Checking Equivalence in a Non-strict Language

JOHN C. KOLESAR, Yale University, USA
RUZICA PISKAC, Yale University, USA
WILLIAM T. HALLAHAN, Binghamton University, USA

Program equivalence checking is the task of confirming that two programs have the same behavior on corresponding inputs. We develop a calculus based on symbolic execution and coinduction to check the equivalence of programs in a non-strict functional language. Additionally, we show that our calculus can be used to derive counterexamples for pairs of inequivalent programs, including counterexamples that arise from non-termination. We describe a fully automated approach for finding both equivalence proofs and counterexamples. Our implementation, NEBULA, proves equivalences of programs written in Haskell. We demonstrate NEBULA's practical effectiveness at both proving equivalence and producing counterexamples automatically by applying NEBULA to existing benchmark properties.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Program verification**.

Additional Key Words and Phrases: coinduction, non-strictness, equivalence, symbolic execution, Haskell

ACM Reference Format:
John C. Kolesar, Ruzica Piskac, and William T. Hallahan. 2022. Checking Equivalence in a Non-strict Language. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 177 (October 2022), 28 pages. https://doi.org/10.1145/3563340

1 INTRODUCTION

Equivalence checking is the task of verifying that two programs behave identically when given identical inputs. Equivalence checking is useful for a number of tasks, such as ensuring compiler optimizations' correctness [Benton 2004; Peyton Jones et al. 2001; Peyton Jones 1996]. Optimizing compilers aim to improve the performance of code with simplifying transformations. Critically, these transformations must preserve the meaning of the code, or they could lead to incorrect behavior that violates the language specification. Equivalence checking has other uses as well, such as ensuring the correctness of refactored code [Schuts et al. 2016], program synthesis [Campbell et al. 2021; Schkufza et al. 2013; Smith and Albarghouthi 2019], and automatic evaluation of students' submissions for programming assignments [Milovancevic et al. 2021].

Non-strict languages allow for the use of conceptually infinite data strutures. Such structures have a number of uses, from memoization [Elliot 2010] to trees representing all moves in an infinite game. Many seemingly obvious equivalences do not hold when we allow infinite data structures. Consider, for instance, subtraction for natural numbers:

```
                                        Z     - _     = Z
data Nat = S Nat | Z                    x     - Z     = x
                                        (S x) - (S y) = x - y
```

Authors' addresses: John C. Kolesar, Computer Science, Yale University, USA, john.kolesar@yale.edu; Ruzica Piskac, Computer Science, Yale University, USA, ruzica.piskac@yale.edu; William T. Hallahan, Computer Science, Binghamton University, USA, whallahan@binghamton.edu.

2475-1421/2022/10-ART177
https://doi.org/10.1145/3563340

One might expect `m - m` to reduce to `Z` for any natural number `m`, but this equivalence does not always hold. With non-strictness, one can define a conceptually infinite `Nat` as `inf = S inf`, and the evaluation of `inf - inf` does not terminate.

We describe the first—to the best of our knowledge—automated equivalence checker for programs in a *non-strict functional language*. Existing approaches for fully automated equivalence checking [Claessen et al. 2012; Dixon and Fleuriot 2003; Farina et al. 2019; Sonnex et al. 2012] assume total and finite input values. In contrast, our approach checks that two programs display the same behavior even when applied to inputs that include infinite or diverging sub-expressions.

Our equivalence checking approach is based on symbolic execution and the principle of coinduction. Symbolic execution is a method for exploring the execution paths of a program exhaustively. Coinduction is a proof technique for deriving conclusions about infinite data structures from cyclic patterns in their behavior. We define a notion of equivalence for a non-strict functional language that incorporates non-total expressions and the possibility of expressions being equivalent by both failing to terminate. We develop a calculus for coinduction and symbolic execution capable of proving equivalence of programs in the non-strict functional language. This calculus also incorporates a sound approach for using auxiliary equivalence lemmas that allow a sub-expression e_1 to be rewritten as an equivalent expression e_2. We show that, while such lemma applications are actually *unsound* in general, they can be used soundly under certain conditions.

In addition to proving equivalence, our approach finds counterexamples that demonstrate the inequivalence of two programs. Our approach can detect not only inequivalences that arise from two programs terminating with different values, but also inequivalences that arise from one program terminating and the other failing to terminate when given the same inputs.

We show that the combination of symbolic execution and coinduction-based tactics allows for *automated* equivalence checking and inequivalence detection. Our algorithm switches between symbolic execution and coinduction automatically to find proofs. Further, we describe an extension of this algorithm that generates and proves helper lemmas automatically.

We implement our approach in NEBULA (Non-strict Equivalence By Using Lemmas and Approximation), a practical tool targeting Haskell code. NEBULA builds on the Haskell symbolic execution engine G2 [Hallahan et al. 2019], and it uses coinduction for automated equivalence checking of higher-order functional programs. Our evaluation demonstrates that NEBULA is capable of both verifying true properties and finding counterexamples for false properties. In particular, we run NEBULA on the Zeno test suite [Sonnex et al. 2012]. As this test suite was developed assuming strict semantics, most of the properties do not hold with non-strict semantics. We verify 92% of the properties that are still true in a non-strict context (i.e. 26% of the entire suite, where 28% of the suite is still true), and we find counterexamples for every property that no longer holds (72% of the suite.) Furthermore, we evaluate NEBULA's ability to identify counterexamples involving non-termination and find that our tool can generate such counterexamples for 73% of the applicable benchmarks. We describe an approach for accommodating total and finite inputs in NEBULA and evaluate NEBULA on altered versions of the Zeno properties that hold even under non-strictness.

In summary, our contributions are the following:

1. Equivalence Checking Calculus Section 3 provides an overview of our formalization of symbolic execution. In Section 4, we develop a calculus combining symbolic execution and coinduction to prove equivalence of non-strict functional programs, and prove the calculus sound.

2. Producing Counterexamples In Section 5, we extend the calculus to produce counterexamples, including counterexamples that demonstrate inequivalence due to differences in termination.

```
prop33_lhs a b = min a b === a         -- full === definition not shown
prop33_rhs a b = a <= b                (S x) === (S y) = x === y

min Z     y     = Z                    Z     <= _     = True
min (S x) Z     = Z                    _     <= Z     = False
min (S x) (S y) = S (min x y)          (S x) <= (S y) = x <= y
```

Fig. 1. Zeno Theorem 33

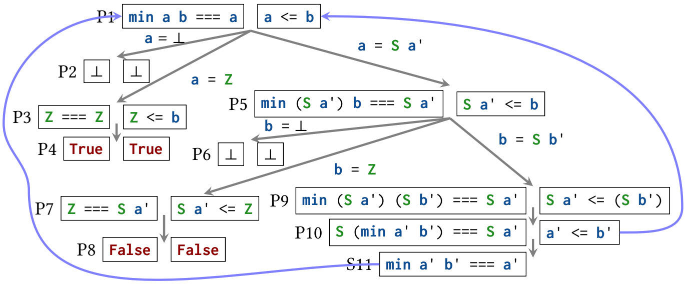

Fig. 2. Overview of how NEBULA proves `prop33`. Gray arrows denote symbolic execution, and blue arrows denote coinduction.

3. Automation Techniques Section 6 introduces an algorithm that searches for both equivalence proofs and counterexamples automatically, guided by symbolic execution and coinduction. Our algorithm also discovers and proves helper lemmas automatically to aid in the verification process.

4. Implementation and Evaluation Finally, in Section 7, we discuss our implementation, NEBULA, that checks equivalence of Haskell expressions. We demonstrate our technique's effectiveness at both proving equivalences and producing counterexamples on benchmarks adapted from existing sources.

For reasons of space, proofs are deferred to the Appendix, available at https://johnckolesar.github.io/files/checking-equivalence.pdf.

2 MOTIVATING EXAMPLES

We present three examples to show how NEBULA proves properties and finds counterexamples.

Example 2.1. Our first example is the property `prop33` taken from the Zeno evaluation suite [Sonnex et al. 2012], which is a Haskell translation of the IsaPlanner evaluation suite [Johansson et al. 2010]. The example is given in Figure 1. Consider the functions `prop33_lhs` and `prop33_rhs`: `prop33_lhs` finds the minimum of two numbers `a` and `b`, and returns whether that minimum value is equal to `a`, while `prop33_rhs` uses <= to check directly whether `a` is less than or equal to `b`. NEBULA can prove the equivalence of `prop33_lhs` and `prop33_rhs` automatically. The equivalence means that evaluating `prop33_lhs` and `prop33_rhs` on any inputs `a` and `b`, including inputs that are infinite or non-total, will produce the same output.

