indicator 0, then the first continuation (**brk** 1) will be encountered on the second iteration of the loop. Even worse, when starting with an indicator value like $x = 2$, the program will enter an infinite loop and never encounter a primitive action or continuation.

Fortunately, these difficulties can be resolved by the iter function (Definition 3.9): we repeat the start of the loop body until we encounter some productive behavior, or we get stuck.

Definition 3.16 (Iterated Start Dynamics). Let S be a set, let $h \in G(S)$, and $b \in \mathrm{BExp}$. We can use the iter function to define $h^b : G(S)$, as follows:

$$h^b \triangleq \mathrm{iter}\left((i, \alpha) \mapsto \begin{cases} \mathbf{exit}(\mathbf{acc}\ i) & \text{if } (i, \alpha) \notin [\![b]\!] \\ \mathbf{cont}(i', \alpha) & \text{if } (i, \alpha) \in [\![b]\!] \text{ and } h(i, \alpha) = \mathbf{acc}\ i' \\ \mathbf{exit}(h(i, \alpha)) & \text{otherwise} \end{cases} \right)$$

In the first case, the input (i, a) doesn't satisfy b, causing the while loop to terminate. In the second case, the loop body accepts (i, α) immediately and returns the exit indicator value i', thus the iteration of loop body will continue with (i', α). And the final case is reached when the program executes an action or encounters a non-local control, then the iteration can also be stopped.

Example 3.17 (Iterated Start Dynamics). Consider program 7 above with indicator set $\{0, 1, 2\}$, no primitive action, no label, and no primitive test. Then the only atom is $\emptyset$, and Thompson's automaton $A_1 \triangleq \langle S_1, \delta_1, \iota_1, \lambda_1 \rangle$ for the loop body can be computed to be as follows:

$$S_1 \triangleq \emptyset \qquad \begin{array}{l} \iota_1(0, \emptyset) \triangleq \mathbf{acc}\ 1 \\ \iota_1(1, \emptyset) \triangleq \mathbf{brk}\ 1 \\ \iota_1(2, \emptyset) \triangleq \mathbf{acc}\ 2 \end{array} \qquad \delta_1 \triangleq ! \qquad \lambda(\ell, i, \alpha) \triangleq \bot$$

Then we compute the iterated start dynamics ι^{true} with input $(0,\emptyset)$ and $(2,\emptyset)$:

$$\begin{aligned} {\iota_1}^{\mathsf{true}}(0,\emptyset) &= {\iota_1}^{\mathsf{true}}(1,\emptyset) && \text{because } (0,\emptyset) \in [\![\mathsf{true}]\!] \text{ and } \iota_1(0,\emptyset) = \mathbf{acc}\ 1 \\ &= \mathbf{brk}\ 1 && \text{because } \iota_1(1,\emptyset) = \mathbf{brk}\ 1 \\ {\iota_1}^{\mathsf{true}}(2,\emptyset) &= {\iota_1}^{\mathsf{true}}(2,\emptyset) && \text{because } (2,\emptyset) \in [\![\mathsf{true}]\!] \text{ and } \iota_1(2,\emptyset) = \mathbf{acc}\ 2 \\ &= \bot && \text{because the input } (2,\emptyset) \text{ is already explored} \end{aligned}$$

With the start dynamics defined, we still need to resolve structures within the loop body, like the break-continuation. To perform break-resolution, we extend the $\lfloor - \rfloor$ operator to dynamics.

Definition 3.18. Let S be a set, and let $h \in G(S)$. We define $\lfloor h \rfloor \in G(S)$ by lifting h via $\lfloor - \rfloor$ (c.f. Definition 2.13) when it returns a continuation, that is to say:

$$\lfloor h \rfloor(i,\alpha) = \begin{cases} \lfloor h(i,\alpha) \rfloor & \text{if } h(i,\alpha) \in C \\ h(i,\alpha) & \text{otherwise} \end{cases}$$

Finally, the transition function δ and jump map λ can be defined by first connecting δ_1 and λ_1 back to start dynamics ι, forming a loop in the automaton; then resolving the **brk** i continuations using $\lfloor - \rfloor$. Formally, the Thompson automaton for loop $e^{(b)}$ can then be defined as follows:

$$S_{e_1^{(b)}} \triangleq S_1 \qquad \delta_{e_1^{(b)}} \triangleq \lfloor \delta_1(s)[{\iota_1}^b] \rfloor \qquad \iota_{e_1^{(b)}} \triangleq \lfloor {\iota_1}^b \rfloor \qquad \lambda_{e_1^{(b)}}(\ell) \triangleq \lfloor \lambda(\ell)[{\iota_1}^b] \rfloor$$

Note how in the case of the jump map, we resolve **brk**-continuations after the uniform continuation with the iterated start dynamics. This way, a jump to a label inside the loop that hits a break statement immediately will continue executing execution after the loop, as is to be expected.