Figure 2 depicts the proof structure that NEBULA uses to prove the equivalence of `prop33_lhs` and `prop33_rhs`. To simplify the presentation, we first explain how the proof obligations are discharged, and then we discuss how the proof is actually derived. In the proof tree, each step P_i consists of two expressions that need to be proven equivalent.

We start with *P1*, representing the two initial expressions, `min a b === a` and `a <= b`. Note that `a` and `b` are *symbolic variables*: it is known that they are of type `Nat`, but their exact values are unknown. We use *symbolic execution* to evaluate these expressions. Evaluating === requires evaluating `min a b` first, which, in turn, requires knowing the value of `a`. To address these requirements, we need to consider all the values that `a` can take, so we split into multiple branches. On each branch, we assign a different value to `a`. In *P3* we concretize `a` to `Z`, in *P5* we concretize `a` to `S a'`, where `a'` is a fresh symbolic variable, and in *P2*, we concretize `a` to ⊥, a special value representing the possibility that `a` either produces an error or does not terminate when evaluated. Each branch symbolically executes `a <= b` with its concretization of `a`. Step *P2* leads to the expression `⊥ <= b` evaluating to ⊥. We conclude trivially that the expressions in *P2* are equivalent, due to their syntactic equality. In the case of *P3*, we have the states `Z === Z` and `Z <= b`. Symbolic execution will reduce both states to `True`, as shown in *P4*, allowing us again to conclude that the expressions are equivalent.

Step *P5* is a more interesting case: we must show that `min (S a') b === S a'` is equivalent to `S a' <= b`. We need to consider all the values that `b` can take, and so `b` is concretized to ⊥ in *P6*, to `Z` in *P7*, and to `S b'` in *P9*. We focus on *P9*, as *P6* and *P7* proceed similarly to *P2* and *P3*. Running further evaluations on both expressions in *P9* results in step *P10*. One final symbolic execution step on the left-hand side reduces `S (min a' b') === S a'` to the expression in *S11*, `min a' b' === a'`.

Notice the similarity between the states we have derived (`min a' b' === a'` and `a' <= b'`) and the states from the start (`min a b === a` and `a <= b`.) Apart from the names of the symbolic variables, the states are identical. This correspondence allows us to apply coinduction to discharge the states. The original left-hand state aligns with the current left-hand state, and the original right-hand state aligns with the current right-hand state. The variables `a` and `b` take the places of `a'` and `b'`, respectively. We have reached a cycle, and that cycle is evidence of the two sides' equivalence in the situation where `a` and `b` are both successors of other natural numbers. This concludes the proof, since all the proof obligations have been discharged.

Proof Derivation To find this proof automatically, NEBULA switches between applying symbolic execution to reduce expressions and looking for opportunities to apply coinduction. Symbolic execution stops at *termination points*. In particular, every function application is a termination point. We attempt to apply coinduction whenever symbolic execution reaches a termination point.

Of course, states need to be in a suitable form for coinduction to apply. In the proof above, the right-hand side of *P10*, `a' <= b'`, is in the correct form for coinduction with the initial state pair. However, the left-hand side of *P10* needs an additional reduction step for coinduction to apply.

Naturally, there is a question: how did NEBULA know to reduce the left side, but not the right side? The answer is that NEBULA, in fact, continues to apply further symbolic execution to both sides. In Figure 2 we presented only relevant steps in the proof, and we left out the further reductions of the right-hand side for simplicity. NEBULA maintains a history of all states on both sides. When trying to apply coinduction, it holds the current left state steady and searches through *all* corresponding right states (and vice versa) in an effort to form a pair that will allow coinduction to succeed.

Example 2.2. Next, we consider the formula `prop01` from the Zeno evaluation suite [Sonnex et al. 2012]. In Figure 3 we define `prop01_lhs` and `prop01_rhs` whose equivalence we want to check. The `take` function takes a natural number `n` and a list as input and returns the first `n` elements of the list. The `drop` function also takes a natural number `n` and a list as input, but it returns all of the elements of the list except the first `n`. The `++` operator represents list concatenation.

For `prop01` to be valid, the natural number `n` needs to be total. If it is not, NEBULA finds a counterexample, with `n` as ⊥ and `xs` as `Z:[]`. The expression `take ⊥ (Z:[])` simplifies to ⊥, and the expression `⊥ ++ drop ⊥ (Z:[])` also simplifies to ⊥ because of its first argument. At the same time, the right-hand side is `Z:[]`, which is a fully-defined expression.

```
prop01_lhs n xs = take n xs ++ drop n xs
prop01_rhs n xs = xs
                                      take Z _ = []
data [a] = [] | a : [a]               take _ [] = []
                                      take (S x) (y:ys) = y : (take x ys)
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys                         drop Z xs = xs
(x:xs) ++ ys = x : (xs ++ ys)         drop _ [] = []
                                      drop (S x) (_:xs) = drop x xs
```

Fig. 3. Zeno Theorem 1

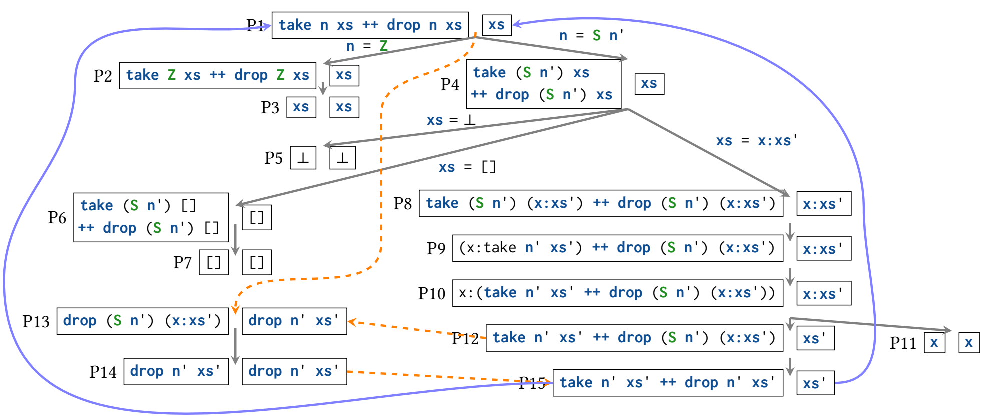

Fig. 4. Overview of how NEBULA proves `prop01`. Gray arrows denote symbolic execution, blue arrows denote coinduction, and orange dashed arrows denote lemma generation or usage.

If the user already knows that certain inputs must be total, then our tool allows the user to mark them as total. NEBULA takes these total inputs' names as command line arguments.

We now discuss the proof steps that NEBULA uses to prove the validity of `prop01` under the assumption that `n` is total. The proof structure is given in Figure 4.

Steps *P1–P9* are similar to those taken in the previous example, so we focus on *P10*. Both sides of *P10* are applications of the list constructor `:`, so they cannot undergo any more non-strict evaluation. We check equivalence of the expressions in *P10* by checking equivalence of both the head and the tail. This results in two new steps: *P11* checks that the list heads are equivalent (and can be discharged trivially by syntactic equality), while *P12* checks that the tails are equivalent. Discharging *P12* requires proving that `take n' xs' ++ drop (S n') (x:xs')` is equivalent to `xs'`.

It might look tempting to apply coinduction between *P12* and *P1*. Unfortunately, this does not work. In the call to `take`, `n'` and `xs'` in *P12* take the place of `n` and `xs` from *P1*, but in the call to `drop`, we have `S n'` and `x:xs'` in *P12* in place of `n` and `xs` in *P1*. No consistent mapping can be formed between the two state pairs, so we cannot apply coinduction to *P12* and *P1*.

To circumvent the problem, we attempt to prove a lemma based on sub-expressions of *P12* and *P1*. Specifically, we automatically derive a potential lemma stating that `drop (S n') (x:xs')` is equivalent to `drop n' xs'`. We form the expression `drop n' xs'` by taking the sub-expression in *P1*

$$
\begin{array}{lllll}
e & ::= & & & \textbf{Expressions} \\
 & | & x & & \text{variable} \\
 & | & s & & \text{symbolic variable} \\
 & | & \lambda x\ .\ e & & \text{lambda} \\
 & | & D & & \text{data constructor} \\
 & | & e\ e & & \text{application} \\
 & | & \mathsf{case}\ e\ \mathsf{of}\ \{\vec{a}\} & & \text{case} \\
 & | & \bot^{L} & & \text{bottom} \\
a & ::= & D\,\vec{x} \rightarrow e & & \textbf{Alternatives}
\end{array}
$$

Fig. 5. The language considered by NEBULA

that should align with `drop (S n') (x:xs')` in *P12* and then applying variable substitutions based on the correspondence that holds for the rest of the expression (i.e. for the applications of `take`). This potential lemma appears as *P13* in the diagram.