We present a summary of the Thompson's construction in **??**. The the full Thompson construction for CF-GKAT in mind, its correctness can now be stated as follows.

Theorem 3.19 (Thompson's construction preserves the continuation semantics). *Given an expression $e \in \mathrm{Exp}$, let A_e be the Thompson automaton for e, then $[\![e]\!] = [\![A_e]\!]$. More specifically, by unfolding the definition of continuation semantics, this boils down to the following:*

$$\forall i \in I, \ell \in \sharp + L, [\![e]\!]_i^\ell = [\![A_e]\!]_i^\ell$$

Proof sketch. By induction on e. This proof is somewhat tedious, but ultimately doable. The inductive cases are very similar to the ones for the Thompson construction in plain GKAT [32]; the only difference is that, this time, the semantics of the jump map needs to be taken into account. □

3.5 Algorithm, Completeness, and Complexity

With the definitions of lowering and Thompson's construction established, the decision procedure mostly follows. Nevertheless, it remains essential to define the alphabets Σ, T, I, and L, representing the set of primitive actions, primitive tests, indicator values, and labels, respectively. We may safely restrict primitive actions, primitive tests, and labels to those explicitly present in the expression, as expanding the alphabet beyond these will preserve (the equational theory of) the trace semantics. This trace is a property that can be validated by induction on the expression itself.

The set of indicator values, however, occupies a unique position. If the initial indicator value is absent from the program, the program's traces may diverge from traces starting from the present indicator values. Program 7 is one of the witnesses of this phenomenon: if the initial indicator value is 0 or 1, the program terminates; however, when starting from an indicator value that doesn't appear in the program, the program will loop indefinitely. An even simpler example is:

$$\mathsf{assert}\ (x = 1); p,$$

which executes p if the initial indicator value is 1, but rejects when started with any other indicator value, including one that is not present in the program. Fortunately, given an expression e, it is not hard to show that if neither i nor i' appears in e, then the behaviors for both values coincide, i.e.,

$$\forall \ell, [\![e]\!]_i^\ell = [\![e]\!]_{i'}^\ell.$$

Thus, when gathering the indicator values, it suffices to take those that appear explicitly in the program and augment this set with a fresh value $*$ that does not appear in the program.

We summarize our decision procedure as follows:

(1) Given two CF-GKAT programs e, f, we first collect their alphabets Σ, T, I, and L. We gather the sets of primitive actions Σ, primitive tests T, and labels L that are present in either e or f. Additionally, we identify the set of indicator values I, encompassing those found in either e or f, along with an additional indicator $*$ that is exclusive to both programs.

(2) We then proceed to compute the Thompson automata of e and f and convert them into CF-GKAT automata, denoted as A_e and A_f. It is noteworthy that these automata preserve the continuation semantics (Theorem 3.19). Formally,

$$[\![A_e]\!] = [\![e]\!] \qquad\qquad [\![A_f]\!] = [\![f]\!].$$

(3) Subsequently, we lower both A_e and A_f to GKAT automata $A_e{\downarrow_i}$ and $A_f{\downarrow_i}$ for each $i \in I$. By Theorem 3.13, the latter exhibits the same traces as e and f starting from i:

$$[\![A_e{\downarrow_i}]\!] = [\![A_e]\!]{\downarrow_i} = [\![e]\!]{\downarrow_i}, \qquad\qquad [\![A_f{\downarrow_i}]\!] = [\![A_f]\!]{\downarrow_i} = [\![f]\!]{\downarrow_i}\ .$$

(4) Finally, we run the equivalence algorithm for GKAT automata [32] on $A_e{\downarrow_i}$ and $A_f{\downarrow_i}$ for each $i \in I$, and return true when all the GKAT automata equivalence checks return true.

Soundness and completeness: The soundness and completeness of this algorithm now follow as a corollary of the corresponding properties for the decision procedure in GKAT. We denote the algorithm introduced above as $\mathrm{equiv}_{\mathsf{CFGKAT}}$, while the decision algorithm for equivalence of GKAT automata is denoted as $\mathrm{equiv}_{\mathsf{GKAT}}$. Thus, we establish the equivalence:

$$\begin{aligned}
\mathrm{equiv}_{\mathsf{CFGKAT}}(e, f) &\iff \forall i \in I, \mathrm{equiv}_{\mathsf{GKAT}}(A_e{\downarrow_i}, A_f{\downarrow_i}) \\
&\iff \forall i \in I, [\![A_e{\downarrow_i}]\!] = [\![A_f{\downarrow_i}]\!] \\
&\iff \forall i \in I, [\![e]\!]{\downarrow_i} = [\![f]\!]{\downarrow_i} \\
&\iff [\![e]\!]{\downarrow} = [\![f]\!]{\downarrow}\ .
\end{aligned}$$

Thus, e and f are trace equivalent if and only if $\mathrm{equiv}_{\mathsf{GKAT}}(e, f)$ returns true.