Proving the lemma in *P13* is straightforward. Using the lemma, NEBULA now rewrites the expression `take n' xs' ++ drop (S n') (x:xs')` as `take n' xs' ++ drop n' xs'`, as shown in *P15*. Finally, this proof obligation can be discharged by applying coinduction with *P1*.

Example 2.3. Our last example, also from the Zeno suite [Sonnex et al. 2012], illustrates how NEBULA finds counterexamples. Consider Zeno theorem 10, which asserts the equivalence of `m - m` and `Z`. This is true under strict semantics but not under non-strict semantics, even when `m` is total. When run on `m - m` and `Z`, NEBULA finds a counterexample exposing this inequivalence. NEBULA starts by applying symbolic execution to `m - m`. Applying symbolic execution to `Z` is not possible, as it is already fully reduced. Evaluating `m - m` requires concretizing `m`. On the branch where `m = S m'`, NEBULA will reduce `S m' - S m'` to `m' - m'`.

So far, this reduction is similar to the process seen in previous examples, and one might expect to apply coinduction between `m - m` and `m' - m'`. However, coinduction cannot be applied here because the other expression, `Z`, is already fully reduced (the reason for this restriction on the use of coinduction will be explained in Section 4.2.) On the contrary, we have found a *cycle counterexample*. The new expression `m' - m'` is as general as the original expression `m - m`. This means that we can follow the same reduction steps that `m - m` took to reduce to `m' - m'` over again. `m' - m'` can reduce to `m'' - m''`, and the process could repeat forever, resulting in non-termination. On the other hand, `Z` has already terminated. Mapping `m' - m'` to `m - m` requires replacing `m'` with `m`, and, in the state `m' - m'`, we have concretized `m` as `S m'`. Thus, we can conclude that letting `m' = m` in `m = S m'` will lead to non-termination, and we obtain the input counterexample `m = S m`.

Note that the direction of the correspondence between the current and previous state to form a cycle counterexample is the *reverse* of that for a proof by coinduction. For coinduction, we show that the past state pair is at least as general as the current state pair, so that any reduction steps that can be applied to the current state pair can also be applied to the past state pair. This means that, if the past state pair cannot be reduced to inequivalent expressions, neither can the current state pair. In contrast, for a cycle counterexample, we show that the current state is at least as general as the past state, so that the current state can continue reduction in the same way as the past state.

3 SYMBOLIC EXECUTION

Symbolic execution is a program analysis technique that runs code with symbolic variables in place of concrete values. Here we describe symbolic execution for a non-strict functional language, which will both allow us to search for counterexamples to proposed equivalences and act as a guide for proof techniques such as coinduction. While symbolic execution as presented here resembles [Hallahan et al. 2019], the formalization has been adapted to account for non-total values. The structure of states and the reduction rules over states have also been simplified.

Syntax Figure 5 shows the core language λ_S used by NEBULA. NEBULA operates over a non-strict typed functional language, consisting of standard elements such as *variables*, *lambdas*, *algebraic datatypes*, and *case statements*. $e : \tau$ denotes that the expression e has type τ. Symbolic variables s are used in λ_S to denote unknown values.

An algebraic datatype is a finite set of constructors with arguments, $D_1 \tau_1^1 \ldots \tau_1^{n_1}, \ldots, D_k \tau_k^1 \ldots \tau_k^{n_k}$. A *bottom* value, denoted $\bot^L$, is an error. The superscript L is a *label*. When we define equivalence in Section 4, two bottoms will be treated as equivalent if and only if they have the same label.

Notation We define $\equiv$ to check syntactic equality of expressions. $e' \in e$ holds if e' is a sub-expression of e. The expression $e\ [e_2\ /\ e_1]$ denotes e with each occurence of the sub-expression e_1 replaced by e_2. If we have a mapping V from symbolic variables to expressions, we write $e\ [V(s)\ /\ s]$ to denote e with all occurrences of s replaced with the expression $V(s)$ for each s in V.

Symbolic Weak Head Normal Form Non-strict semantics reduces expressions to Weak Head Normal Form (WHNF) [Peyton Jones 1996], i.e. a lambda expression or data constructor application. Correspondingly, symbolic execution reduces expressions to *Symbolic Weak Head Normal Form (SWHNF)*. SWHNF is defined as follows:

$$\mathtt{SWHNF}(e) = \begin{cases} \text{True} & e \equiv s \\ \text{True} & e \equiv D\,\vec{e} \\ \text{True} & e \equiv \lambda x\ .\ e \\ \text{True} & e \equiv \bot^L \\ \text{False} & \text{otherwise} \end{cases}$$

Symbolic variables and bottoms are in SWHNF because they function as stopping points for symbolic execution, just as lambda expressions and data constructor applications do.

States Symbolic execution operates on *states* of the form $(e,\ Y)$. e is the expression being evaluated. The *symbolic store* Y is used to record values assigned to symbolic variables. Symbolic variables map to data constructors that are fully applied to symbolic variables. We refer to the mappings as *concretizations*. We write $s \in Y$ if Y has a mapping for s. We overload $\in$, so that $(s, e) \in Y$ denotes that s is mapped to e in Y. $\mathtt{lookup}(s, Y)$ denotes the data constructor application that Y contains for s. $Y\{s \rightarrow D\,\vec{s}\}$ denotes the symbolic store Y with s mapped to $D\,\vec{s}$.

Reduction We formalize evaluation in terms of small-step reduction rules. We write $S \hookrightarrow S'$ to indicate that S can take a single step to the state S'. We write $S \hookrightarrow^* S'$ to indicate that S can be reduced to the state S' by zero or more applications of $\hookrightarrow$. Because expressions can contain symbolic values, it is sometimes possible to apply more than one reduction rule to a state or to apply the same rule in multiple different ways. Whenever this situation arises in symbolic execution, the state is duplicated, and *each* possible rule is applied to a distinct copy of the state. This enables the execution to explore all possible paths through a program.

Figure 6 shows the reduction rules. The rules for lambda expressions and applications are standard. VAR looks up expressions (such as the definitions of `min` or <= in Example 2.1) in an implicit environment. Note that these expressions may be recursive. A case expression $\mathtt{case}\ e\ \mathtt{of}\ \{\vec{a}\}$ branches depending on the value of e, which we call the *scrutinee*. The CSEV rule for case statements reduces the scrutinee of the case statement to SWHNF, so that CSDC can be used to select the appropriate branch. If the scrutinee of the case statement evaluates to a symbolic variable s, the applicable rule depends on whether the symbolic variable is already in the state's symbolic store Y. If $s \in Y$, the

$$\textsc{Var}\ \frac{}{\begin{array}{c}(x,\ Y) \hookrightarrow \\ (\texttt{lookup}(x),\ Y)\end{array}} \qquad \textsc{App}\ \frac{\begin{array}{c}\neg\mathsf{SWHNF}(f) \\ (f,\ Y) \hookrightarrow (f',\ Y')\end{array}}{\begin{array}{c}(f\ e,\ Y) \hookrightarrow \\ (f'\ e,\ Y')\end{array}} \qquad \textsc{App}\lambda\ \frac{}{\begin{array}{c}((\lambda x\ .\ e')\ e,\ Y) \hookrightarrow \\ (e'\ [e\ /\ x],\ Y)\end{array}}$$

$$\textsc{CsEv}\ \frac{(e,\ Y) \hookrightarrow (e',\ Y')}{\begin{array}{c}(\mathsf{case}\ e\ \mathsf{of}\ \{\vec{a}\},\ Y) \hookrightarrow \\ (\mathsf{case}\ e'\ \mathsf{of}\ \{\vec{a}\},\ Y')\end{array}} \qquad \textsc{CsDC}\ \frac{}{\begin{array}{c}(\mathsf{case}\ D\ \vec{e}\ \mathsf{of}\ \{D\ \vec{x} \to e_a; \ldots\},\ Y) \hookrightarrow \\ (e_a\ [\vec{e}\ /\ \vec{x}],\ Y)\end{array}}$$

$$\textsc{FrDC}\ \frac{s \notin Y \qquad \vec{s}\ \text{fresh}}{\begin{array}{c}(\mathsf{case}\ s\ \mathsf{of}\ \{D\ \vec{x} \to e_a; \ldots\},\ Y) \hookrightarrow \\ (e_a\ [\vec{s}\ /\ \vec{x}],\ Y\{s \to D\ \vec{s}\})\end{array}} \qquad \textsc{LkDC}\ \frac{s \in Y \qquad D\ \vec{s} = \texttt{lookup}(s, Y)}{\begin{array}{c}(\mathsf{case}\ s\ \mathsf{of}\ \{D\ \vec{x} \to e_a; \ldots\},\ Y) \hookrightarrow \\ (e_a\ [\vec{s}\ /\ \vec{x}],\ Y)\end{array}}$$

$$\textsc{BtDC}\ \frac{L\ \texttt{fresh}}{\begin{array}{c}(\mathsf{case}\ s\ \mathsf{of}\ \{D\ \vec{x} \to e_a; \ldots\},\ Y) \\ \hookrightarrow (\bot^L,\ Y\{s \to \bot^L\})\end{array}} \qquad \textsc{BtApp}\ \frac{}{\begin{array}{c}(\bot^L\ e,\ Y) \\ \hookrightarrow (\bot^L,\ Y)\end{array}} \qquad \textsc{BtCs}\ \frac{}{\begin{array}{c}(\mathsf{case}\ \bot^L\ \mathsf{of}\ \{\vec{a}\},\ Y) \\ \hookrightarrow (\bot^L,\ Y)\end{array}}$$

Fig. 6. Reduction Rules

rule LkDC selects the appropriate case statement branch to continue evaluation. If $s \notin Y$, then FrDC splits the state to explore *each* possible branch, and it records the choice made along each branch in Y so that LkDC can be applied the next time each state branches on s.

BtApp and BtCs force any expression which must evaluate $\bot^L$ to reduce to $\bot^L$ itself. BtDC concretizes a symbolic variable to $\bot^L$ with a fresh label L. The inclusion of BtDC requires any proofs relying on our symbolic execution engine to consider the possibility of a partial input for any of a program's arguments. Labels can be used to distinguish between errors from distinct sources.

Our reduction rules, as we present them here, assume that all symbolic values are first-order. Nevertheless, our system is capable of proving properties that involve symbolic functions. We describe our method of handling symbolic functions in Section 6.

Approximation We define an *approximation relation* $\sqsubseteq_V$ on states. Intuitively, $S \sqsubseteq_V S'$ ("S is approximated by S'" or "S' approximates S") if S is a more concrete version of S'—that is, if S replaces all the symbolic variables in S' with other expressions in a consistent way and is the same as S' otherwise.

We formalize $\sqsubseteq_V$ in Figure 7. $S \sqsubseteq_V S'$ holds if there is any inference tree with $S \sqsubseteq_V S'$ as the root. The subscript V is a mapping $V = \{\ldots(s, e), \ldots\}$ from symbolic variables in S' to expressions in S. We define $\texttt{lookup}(s, V)$ to refer to the expression e such that $(s, e) \in V$. We overload $\in$, so that $s \in V$ holds if there is some mapping for s in V. We use $S \sqsubseteq S'$ as shorthand for $\exists V . S \sqsubseteq_V S'$.

It should be noted that checking whether one state approximates another is undecidable in general, as it requires checking if a state's execution (alternatively, a program's execution) will reach a particular point eventually. However, our formalization of $\sqsubseteq$ carefully ensures that symbolic execution explores all paths through a program, and thus can be used to verify properties of programs. We state this formally as Theorem 3.1:

Theorem 3.1 (Symbolic Execution Completeness). *Let S_1 and S_2 be states such that $S_1 \sqsubseteq S_2$. If $S_1 \hookrightarrow S_1'$, then either $S_1' \sqsubseteq S_2$, or there exists S_2' such that $S_2 \hookrightarrow S_2'$, and $S_1' \sqsubseteq S_2'$.*

Most of the rules of $\sqsubseteq$ simply walk over the two states' expressions recursively. The most interesting piece of the definition of $\sqsubseteq_V$ is the handling of symbolic variables on the right-hand

$$\sqsubseteq\text{-}\textsc{Eval}\ \dfrac{\exists e'.(e_1,\ Y_1) \hookrightarrow^* (e',\ Y_1) \land (e',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)}{(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)}$$

$$\sqsubseteq\text{-}\textsc{Sym1}\ \dfrac{\begin{array}{c}\exists e' = \texttt{lookup}(s, V), e''.(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1) \land (e_1,\ Y_1) \sqsubseteq_V (e'',\ Y_2) \\ \exists e = \texttt{lookup}(s, Y_2) \qquad\qquad (e_1,\ Y_1) \sqsubseteq_V (e,\ Y_2)\end{array}}{(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)}$$

$$\sqsubseteq\text{-}\textsc{Sym2}\ \dfrac{s \notin Y_2 \qquad \exists e = \texttt{lookup}(s, V), e'.(e,\ Y_1) \hookrightarrow^* (e',\ Y_1) \land (e_1,\ Y_1) \sqsubseteq_V (e',\ Y_2)}{(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)}$$

$$\sqsubseteq\text{-}\textsc{Var}\ \dfrac{}{(x,\ Y_1) \sqsubseteq_V (x,\ Y_2)} \qquad \sqsubseteq\text{-}\textsc{Lam}\ \dfrac{(e_1[x/x_1],\ Y_1) \sqsubseteq_V (e_2[x/x_2],\ Y_2) \qquad x \text{ fresh}}{(\lambda x_1 \,.\, e_1,\ Y_1) \sqsubseteq_V (\lambda x_2 \,.\, e_2,\ Y_2)}$$

$$\sqsubseteq\text{-}\textsc{Case}\ \dfrac{\begin{array}{c}(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2) \\ \forall (D\ \vec{x_1} \to e_1^a) \in a_1 . \exists (D\ \vec{x_2} \to e_2^a) \in a_2, \vec{x} \text{ fresh}.(e_1^a[\vec{x}/\vec{x_1}],\ Y_1) \sqsubseteq_V (e_2^a[\vec{x}/\vec{x_2}],\ Y_2)\end{array}}{(\mathsf{case}\ e_1\ \mathsf{of}\ \{\vec{a_1}\},\ Y_1) \sqsubseteq_V (\mathsf{case}\ e_2\ \mathsf{of}\ \{\vec{a_2}\},\ Y_2)}$$

$$\sqsubseteq\text{-}\textsc{DC}\ \dfrac{}{(D,\ Y_1) \sqsubseteq_V (D,\ Y_2)} \quad \sqsubseteq\text{-}\textsc{App}\ \dfrac{\begin{array}{c}(e_1,\ Y_1) \sqsubseteq_V (e_1',\ Y_2) \\ (e_2,\ Y_1) \sqsubseteq_V (e_2',\ Y_2)\end{array}}{(e_1\ e_2,\ Y_1) \sqsubseteq_V (e_1'\ e_2',\ Y_2)} \quad \sqsubseteq\text{-}\textsc{Bt}\ \dfrac{}{(\bot^L,\ Y_1) \sqsubseteq_V (\bot^L,\ Y_2)}$$

Fig. 7. Approximation Definition

side of the relation. The rule ⊑-Sym2 allows us to establish that $(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)$ when $s \notin Y_2$, by fetching $e = \texttt{lookup}(s, V)$ and checking if there is some e' such that $(e,\ Y_1) \hookrightarrow^* (e',\ Y_1)$ and $(e_1,\ Y_1) \sqsubseteq_V (e',\ Y_2)$. ⊑-Sym1 is similar to ⊑-Sym2, but it applies to the case where there is some $e = \texttt{lookup}(s, V)$, and thus requires additionally that $e_1 \sqsubseteq e$. The final rule of interest is ⊑-Eval, which states that $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ if there is some e' such that $(e_1,\ Y_1) \hookrightarrow^* (e',\ Y_1)$ and $(e',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$. In other words, an arbitrary number of deterministic reduction rules can be applied to the left-hand expression of $\sqsubseteq_V$.