Algorithm complexity: We now give a rough account of the computational cost for deciding equivalence of CF-GKAT programs. Like GKAT, we consider T to be fixed for the purpose of analyzing complexity; otherwise, deciding equivalence of (CF-)GKAT programs is co-NP hard [32].

Starting with an expression e, we observe that the number of states in the Thompson automaton A_e is bounded by $|e|$, which is the size of e as a term. While computing this automaton we must bear in mind that deriving the iterated start dynamics of a loop using iter may take up to $|I|$ steps (assuming we use memoization to compute the output for each input), and this happens at most $|e|$ times. After lowering, each GKAT automaton $A_e{\downarrow}$ contains at most $|I| \times |e|$ states, and computing the jump resolution using iter takes on the order of $|L| \times |I|$ steps (again using memoization).

For each $i \in I$, determining the equivalence between $A_e{\downarrow_i}$ and $A_f{\downarrow_i}$ is accomplished in (nearly[2]) linear time relative to the number of states, so this check takes about $|I| \times (|e| + |f|)$ steps. To verify trace equivalence between e and f, this check is required for $A_e{\downarrow_i}$ and $A_f{\downarrow_i}$ across all $i \in I$.

Therefore, the overall time spent on the equivalence checks is on the order of $|I|^2 \times (|e| + |f|)$, while computing the automata can take roughly $|L| \times |I| + |I| \times (|e| + |f|)$ time. This implies that the algorithm's complexity scales (nearly) linearly with the sizes of e and f, linearly in the size of L, and quadratically with respect to the number of indicator values $|I|$.

4 CONTROL FLOW VERIFICATION

We hypothesize that CF-GKAT can be a useful tool to check whether two programs have the same control flow, i.e.: under the same circumstances, they execute the same sequence of primitive commands. An example use case could be to validate the *control flow structuring* stage of a decompiler. Briefly put, a decompiler is a program tasked with inferring a high-level language representation of a binary executable file. In earlier stages, the decompiler builds a *control-flow graph* from the binary [35], in which the vertices represent different blocks of instructions, and the edges encode how control may transfer from the end of one block to the beginning of the other. The control flow structuring pass is tasked with inferring an equivalent representation of this control flow graph in terms of constructs like `if-then-else` and `while-do`. A tool that validates the output of a control flow structuring algorithm could leverage CF-GKAT, by casting the control flow graph as a GKAT automaton, and comparing that to the GKAT automaton that corresponds to the inferred program.

Another use case would be to validate the correctness of refactoring operations aimed at making code more readable by eliminating or introducing early loop termination. In general, algorithms along these lines can never be *complete* with respect to input-output equivalences, as they cannot automatically validate the correctness of refactorings that introduce or eliminate primitive commands, even when this produces a functionally equivalent program. However, CF-GKAT should be applicable to refactoring operations that rearrange the code for the sake of improving the presentation of the control flow.

In this section, we test our hypotheses on the applicability and efficiency of CF-GKAT. We start by describing our proof of concept of the proposed decision procedure. Next, we report on a case study that reflects the two use cases described above.

4.1 Implementation

We target the C language as it is a widely used programming language that allows for non-local flow control. Our implementation is written in OCaml and can be divided into two parts:

- The *front-end* converts a function defined in a C file to a CF-GKAT expression. This conversion is based on the `clangML`[3] project which provides OCaml bindings for the `clang` compiler.[4] The conversion first determines which variables, if any, qualify as indicator variables and picks one variable from the set when possible. Next, it lifts the C syntax tree to a CF-GKAT program, mapping (1) all assignments and comparisons of the indicator variable to their CF-GKAT counterparts, (2) all other primitive statements to uninterpreted actions, and (3) all control flow constructs to the corresponding CF-GKAT operator whenever possible.
- The *back-end* is responsible for compiling a CF-GKAT expression down to a GKAT automaton, and for comparing two GKAT automata. This GKAT automaton is derived via the

[2]We omit the factor coming from the inverse Ackermann function $\hat{\alpha}(n)$, which is at most 5 for any realistic number of states, out of consideration for the sake of simplicity.

[3]https://memcad.gitlabpages.inria.fr/clangml/

[4]https://clang.llvm.org/

Thompson construction described in Section 3; the equivalence check for GKAT automata uses the bisimulation checking procedure proposed in [32].

The front-end and back-end combine into a tool that accepts two C files, and checks whether the functions defined therein (paired by their name) have the same control flow. Given that not all C programs can be faithfully converted to CF-KAT expressions, our front-end does not aim to be complete; instead, we aimed to support the transformation of a sizeable portion of the code in the GNU `coreutils` project to perform the experiments described in the next section.

Remark. In particular, our front-end internally converts `for`-loops to `while`-loops, and `do-while` loops to (partially unrolled) `while` loops. The former transformation is sound; the latter is admissible when the loop body does not contain `break` or a label, which is the case for our experiment below.