Allowing arbitrary evaluation at various points is essential to ensure that Theorem 3.1 holds. The following example illustrates this:

Example 3.1. Consider the approximation

$$(\mathsf{case}\ id\ D\ \mathsf{of}\ \{D \to f\ (id\ D)\},\ \{\}) \sqsubseteq_{\{s \to id\ D\}} (\mathsf{case}\ s\ \mathsf{of}\ \{D \to f\ s\},\ \{\})$$

where id is the identity function, $\lambda x \,.\, x$, and f is an arbitrary function. After a single reduction step, the left-hand side of the expression will have inlined the definition of id, reducing to this:

$$(\mathsf{case}\ (\lambda x \,.\, x)\ D\ \mathsf{of}\ \{D \to f\ (id\ D)\},\ \{\}).$$

If $\sqsubseteq$ required that a symbolic variable on the right map *precisely* to the expression on the left, then

$$(\mathsf{case}\ (\lambda x \,.\, x)\ D\ \mathsf{of}\ \{D \to f\ (id\ D)\},\ \{\}) \sqsubseteq_V (\mathsf{case}\ s\ \mathsf{of}\ \{D \to f\ s\},\ \{\})$$

would not hold for any V. ⊑-Sym2 allows leaving $V = \{s \to id\ D\}$, to preserve the approximation.

In Section 6, we will formalize a simpler computable relation $\subseteq$ that *implies* approximation. In our implementation of NEBULA, we use $\subseteq$ rather than $\sqsubseteq$ to satisfy the premises of our proof rules.

$$\textsc{Syn-Eq-Equiv}\frac{e_1 = e_2}{R,\ Y, e_1 \ \equiv\ e_2} \qquad \textsc{DC-Equiv}\frac{\forall_{i=1}^{k} R,\ Y, e_i^1 \ \equiv\ e_i^2}{R,\ Y, D\, e_1^1 \ldots e_k^1 \ \equiv\ D\, e_1^2 \ldots e_k^2}$$

$$\textsc{Lam-Equiv}\frac{\begin{array}{c}s\ \texttt{fresh}\\ R,\ Y, (\lambda x_1 \,.\, e_1)\, s \ \equiv\ (\lambda x_2 \,.\, e_2)\, s\end{array}}{R,\ Y, \lambda x_1 \,.\, e_1 \ \equiv\ \lambda x_2 \,.\, e_2} \qquad \textsc{Bot-Equiv}\frac{}{R,\ Y, \bot^L \ \equiv\ \bot^L}$$

Fig. 8. Syntactic equivalence and equivalence based on splitting SWHNF expressions

4 EQUIVALENCE

Consider two expressions e_1 and e_2 that share a set of free (symbolic) variables $\{s_1 \ldots s_k\}$. We wish to define equivalence $\equiv$ for non-strictly computed values. Intuitively, equivalence for non-strictly computed values means that the two expressions both evaluate to the same value or both fail to terminate. We will formalize this with some mutually recursive definitions. First, we define $\equiv^{WHNF}$, which checks equivalence only on WHNF expressions and labeled bottoms (and treats bottoms with different labels as inequivalent):

$$(e_1 \ \equiv^{WHNF} \ e_2) = \begin{cases} \forall_{i=1}^{k}.e_i^1 \ \equiv\ e_i^2 & e_1 = (D_1\, e_1^1 \ldots e_k^1) \wedge e_2 = (D_1\, e_1^2 \ldots e_k^2) \\ \forall e.e_1' \,[e \,/\, s_1] \ \equiv\ e_2' \,[e \,/\, s_2] & e_1 = \lambda s_1 \,.\, e_1' \wedge e_2 = \lambda s_2 \,.\, e_2' \\ L_1 = L_2 & e_1 = \bot^{L_1} \wedge e_2 = \bot^{L_2} \\ \text{False} & \text{otherwise} \end{cases}$$

Next, we say that a group of concretizations $e_1^a, \ldots e_k^a$ for variables $\{s_1 \ldots s_k\}$ *satisfies* Y if there exists some mapping V such that, for every $1 \le i \le k$, either s_i is unmapped in Y or $(e_i^a,\ Y) \sqsubseteq_V (e_i,\ Y)$, where $e_i = \texttt{lookup}(s_i, Y)$. Now we can define general equivalence. We say that e_1 and e_2 are equivalent with respect to some symbolic store Y and write $e_1 \ \equiv_{Y,P}\ e_2$ if, for all concrete assignments $e_1^a, \ldots e_k^a$ to $\{s_1 \ldots s_k\}$ that satisfy Y, both expressions either (1) evaluate to the same WHNF expression, with corresponding internal values or thunks also equivalent:

$$\exists e_1', e_2'.e_1[e_1^a \,/\, s_1 \ldots e_k^a \,/\, s_k] \hookrightarrow^* e_1' \wedge e_2[e_1^a \,/\, s_1 \ldots e_k^a \,/\, s_k] \hookrightarrow^* e_2' \wedge e_1' \ \equiv^{WHNF} \ e_2'$$

or (2) do not terminate:

$$\begin{aligned}\forall e_1', e_2'.(e_1[e_1^a \,/\, s_1 \ldots e_k^a \,/\, s_k] \hookrightarrow^* e_1' \wedge e_2[e_1^a \,/\, s_1 \ldots e_k^a \,/\, s_k] \hookrightarrow^* e_2')\\ \implies (\neg\mathsf{SWHNF}(e_1') \wedge \neg\mathsf{SWHNF}(e_2'))\end{aligned}$$

We treat bottom values with different labels as distinct because programmers might not want to treat errors with different sources as interchangeable. Recall that, when a symbolic variable is concretized as a bottom value, it receives a fresh label to distinguish it from other bottom values. This also means we do not need to distinguish between a symbolic variable's evaluation terminating with an error or failing to terminate: the labeled bottom can represent either behavior since it is distinct from non-terminating expressions and from other bottom values.

4.1 Equivalence Rules

We define a relation on states $S \ \equiv\ S'$ that is true if and only if corresponding inputs to S and S' produce syntactically equivalent outputs. Here, we formalize proof rules that allow NEBULA to show that $S \ \equiv\ S'$ holds. In Section 6, we will discuss the actual implementation of these rules in NEBULA.

Syntactic and SWHNF Equivalence The rules in Figure 8 allow us to prove the equivalence of two expressions. The rule Syn-Eq-Equiv allows us to discharge two expressions as equivalent if they are syntactically equal. The other three rules concern expressions in SWHNF. Given two

$$\textsc{Red-L}\ \frac{\begin{array}{c}\forall(e_1',\ Y')s.t.(e_1,\ Y) \hookrightarrow (e_1',\ Y').\\ R,\ Y', e_1' \ \equiv\ e_2\end{array}}{R,\ Y, e_1 \ \equiv\ e_2} \qquad \textsc{Red-R}\ \frac{\begin{array}{c}\forall(e_2',\ Y')s.t.(e_2,\ Y) \hookrightarrow (e_2',\ Y').\\ R,\ Y', e_1 \ \equiv\ e_2'\end{array}}{R,\ Y, e_1 \ \equiv\ e_2}$$

Fig. 9. Reduction Rules

$$\textsc{RAdd}\ \frac{R \cup (e_1, e_2, Y),\ Y, e_1 \ \equiv\ e_2}{R,\ Y, e_1 \ \equiv\ e_2} \quad \textsc{U-Coind}\ \frac{\begin{array}{c}(e_1^R, e_2^R, Y^R) \in R \qquad \neg\mathsf{SWHNF}(e_1^R) \qquad \neg\mathsf{SWHNF}(e_2^R)\\ \exists V.(e_1,\ Y) \sqsubseteq_V (e_1^R,\ Y^R) \wedge (e_2,\ Y) \sqsubseteq_V (e_2^R,\ Y^R)\end{array}}{R,\ Y, e_1 \ \equiv\ e_2}$$

$$\textsc{G-Coind}\ \frac{\exists (e_1^R, e_2^R, Y^R) \in R, V\ .\ (e_1,\ Y) \sqsubseteq_V (e_1^R,\ Y^R) \wedge (e_2,\ Y) \sqsubseteq_V (e_2^R,\ Y^R)}{R,\ Y, e_1 \ \equiv\ e_2}$$

Fig. 10. Unguarded and Guarded Coinduction

expressions that are applications of the same data constructor, $e_1 = D\ e_1^1 \ldots e_k^1$ and $e_2 = D\ e_1^2 \ldots e_k^2$, the rule DC-Equiv reduces checking the equivalence of e_1 and e_2 to checking the equivalence of each matching argument pair (e_i^1, e_i^2). Lam-Equiv states that two lambda expressions are equivalent if their applications to a fresh symbolic value are equivalent. Bot-Equiv says two bottoms are equivalent if they share a label. These rules follow easily from the definition of equivalence.

Reduction Rules Figure 9 shows the rules Red-L and Red-R, which apply symbolic execution to the left and right state, respectively, being checked by the relation. The correctness of these rules is justified by Theorem 3.1, which establishes the completeness of symbolic execution.