4.2 A Case Study From GNU `coreutils`

We used GNU `coreutils` as a source of C code containing non-local control flow. As is customary with projects of this size, the code is organized into several files that collectively define thousands of functions, ranging from very simple to very complex, that implement utilities commonly found in POSIX systems. Throughout this section, we will use the function `mp_factor_using_pollard_rho` in `factor.c` as an example of the validation machinery we built on top of CF-GKAT. The code in Figure 1a shows an abridged version of this function in `coreutils` version 9.5. We now discuss the two transformations we targeted as applications that can be validated via CF-GKAT.

Compilation-decompilation. We hypothesize that the theory behind CF-GKAT should be usable to validate the output of control flow structuring algorithms in decompilers. To fully test this hypothesis, we would need to have access to the internal representations used before and after control flow structuring, and convert those to GKAT automata and CF-GKAT expressions, respectively. Unfortunately, doing this would entail a substantial engineering effort, which would go beyond the scope of this project. As a more feasible but slightly less rigorous benchmark, we opted to *compile* C code to x86 binary code, and then *decompile* the result using Ghidra.[5] This transformation can be implemented without modifying existing codebases, and should still give some insight into decompiler correctness in that it lets us compare the decompiled source to the original.

However, we immediately face the challenge of pairing the primitive actions from the decompiled code to their corresponding actions in the source code. For instance, a primitive action `i += 1` in the source code can be decompiled to `i++` in certain contexts. Detecting *all* such transformations would require a large engineering effort. To address this, we make the compiler *blind* to the nature of the primitive actions and primitive tests by replacing primitive actions and primitive tests with calls to new functions `void pact(int)` and `bool pbool(int)`, respectively. The parameter distinguishes the primitives, and the correspondence in the decompiled code can be inferred from it. Figure 1b shows the (abbreviated) *blinded* version of our case study function. Note that this transformation does not alter control flow, but the blinded code can be longer, as the blinder also expands preprocessor macros. Crucially, blinding depends on indicator variable detection, since indicator tests and assignments need to remain in the blinded code.

In this experiment, we utilize Ghidra as our decompiler of choice, and clang as our compiler. The C code obtained from compiling and then decompiling the blinded code (Figure 1b) is shown in Figure 2a. We have manually removed some decompilation artifacts, for example the expression `(pbool(n)& 1) != 0` is simplified to `pbool(n)` in conditional expressions. The code produced by Ghidra is markedly different from the source—it has 3 more labels (and 3 additional corresponding

[5] https://ghidra-sre.org/

```
static void mp_factor_using_pollard_rho(...) {
mpz_t x, z, y, P;
mpz_t t, t2;
devmsg("...", a);
...
while (mpz_cmp_ui(n, 1) != 0) {
  for (;;) {
    do {
      mpz_mul(t, x, x);
      ...
      if (k % 32 == 1) {
        mpz_gcd(t, P, n);
        if (mpz_cmp_ui(t, 1) != 0)
          goto factor_found;
        mpz_set(y, x);
      }
    } while (--k != 0);
    mpz_set(z, x);
    ...
    for (unsigned long long int i = 0; i < k; i
         ++) {
      mpz_mul(t, x, x);
      ...
    }
    mpz_set(y, x);
  }
factor_found:
  do {
    mpz_mul(t, y, y);
    ...
  } while (mpz_cmp_ui(t, 1) == 0);
  mpz_divexact(n, n, t);
  if (!mp_prime_p(t)) {
    devmsg("...");
    ...
  } else {
    mp_factor_insert(factors, t);
  }
  if (mp_prime_p(n)) {
    mp_factor_insert(factors, n);
    break;
  }
  mpz_mod(x, x, n);
  ...
}
mpz_clears(P, t2, t, z, x, y, nullptr);
}
```

(a) Original code of the function.

```
void mp_factor_using_pollard_rho() {
  pact(197);
  ...
  do {
    if (pbool(83)) {
      pact(194);
    }

  } while (0);
  pact(193);
  ...
  while (pbool(83)) {
    for (;;) {
      do {
        pact(176);
        ...
        if (pbool(183)) {
          pact(182);
          if (pbool(83)) {
            goto factor_found;
          }
          pact(173);
        }

      } while (pbool(181));
      pact(180);
      ...
      for (pact(177); pbool(138); pbool(59)) {
        pact(176);
        ...
      }
      pact(173);
    }
  factor_found:
    do {
      pact(172);
      ...
    } while (pbool(83));
    pact(166);
    if (!pbool(165)) {
      do {
        if (pbool(83)) {
          pact(164);
        }
      } while (0);
      pact(163);
    } else {
      pact(161);
    }
    if (pbool(160)) {
      pact(159);
      break;
    }
    pact(158);
    ...
  }
  pact(155);
}
```

(b) "Blinded" code of the function.

Fig. 1. Different versions of `mp_factor_using_pollard_rho` in `factor.c`, part of GNU `coreutils`.

```
void mp_factor_using_pollard_rho(void) {
  pact(0xc5);
  ...
  if (pbool(0x53)) {
    pact(0xc2);
  }
  pact(0xc1);
  ...
LAB_00100a0c:
  if (pbool(0x53)) {
  LAB_00100a2d:
    pact(0xb0);
    ...
    if (pbool(0xb7)) {
      pact(0xb6);
      if (pbool(0x53))
        goto LAB_00100b47;
      pact(0xad);
    }
    if (!pbool(0xb5)) {
      pact(0xb4);
      ...
      while (pbool(0x8a)) {
        pact(0xb0);
        ...
      }
      pact(0xad);
    }
    goto LAB_00100a2d;
  }
LAB_00100c34:
  pact(0x9b);
  return;
LAB_00100b47:
  do {
    pact(0xac);
    ...
  } while (pbool(0x53));
  pact(0xa6);
  if (!pbool(0xa5)) {
    if (pbool(0x53)) {
      pact(0xa4);
    }
    pact(0xa3);
  } else {
    pact(0xa1);
  }
  if (pbool(0xa0)) {
    pact(0x9f);
    goto LAB_00100c34;
  }
  pact(0x9e);
  ...
  goto LAB_00100a0c;
}
```

(a) Code of the Ghidra's decompilation of the function.

```
void mp_factor_using_pollard_rho() {
  int factor_found = 0;
  pact(197);
  ...
  do {
    if (pbool(83)) {
      pact(194);
    }
  } while (0);
  pact(193);
  ...
  while (pbool(83)) {
    for (; factor_found == 0;) {
      do {
        pact(176);
        ...
        if (pbool(183)) {
          pact(182);
          if (pbool(83)) {
            factor_found = 1;
          }
          if (factor_found == 0)
            pact(173);
        }
      } while ((factor_found == 0) && pbool(181));
      if (factor_found == 0) {
        pact(180);
        ...
        for (pact(177); pbool(138); pbool(59)) {
          pact(176);
          ...
        }
        pact(173);
      }
    }
    factor_found = 0;
    do {
      pact(172);
      ...
    } while (pbool(83));
    pact(166);
    if (!pbool(165)) {
      do {
        if (pbool(83)) {
          pact(164);
        }
      } while (0);
      pact(163);
    } else {
      pact(161);
    }
    if (pbool(160)) {
      pact(159);
      break;
    }
    pact(158);
    ...
  }
  pact(155);
}
```

(b) Code of the function with gotos removed by Calipso.

Fig. 2. Different versions of the function `mp_factor_using_pollard_rho` in `factor.c`, part of GNU coreutils.

gotos)—yet our implementation is able to validate, in a fraction of a second[6], that this output is equivalent to the original.

Goto-elimination. In general, `goto` statements can be eliminated from C code by introducing additional indicator variables to guide the control flow [5, 13, 36]. This is the idea that underpins a classic `goto`-elimination algorithm proposed by Erosa and Hendren [13]. Calipso [7][7] provides an improved implementation of their algorithm.

We ran Calipso on the blinded code in Figure 1b, and manually adjusted the output to change instances where the newly-introduced variable `factor_found` is used as a Boolean (as integers and Booleans are indistinguishable in C) into proper comparisons (e.g., `if (factor_found == 0)`). Our tool confirms, again in a fraction of a second, that the code thus obtained is equivalent to the blinded input. Note that indicator detection is crucial: if `factor_found` is not detected as an indicator, its assignments and tests will be converted into primitive actions and tests respectively. Thus, the output by Calipso will be fundamentally different from the blinded source, as it contains at least one primitive action or test that is not present in the input.

Overall, we consider our results quite promising: despite the fact that our current prototype does not support C constructs like the `switch` statement, we are able to blind 836 of the approximately 1.4K functions in `coreutils` for our experiments. A relatively simple program transformation would allow us to support `switch` statements when the expression being switched on is a variable. However, supporting `switch` statements in general still requires notable additions to our theory. Support for other constructs, e.g., properly supporting `do-while` loops and providing robust support for the conditional (ternary) operator would also require extensions to our theory, but it is quite possible to embed special cases in the current theory of CF-GKAT.

5 RELATED WORK

5.1 (De)compiler Verification and Validation

The verification of (de)compilation is a well-developed area with many existing techniques. Each of these techniques have its unique trade-offs.

One approach involves fully formally verifying the (de)compilation algorithm [12, 28]. This method offers a formal guarantee of correctness without imposing runtime overhead. However, its implementation is often language-specific; and the techniques required to design fully-verified (de)compiler can be challenging to adapt on a large scale, due to the steep learning curve associated with programming in proof assistants.

Another approach is compiler testing [8, 27, 37], which feeds synthesized programs into different compilers, and checks whether the compilation results produce the same output given the same input. While this approach is accessible with automated tools [37], it provides no formal guarantee of correctness, and may struggle with complex semantics, such as probabilistic computation, which is supported by GKAT [32].