When used alongside the SWHNF equivalence rules, Red-L and Red-R are sufficient to *check* equivalence up to some input depth, on programs that terminate for all finite inputs. In the next section, we will see how coinduction can be used to extend this result to arbitrarily large inputs and programs which do not necessarily terminate, allowing full *verification* of equivalence.

4.2 Equivalence Verification with Coinduction

The basis of nebula's approach to verification is coinduction. Coinduction is a proof technique that applies to infinite data structures, just as induction applies to finite data structures. Whereas induction might be seen as constructing a complex object from a base case and inductive steps, coinduction works in the opposite direction. Coinduction relies on a proof that an object upholds a property and then deconstructs the object to show that each of its parts satisfies the same property [Gordon 1995; Kozen and Silva 2017]. Coinduction uses a *bisimulation* to prove two states' equivalence. A bisimulation is a relation between states, in which two states are related only if they are still related after being reduced. We formalize our use of coinduction as the rules RAdd, U-Coind, and G-Coind in Figure 10. In our calculus, we build a bisimulation R as a set of state pairs (S_1, S_2). R relates S_1 and S_2 if either (1) evaluating S_1 and S_2 results in a cycle where the two states are approximated (as defined in Section 3) by other states in R or (2) S_1 and S_2 are equivalent when reduced to SWHNF. In the case that both states reach SWHNF expressions with sub-expressions, equivalence of the sub-expressions can be established either by coinduction (relating the sub-expressions with R) or by some other technique such as syntactic equality.

As previously stated, Figure 10 shows the coinduction rules RAdd, U-Coind, and G-Coind that nebula uses to prove state pairs' equivalence. RAdd attempts to build a bisimulation by adding an expression pair (e_1^R, e_2^R) and a corresponding symbolic store Y^R to R. U-Coind allows nebula

$$\textsc{LemmaLeft}\ \frac{\begin{array}{ccc}\{\},\ Y^L, e_1^L \ \equiv\ e_2^L & e_1 = f\ e_1^a\ \dots\ e_k^a & \exists e_1' \in e_1.(e_1',\ Y) \sqsubseteq_V (e_1^L,\ Y^L) \\ e_2^V = e_2^L\ [V(s)\ /\ s] & \neg\texttt{calls}(e_2^V, f) & R,\ Y, e_1\ [e_2^V\ /\ e_1'] \ \equiv\ e_2\end{array}}{R,\ Y, e_1 \ \equiv\ e_2}$$

$$\textsc{LemmaRight}\ \frac{\begin{array}{ccc}\{\},\ Y, e_1 \ \equiv\ e_2 & e_2 = f\ e_1^a\ \dots\ e_k^a & \exists e_2' \in e_2.(e_2',\ Y) \sqsubseteq_V (e_2^L,\ Y^L) \\ e_1^V = e_1^L\ [V(s)\ /\ s] & \neg\texttt{calls}(e_1^V, f) & R,\ Y, e_1 \ \equiv\ e_2\ [e_1^V\ /\ e_2']\end{array}}{R,\ Y, e_1 \ \equiv\ e_2}$$

$$\textsc{LemmaOver}\ \frac{\{\},\ Y^L, e_1^L \ \equiv\ e_2^L \qquad (e_1,\ Y) \sqsubseteq_V (e_1^L,\ Y^L) \qquad (e_2,\ Y) \sqsubseteq_V (e_2^L,\ Y^L)}{R,\ Y, e_1 \ \equiv\ e_2}$$

Fig. 11. Proof Rules for Lemmas

to discharge a pair of expressions (e_1, e_2) and a corresponding symbolic store Y if $\neg\mathsf{SWHNF}(e_1^R)$, $\neg\mathsf{SWHNF}(e_2^R)$, and there is a mapping V such that $(e_1,\ Y) \sqsubseteq_V (e_1^R,\ Y^R)$ and $(e_2,\ Y) \sqsubseteq_V (e_2^R,\ Y^R)$. G-Coind allows nebula to discharge a pair of expressions (e_1, e_2) and a corresponding symbolic store Y if there is a mapping V such that $(e_1,\ Y) \sqsubseteq_V (e_1^R,\ Y^R)$ and $(e_2,\ Y) \sqsubseteq_V (e_2^R,\ Y^R)$.

At a high level, U-Coind and G-Coind are both sound because of Theorem 3.1. If there is a path that could lead to a counterexample between $(e_1,\ Y)$ and $(e_2,\ Y)$, then there must also be a path that leads to a counterexample between $(e_1^R,\ Y^R)$ and $(e_2^R,\ Y^R)$.

To uphold soundness, we enforce *productivity* properties for our proof trees when applications of RAdd, U-Coind, and G-Coind occur. The productivity properties involve the rules from Figures 8 and 9:

Definition 4.1 (U-Productivity). A proof tree is U-productive if both an application of Red-L and an application of Red-R occur between every use of RAdd and every corresponding use of U-Coind.

Definition 4.2 (G-productivity). A proof tree is G-productive if an application of DC-Equiv or Lam-Equiv occurs between every use of RAdd and every corresponding use of G-Coind.

A proof tree must be both U-productive and G-productive in order to be valid. Enforcing U-productivity prevents us from making circular proofs that add states to R and then immediately use the added states to discharge the branch. G-productivity prevents circular proofs in the same way that U-productivity does, but it allows us to use states that are in SWHNF during coinduction. This is important if a state enters SWHNF immediately after an application of DC-Equiv or Lam-Equiv.

Soundness We define the soundness of an equivalence checker as follows:

Definition 4.3 (Soundness). A set of proof rules is sound if a productive proof tree using those rules, and with the conclusion $R,\ \{\}, e_1 \ \equiv\ e_2$, can be constructed only if e_1 and e_2 are equivalent.

We formally state the soundness of the coinduction rules, in combination with the rules from the prior sections, as the following theorem:

Theorem 4.4 (Soundness of Coinduction Rules). *The syntactic equality rule (Syn-Eq-Equiv), the SWHNF equivalence rules (DC-Equiv and Lam-Equiv), the reduction rules (Red-L and Red-R), and the coinduction rules (RAdd, U-Coind, and G-Coind) are sound when used in a productive proof tree.*

4.3 Lemmas

As we mentioned in Example 2.2, direct applications of coinduction are not always possible. Sometimes we need *lemmas*—extra state pairs that we have proven equivalent—in order to guide an expression into a form more amenable to $\sqsubseteq$ and coinduction.

In Figure 11 we introduce three rules, LemmaLeft, LemmaRight, and LemmaOver, that allow us to apply lemmas soundly alongside coinduction.

LemmaLeft and LemmaRight The rule LemmaLeft substitutes one expression for another on the left-hand side of a state pair and uses a lemma to justify the substitution. The first step in applying the rule is proving some lemma $S^L_1 \equiv S^L_2$. The next step is to check if there is some $e'_1 \in e_1$ such that $(e'_1,\ Y_1) \sqsubseteq_V S^L_1$. If there is, we can substitute the mapping V into e^L_2, forming $e^V_2 = e^L_2\ [V(s)\ /\ s]$. Then we simply need to prove the equivalence $R,\ Y, e_1\ [e^V_2\ /\ e'_1] \ \equiv\ e_2$.

For soundness, LemmaLeft requires that two other *lemma productivity properties* hold. First, we require that the expression e_1 be in *function application form*: simply put, e_1 must be a function application $f\ e^a_1 \dots e^a_k$. Second, we require that f, the function being applied, is *not* syntactically included in e^V_2 or syntactically included in any functions invoked by e^V_2, either directly or indirectly.

The two lemma productivity properties prevent us from using lemmas to prove that terminating expressions are equivalent to non-terminating expressions. The need for the two properties arises from the fact that the correctness of coinduction relies in part on the directionality of reduction $\hookrightarrow$. Recall that coinduction relies on detecting cycles in the execution of a program. If we allowed lemma application *without* the lemma productivity properties, lemmas could be used to reverse reduction steps, without completing a cycle, thus allowing for unsound applications of coinduction.

Why do these two requirements prevent this unsoundness? In short, in a finite reduction sequence, a given function f may be called only finitely many times. The equivalence guaranteed by the lemma $(e_1,\ Y) \equiv (e_2,\ Y)$ and the second productivity requirement ensure that, even after lemma substitution, the number of calls to f required for an equivalent (modulo any differences between the reduction of e_1 and e_2) reduction sequence will not be increased by a lemma application. By induction on the number of applications of f, we can then show that, if there exists a reduction path that would demonstrate an inequivalence between the two expressions without the lemma being applied, we will still discover it even after applying the lemma.