Similar to our approach, translation validation [15, 17, 18, 29, 38] employs bisimulation-based techniques, offers formal guarantees of equivalences, and provide approachable automation tools. However, the primary goal of translation validation is to provide end-to-end verification of compiler transformations, whereas our project focuses on verifying control-flow transformations. This difference in focus inspires our distinct trade-offs. Specifically, we work with a minimal and general language that is sufficient to model well-studied control-flow transformation algorithms [5, 13, 16, 36]. This minimal language also enables us to establish desirable theorems, such as completeness, decidability, low complexity, and the correspondence between denotational and

[6]The experiments ran on a system with an Intel Core i7-9750H CPU @ 2.60GHz and 16 GB of RAM.
[7]https://github.com/BinaryAnalysisPlatform/FrontC

operational semantics; many of these properties are unobtainable in the setting of translation-validation research. Finally, the algebraic nature of our foundation, guarded Kleene algebra with tests, provides the possibility of equational verification in this domain.

5.2 Kleene Algebra And Control Flow Verification

Existing work has explored non-local control-flow structures and indicator variables within the framework of KAT, albeit with a number of differences from our work.

Kozen characterized the semantics for programs with non-local control-flow structures as a family of KAT expressions [20]. This approach yields a decision procedure for program equivalences, by reducing them to KAT equivalences. In contrast, CF-GKAT takes a more explicit approach by defining the continuation semantics, and the equivalence is computed by converting programs directly into automata. Our method closed an open question by Kozen [20], on whether non-local control flow structures can be treated "directly", without being converted into KAT expressions.

Grathwohl et al. [16] proposed *KAT+B!* an extension of KAT with "mutable tests", which can be regarded as indicator variable where there are only two possible indicator values $I = \{\texttt{true}, \texttt{false}\}$. Concretely, their setter $b!$ equates to indicator assignment $x = \texttt{true}$ and $\bar{b}!$ to $x = \texttt{false}$; similarly, their tester $b?$ corresponds to primitive indicator tests. Although this is a special case of indicator variable, KAT+B! can simulate indicator variable over a finite set I with $|I|$ mutable tests. For example, indicator assignments $x := i$ can be simulated by $b_i!; \Pi_{i' \neq i}(\overline{b_{i'}}!)$, where each mutable test b_i records whether the indicator variable x is assigned to i. We opt to treat indicator assignments and tests as primitives, rather than restricting ourselves to (boolean-valued) mutable tests.

Our treatment of indicator variables also draw inspirations from NetKAT [2]. Specifically, NetKAT can be seen as a special case of KAT with indicator variables, the only primitive action is dup.

To reason about variable assignment beyond indicator variables, Schematic KAT [3] provides a fine-grained algebraic theory for assignments over uninterpreted functions. Later, Schematic KAT was also extended to reason about local variables [1]. Neither work covers the complexity of the equivalence problem for schematic KAT and its extensions. Kleene algebra can also be extended with nominal techniques [14, 21, 22], which may help to reason about potentially infinite data domains, although the inclusion of tests to nominal Kleene algebra has not yet been investigated.

Separately, Kozen and Patron [23] have applied KAT to verify compiler correctness. Their system directly uses postulated equalities for parts of their verification task. In contrast, our framework is based on the trace semantics, a commonly accepted semantics for `while`-programs.

5.3 Complexity And Expressivity

Finally, unlike the systems above, our system is based on GKAT, instead of KAT. This enhances the scalability of our equivalence checking algorithms to accommodate larger programs: whereas equivalence of KAT expressions is PSPACE-complete [10], equivalence of CF-GKAT expressions can be verified in polynomial time for a fixed number of tests [32]. Symbolic techniques previously applied to KAT may also provide better ways of mitigating the complexity of tests [30].

Our system is the first to integrate both indicator variables and non-local control flow in a unified framework, which enables the verification of complex control-flow transformations that leverage both indicator variables and non-local control flow [36]. Our notion of trace equivalence is also coarser than previous systems; CF-GKAT equates programs ending in different indicator values:

$$x := \texttt{true}; \texttt{if } x = \texttt{true} \texttt{ then print(1) else print(2)}$$
$$x := \texttt{false}; \texttt{if } x = \texttt{false} \texttt{ then print(1) else print(2)}$$

The above two programs are equivalent to an outside observer, as both of them will print 1; yet the assignment of x is different at the end of the program, thus previous systems like KAT+B! [16] will

not be able to equate these two programs. As a trade-off, our equivalence is not a congruence. For example, equivalence is not preserved under sequencing with $\texttt{assert}\ (x = 1)$:

$$(x := 1) \equiv (x := 0)$$

$$(x := 1); \texttt{assert}\ (x = 1) \equiv \texttt{assert}\ \texttt{true} \not\equiv \texttt{assert}\ \texttt{false} \equiv (x := 0); \texttt{assert}\ (x = 1).$$

Intuitively, $(x := 1) \equiv (x := 0)$ because we disregard the different ending indicator values 1 and 0. However, this distinction can be observed by sequencing both programs with the assertion $\texttt{assert}\ (x = 1)$: if the the indicator value is 1, then the assertion will not have any observable effect; if the indicator value is 0, the assertion will reject all the previous traces. Thus sequencing an assertion after two equivalent programs will not necessarily preserve the equivalence.

6 CONCLUSION AND FUTURE WORK

In this paper, we introduced CF-GKAT (Control Flow GKAT), a system that extends GKAT with non-local control flow and indicator variables to validate control flow transformation programs. We formalized two semantics for CF-GKAT. The first is the continuation semantics, where each trace, represented as a guarded word, is augmented with a continuation. The second is the trace semantics, which is obtained by resolving the continuations in the continuation semantics.

We proposed CF-GKAT automata as the operational model for CF-GKAT programs, where the operational semantics is obtained by the Thompson's construction [32, 34]. Concretely, Thompson's construction for CF-GKAT turns every CF-GKAT program into a CF-GKAT automaton, while preserving its continuation semantics. This conversion allows us to design an efficient equivalence checker for CF-GKAT programs, by lowering their Thompson automata into GKAT automata. Notably, the complexity of this decision procedure scales nearly linearly with respect to the size of the input CF-GKAT programs. Thus, our work provides an efficient validation algorithm for various control flow transformations that utilize indicator variables and non-local control flow [13, 36].

While we successfully addressed one of Kozen's questions [20] by presenting an algorithm to directly convert CF-GKAT programs into automata, we have yet to develop a coalgebraic perspective on non-local control flow utilizing Brzozowski derivatives [6]. Such an approach could streamline several proofs, such as trace preservation of the lowering (Theorem 3.13) and the correctness of the operational semantics (Theorem 3.19), and lead to a memory-efficient on-the-fly algorithm for trace equivalences between CF-GKAT programs. A coalgebraic checker could also make use of symbolic techniques [30] to prevent explicit calculations based on the atoms of a Boolean algebra.

Additional future work could be the inclusion of the `continue` command within loops, as well as other types of control flow found in modern programming languages such as `do-while` and `switch`. In terms of the theory's extensibility, it would be beneficial to separate the treatment of indicator variables and non-local control. Currently, both components are integrated into the CF-GKAT automata signature as a unified entity. While this approach provides a compact definition of operational semantics, it also introduces complexities when incorporating other non-local controls like `continue`. Specifically, we will need to pass the indicator value in the continuation for `continue`, despite none of the non-local control-flow structures changing the indicator variable.

REFERENCES

[1] Kamal Aboul-Hosn and Dexter Kozen. 2008. Local Variable Scoping and Kleene Algebra with Tests. *The Journal of Logic and Algebraic Programming* 76, 1 (May 2008), 3–17. https://doi.org/10.1016/j.jlap.2007.10.007

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. *ACM SIGPLAN Notices* 49, 1 (Jan. 2014), 113–126. https://doi.org/10.1145/2578855.2535862

[3] Allegra Angus and Dexter Kozen. 2001. *Kleene Algebra with Tests and Program Schematology.* Technical Report. Cornell University, USA.

[4] Arthur Azevedo de Amorim, Cheng Zhang, and Marco Gaboardi. 2024. Kleene Algebra with Commutativity Conditions Is Undecidable. (April 2024).

[5] Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (1966), 366–371. https://doi.org/10.1145/355592.365646

[6] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

[7] Hugues Cassé, Louis Féraud, Christine Rochange, and Pascal Sainrat. 2002. Une approche pour réduire la complexité du flot de contrôle dans les programmes C. *Tech. Sci. Informatiques* 21, 7 (2002), 1009–1032. http://tsi.revuesonline.com/article.jsp?articleId=3831

[8] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (Jan. 2021), 1–36. https://doi.org/10.1145/3363562

[9] Cristina Cifuentes. 1994. *Reverse compilation techniques.* Ph. D. Dissertation. Queensland University of Technology.

[10] Ernie Cohen, Dexter Kozen, and Frederick Smith. 1996. *The Complexity of Kleene Algebra with Tests.* Technical Report TR96-1598.

[11] Coq Development Team. 2022. *The Coq Reference Manual, version 8.15.* Available electronically at http://coq.inria.fr/doc.