LemmaRight resembles LemmaLeft but substitutes on the right-hand side of the state pair.

LemmaOver The rule LemmaOver uses a lemma to discharge an equivalence *immediately* rather than modifying the states for the equivalence. More specifically, LemmaOver derives the conclusion that $(R, Y, e_1 \ \equiv\ e_2)$ from the existence of some e^L_1, e^L_2, and Y^L such that $(\{\}, Y^L, e^L_1 \ \equiv\ e^L_2)$, $(e_1,\ Y) \sqsubseteq_V (e^L_1,\ Y^L)$, and $(e_2,\ Y) \sqsubseteq_V (e^L_2,\ Y^L)$. The justification for the rule is straightforward. Since $(e_1,\ Y) \sqsubseteq_V (e^L_1,\ Y^L)$ and $(e_2,\ Y) \sqsubseteq_V (e^L_2,\ Y^L)$, it must be the case that $(e^L_1,\ Y^L)$ and $(e^L_2,\ Y^L)$ are *generalizations* of $(e_1,\ Y)$ and $(e_2,\ Y)$. That is, $(e^L_1,\ Y^L)$ and $(e^L_2,\ Y^L)$ must over-approximate the behavior of $(e_1,\ Y)$ and $(e_2,\ Y)$. Consequently, if $(e^L_1,\ Y^L)$ and $(e^L_2,\ Y^L)$ are equivalent, so are $(e_1,\ Y)$ and $(e_2,\ Y)$.

5 COUNTEREXAMPLE DETECTION

We now discuss our techniques for detecting *inequivalence* and producing counterexamples. We begin with the simple case, where the inequivalence manifests itself through the expressions terminating with different SWHNF values. Then we explain how we detect *one-sided cycles*: situations where one expression evaluates to a SWHNF value and the other expression fails to terminate.

Inequivalent Values The Inequiv-DC rule, shown in Figure 12, applies when the left-hand and right-hand expressions have been reduced to SWHNF expressions that have distinct outermost data constructors. In this case, the two expressions are inequivalent, and we report their execution path as a counterexample. The rules Inequiv-BotL and Inequiv-BotR state that a labeled bottom is inequivalent to any SWHNF expression except itself.

One-Sided Cycle Detection The one-sided cycle detection rules, CyL and CyR, are shown in Figure 12. The cycle detection rules check if one expression has a non-terminating path while the other

$$\textsc{Inequiv-DC}\,\frac{D_1 \neq D_2}{R,\ Y, D_1\ \vec{e_1}\ \not\equiv\ D_2\ \vec{e_2}}$$

$$\textsc{Inequiv-BotL}\,\frac{\mathsf{SWHNF}(e_2) \qquad \bot^L \neq e_2}{R,\ Y, \bot^L\ \not\equiv\ e_2} \qquad \textsc{Inequiv-BotR}\,\frac{\mathsf{SWHNF}(e_1) \qquad \bot^L \neq e_1}{R,\ Y, e_1\ \not\equiv\ \bot^L}$$

$$\textsc{CyL}\,\frac{\mathsf{SWHNF}(e_2) \qquad (e_1,\ Y) \hookrightarrow^* (e_1',\ Y') \qquad (e_1,\ Y) \sqsubseteq (e_1',\ Y')}{R,\ Y, e_1\ \not\equiv\ e_2}$$

$$\textsc{CyR}\,\frac{\mathsf{SWHNF}(e_1) \qquad (e_2,\ Y) \hookrightarrow^* (e_2',\ Y') \qquad (e_2,\ Y) \sqsubseteq (e_2',\ Y')}{R,\ Y, e_1\ \not\equiv\ e_2}$$

Fig. 12. Counterexample Rules

expression has already terminated. CyL detects the case where the left-hand state $(e_1,\ Y)$ can loop infinitely while $(e_2,\ Y)$ has already reached SWHNF and terminated. To detect non-termination, CyL checks if there is some $(e_1',\ Y')$ such that $(e_1,\ Y) \hookrightarrow^* (e_1',\ Y')$ and $(e_1,\ Y) \sqsubseteq (e_1',\ Y')$. If this is the case, then, by Theorem 3.1, there is an infinite reduction sequence beginning with $(e_1,\ Y)$. Intuitively, the premises $(e_1,\ Y) \hookrightarrow^* (e_1',\ Y')$ and $(e_1,\ Y) \sqsubseteq (e_1',\ Y')$ mean that $(e_1,\ Y)$ can evaluate to a state that is at least as general as itself. Since $(e_1',\ Y')$ is at least as general as $(e_1,\ Y)$, $(e_1',\ Y')$ must have an execution path corresponding to any execution path that $(e_1,\ Y)$ has. $(e_1',\ Y')$ can follow the path corresponding to $(e_1,\ Y) \hookrightarrow^* (e_1',\ Y')$ to reach another state $(e_1'',\ Y'')$ such that $(e_1',\ Y') \sqsubseteq (e_1'',\ Y'')$, and so on to infinity, so we have an infinite reduction sequence. Because this infinite sequence exists, $(e_1,\ Y)$ cannot be equivalent to an expression that has already terminated. We report the one-sided cycle as a counterexample immediately. CyR works in the same way that CyL does, but it handles the case where the right-hand expression is the non-terminating one.

6 AUTOMATED EQUIVALENCE CHECKING

We now detail the automation of NEBULA. NEBULA aims to prove the equivalence of two expressions automatically, or to find a counterexample showing that the expressions are inequivalent, given an initial mapping between the expressions' symbolic variables.

6.1 Approximation Relations

The theoretical approximation relation $\sqsubseteq$ defined in Figure 7 is not computable. To implement the equivalence checking algorithm, we use a simpler approximation relation $\subseteq$, defined in Figure 13. $\sqsubseteq$ is not computable because certain rules check whether one expression can be reduced to another expression. The corresponding rules for $\subseteq$ simply check for syntactic alignment between two states.

As we state in Section 3 and demonstrate with Example 3.1, the use of evaluation in the definition of $\sqsubseteq$ is essential to establish Theorem 3.1, the completeness of symbolic execution. The following theorem, which can be proven by case analysis on the definitions of $\sqsubseteq$ and $\subseteq$, allows us to use $\sqsubseteq$ and to benefit from symbolic execution completeness in theory, while using the computable $\subseteq$ in practice:

Theorem 6.1. *If* $S_1 \subseteq S_2$*, then* $S_1 \sqsubseteq S_2$*.*

Because of this correspondence, we can justify the claim that $S_1 \sqsubseteq S_2$ holds by checking that $S_1 \subseteq S_2$ holds. The rules in Figure 13 compute a mapping V such that $S_1 \subseteq_V S_2$ (alternatively, $S_1 \sqsubseteq_V S_2$.) These rules' premises are judgments of the form $V' \vdash e_1 \lhd_{V, Y_1, Y_2} e_2$, which means that the mapping V can be extended to a new mapping V' such that $(e_1,\ Y_1) \sqsubseteq_{V'} (e_2,\ Y_2)$. Most of the rules walk over the structure of the expressions inductively. The most interesting rules are

$$\lhd\text{-}\textsc{SymV1}\ \frac{s \notin Y_2 \qquad s \notin V}{V \cup \{s \to e\} \vdash e \lhd_{V,Y_1,Y_2} s} \qquad \lhd\text{-}\textsc{SymV2}\ \frac{s \notin Y_2 \qquad e = \texttt{lookup}(s, V)}{V \vdash e \lhd_{V,Y_1,Y_2} s}$$

$$\lhd\text{-}\textsc{SymLkL}\ \frac{\begin{array}{c}\exists e = \texttt{lookup}(s, Y_1)\\ V' \vdash e \lhd_{V,Y_1,Y_2} e_2\end{array}}{V' \vdash s \lhd_{V,Y_1,Y_2} e_2} \qquad \lhd\text{-}\textsc{SymLkR}\ \frac{\begin{array}{c}\exists e = \texttt{lookup}(s, Y_2)\\ V' \vdash e_1 \lhd_{V,Y_1,Y_2} e\end{array}}{V' \vdash e_1 \lhd_{V,Y_1,Y_2} s}$$