[12] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable Validation of Binary Lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, London UK, 655–671. https://doi.org/10.1145/3385412.3385964

[13] Ana M. Erosa and Laurie J. Hendren. 1994. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *ICCL*. IEEE Computer Society, 229–240. https://doi.org/10.1109/ICCL.1994.288377

[14] Murdoch J. Gabbay and Vincenzo Ciancia. 2011. Freshness and Name-Restriction in Sets of Traces with Names. In *Foundations of Software Science and Computational Structures*, Martin Hofmann (Ed.). Springer, Berlin, Heidelberg, 365–380. https://doi.org/10.1007/978-3-642-19805-2_25

[15] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. 2005. Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers. *Electronic Notes in Theoretical Computer Science* 132, 1 (May 2005), 53–71. https://doi.org/10.1016/j.entcs.2005.01.030

[16] Niels Bjørn Bugge Grathwohl, Dexter Kozen, and Konstantinos Mamouras. 2014. KAT + B!. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2603088.2603095

[17] Theodoros Kasampalis. 2021. *Translation Validation for Compilation Verification.* Thesis. University of Illinois at Urbana-Champaign.

[18] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 1004–1019. https://doi.org/10.1145/3445814.3446751

[19] Dexter Kozen. 1996. Kleene Algebra with Tests and Commutativity Conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Tiziana Margaria, and Bernhard Steffen (Eds.). Vol. 1055. Springer Berlin Heidelberg, Berlin, Heidelberg, 14–33. https://doi.org/10.1007/3-540-61042-1_35

[20] Dexter Kozen. 2008. Nonlocal Flow of Control and Kleene Algebra with Tests. *2008 23rd Annual IEEE Symposium on Logic in Computer Science* (June 2008), 105–117. https://doi.org/10.1109/LICS.2008.32

[21] Dexter Kozen, Konstantinos Mamouras, Daniela Petrişan, and Alexandra Silva. 2015. Nominal Kleene Coalgebra. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Vol. 9135. Springer Berlin Heidelberg, Berlin, Heidelberg, 286–298. https://doi.org/10.1007/978-3-662-47666-6_23

[22] Dexter Kozen, Konstantinos Mamouras, and Alexandra Silva. 2015. Completeness and Incompleteness in Nominal Kleene Algebra. In *Relational and Algebraic Methods in Computer Science*, Wolfram Kahl, Michael Winter, and José Oliveira (Eds.). Springer International Publishing, Cham, 51–66. https://doi.org/10.1007/978-3-319-24704-5_4

[23] Dexter Kozen and Maria-Cristina Patron. 2000. Certification of Compiler Optimizations Using Kleene Algebra with Tests. In *Computational Logic — CL 2000 (Lecture Notes in Computer Science)*, John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey (Eds.). Springer, Berlin, Heidelberg, 568–582. https://doi.org/10.1007/3-540-44957-4_38

[24] Dexter Kozen and Frederick Smith. 1997. Kleene Algebra with Tests: Completeness and Decidability. In *Computer Science Logic*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Dirk Dalen, and Marc Bezem (Eds.). Vol. 1258. Springer Berlin Heidelberg, Berlin, Heidelberg, 244–259. https://doi.org/10.1007/3-540-63172-0_43

[25] Dexter Kozen and Wei-Lung Dustin Tseng. 2008. The Böhm–Jacopini Theorem Is False, Propositionally. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Vol. 5133. Springer Berlin Heidelberg, Berlin, Heidelberg, 177–192. https://doi.org/10.1007/978-3-540-70594-9_11

[26] Stepan L. Kuznetsov. 2023. On the Complexity of Reasoning in Kleene Algebra with Commutativity Conditions. In *Theoretical Aspects of Computing – ICTAC 2023*, Erika Ábrahám, Clemens Dubslaff, and Silvia Lizeth Tapia Tarifa (Eds.). Springer Nature Switzerland, Cham, 83–99. https://doi.org/10.1007/978-3-031-47963-2_7

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Edinburgh United Kingdom, 216–226. https://doi.org/10.1145/2594291.2594334

[28] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[29] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. *SIGPLAN Not.* 35, 5 (May 2000), 83–94. https://doi.org/10.1145/358438.349314

[30] Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. In *POPL*. 357–368. https://doi.org/10.1145/2676726.2677007

[31] Todd Schmid, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2021. Guarded Kleene Algebra with Tests: Coequations, Coinduction, and Completeness. https://doi.org/10.4230/LIPIcs.ICALP.2021.142 arXiv:2102.08286 [cs]

[32] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2020. Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–28. https://doi.org/10.1145/3371129

[33] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225. https://doi.org/10.1145/321879.321884

[34] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387

[35] Freek Verbeek, Joshua A. Bockenek, Zhoulai Fu, and Binoy Ravindran. 2022. Formally verified lifting of C-compiled x86-64 binaries. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 934–949. https://doi.org/10.1145/3519939.3523702

[36] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2015.23185

[37] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294. https://doi.org/10.1145/1993316.1993532

[38] Yiji Zhang and Lenore D. Zuck. 2018. Formal Verification of Optimizing Compilers. In *Distributed Computing and Internet Technology*, Atul Negi, Raj Bhatnagar, and Laxmi Parida (Eds.). Springer International Publishing, Cham, 50–65. https://doi.org/10.1007/978-3-319-72344-0_3