$$\lhd\text{-}\textsc{Case}\ \frac{\begin{array}{c}V_1 \vdash e_1 \lhd_{V,Y_1,Y_2} e_2\\ \forall (D\,\vec{x}_1 \to e_1^i) \in \vec{a}_1 . \exists (D\,\vec{x}_2 \to e_2^j) \in \vec{a}_2 . V_{i+1} \vdash e_1^a \lhd_{V_i,Y_1,Y_2} e_2^a[\vec{x}_1/\vec{x}_2]\end{array}}{V_{m+1} \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \{(\vec{a}_1 = a_1^1 \ldots a_1^m)\} \lhd_{V,Y_1,Y_2} \mathsf{case}\ e_2\ \mathsf{of}\ \{(\vec{a}_2 = a_2^1 \ldots a_2^m)\}}$$

$$\lhd\text{-}\textsc{Var}\ \frac{}{V \vdash x \lhd_{V,Y_1,Y_2} x} \qquad \lhd\text{-}\textsc{Lam}\ \frac{V' \vdash e_1 \lhd_{V,Y_1,Y_2} e_2[x_1/x_2]}{V' \vdash \lambda x_1 \,.\, e_1 \lhd_{V,Y_1,Y_2} \lambda x_2 \,.\, e_2} \qquad \lhd\text{-}\textsc{DC}\ \frac{}{V \vdash D \lhd_{V,Y_1,Y_2} D}$$

$$\lhd\text{-}\textsc{App}\ \frac{\begin{array}{c}V' \vdash e_1 \lhd_{V,Y_1,Y_2} e_1'\\ V'' \vdash e_2 \lhd_{V',Y_1,Y_2} e_2'\end{array}}{V'' \vdash e_1\, e_2 \lhd_{V,Y_1,Y_2} e_1'\, e_2'} \qquad \lhd\text{-}\textsc{Bt}\ \frac{}{V \vdash \bot^L \lhd_{V,Y_1,Y_2} \bot^L}$$

$$\subseteq\text{-}\textsc{Link}\ \frac{V \vdash e_1 \lhd_{\{\},Y_1,Y_2} e_2}{(e_1,\ Y_1) \subseteq_V (e_2,\ Y_2)}$$

Fig. 13. Computable Approximation

$\lhd$-SymV1 and $\lhd$-SymV2. $\lhd$-SymV1 applies when e_2 is a symbolic variable not mapped by the current V, and adds e_1 as the mapping for e_2: $V \cup \{s \to e\} \vdash e_1 \lhd_{V,Y_1,Y_2} s$. The rule $\lhd$-SymV2 applies when e_2 is a symbolic variable already in V, and checks that e_1 is syntactically equal to the existing mapping—that is, $V \vdash e_1 \lhd_{V,Y_1,Y_2} s$ if $e_1 = \texttt{lookup}(s, V)$.

6.2 Equivalence Checking Loop

We describe the main verification algorithm here. In this section, we ignore the generation, proving, and usage of lemmas. We will discuss integration of lemmas into the algorithm in Section 6.4.

The algorithm runs symbolic execution on pairs of states, keeping track of all of the branching paths that it encounters. The execution stops periodically so that nebula can attempt to discharge branches by proving the equivalence of the two expressions on a branch. The algorithm terminates when it discharges every branch or finds a contradiction.

Tactics are the basis of nebula's approach to proving equivalence. The main purpose of applying a tactic to a branch is to discharge the branch by proving the equivalence of its two sides, but tactics can also produce potential lemmas or identify counterexamples. We enumerate the proof tactics employed by nebula in Section 6.3.

We refer to the branches that descend from the original proof goal as *obligations*. An obligation is a linear record of the history of two expressions' symbolic execution, divided into *blocks* that represent different stages of simplification of the expressions. A new block is introduced each time an expression reaches SWHNF and the rule DC-Equiv or Lam-Equiv from Figure 8 is applied. Blocks allow us to enforce the productivity properties for both guarded and unguarded coinduction. The verification algorithm deals mainly with obligations rather than dealing with state pairs directly

$$\textsc{LkDC-Sync}\ \frac{s \notin Y \qquad s \in Y_2 \qquad D\,\vec{s} = \texttt{lookup}(s, Y_2)}{(\mathsf{case}\ s\ \mathsf{of}\ \{D\,\vec{x} \to e_a; \ldots\},\ Y) \hookrightarrow_{Y_2} (e_a\ [\vec{s} \,/\, \vec{x}],\ Y\{s \to D\,\vec{s}\})}$$

Fig. 14. Symbolic Store Synchronization

because our primary techniques for proving equivalences require comparisons between different points in expressions' evaluation histories.

The main algorithm, shown as Algorithm 1, maintains a set $\overline{H}$ of obligations. Reduction for the most recent state pair in each obligation continues until it reaches a *termination point*—a point where we consider applying coinduction or other tactics to the state. We will cover the formal definition of a termination point later in this section. Once reduction finishes for each obligation, we generate a set of updated obligations. An individual obligation from the old set can produce one new obligation, multiple new obligations, or no obligations at all. We then apply tactics to the obligations. If any application of a tactic to an obligation finds a contradiction, we terminate the main loop and report that the two original expressions are not equivalent. After attempting to apply every tactic to every obligation, we use the remaining obligations as the starting point for the next loop iteration. If the set of obligations ever becomes empty, we terminate the loop and report that the two original expressions are equivalent.

```
$\overline{H} \leftarrow \{[((e_1, \{\}); (e_2, \{\}))]\}$;
while $\overline{H}$ not empty do
    $\overline{H'} \leftarrow \{\}$;
    for $[\ldots, (S^1_a, \ldots, S^1_b; S^2_c, \ldots, S^2_d)] \in \overline{H}$ do
        Run symbolic execution on $S^1_b$ and $S^2_d$;
        Get $\overline{(S^1_{b+1}, S^2_{d+1})}$ from stopping points on both sides;
        for $(S^1_{b+1}, S^2_{d+1}) \in \overline{(S^1_{b+1}, S^2_{d+1})}$ do
            Make new obligations from $(S^1_{b+1}, S^2_{d+1})$ if possible;
            if obligation creation fails then
                return $(S^1_{b+1}, S^2_{d+1})$ as a counterexample;
            else
                Add the new obligations to $\overline{H'}$;
    for $t \in$ tactics do
        Filter $\overline{H'}$ with $t$;
        if $t$ fails on any obligation then
            return the obligation as a counterexample;
    $\overline{H} \leftarrow \overline{H'}$;
return VERIFIED;
```

Algorithm 1: Verification Algorithm without Lemmas

Obligation Reductions Formally, an obligation H is a list of blocks, where a block B is a pair of lists of states $(S^1_a, \ldots, S^1_b; S^2_i, \ldots, S^2_j)$ such that $\forall a \le c < b. S^1_c \hookrightarrow^*_{Y_j} S^1_{c+1}$ and $\forall i \le k < j. S^2_k \hookrightarrow^*_{Y_b} S^2_{k+1}$. The reductions $\hookrightarrow_{Y_2}$ and $\hookrightarrow^*_{Y_2}$ are the same as $\hookrightarrow$ and $\hookrightarrow^*$, except with a single additional rule: LkDC-Sync, shown in Figure 14. The rule LkDC-Sync ensures that concretizations of a variable stay consistent between the two sides of an obligation. In $\hookrightarrow_{Y_2}$ and $\hookrightarrow^*_{Y_2}$, Y_2 is the symbolic store from the latest state on the opposite side of the obligation. If s has a concretization on the opposite side but not on the side being evaluated, LkDC-Sync copies the concretization from the opposite side's store into the store of the current state.

As a matter of notation, we denote the first state on either side of the first block of an obligation as having an index of 1. If j and k are the last state indices on the two sides of block B_i, then the first states on the corresponding sides of block B_{i+1} have indices of $j+1$ and $k+1$.

Recall that we form a new block whenever we apply DC-Equiv or Lam-Equiv. If S^1_j and S^2_k are the final states in a block B_i, the expressions inside S^1_j and S^2_k must be either data constructor applications or lambda expressions. If the expressions are data constructor applications, then the expressions in the starting states S^1_{j+1} and S^2_{k+1} in B_{i+1} are corresponding arguments from the applications. If S^1_j and S^2_k are lambda expressions, then the expressions in S^1_{j+1} and S^2_{k+1} are applications of those lambda expressions to the same fresh symbolic argument. We divide the state histories in an obligation into blocks in order to uphold soundness for our proof tactics. Since we treat the evaluation sequences on the left and right sides as decoupled, we need a way to ensure that the two states we classify as equivalent actually represent corresponding points in the two sides' evaluation. Example 6.1 demonstrates why blocks are necessary for soundness:

Example 6.1. If we disregarded blocks, we could prove wrongly that `S (S Z) = S Z`. Let P_1 be the starting proof goal, namely `S (S Z) = S Z`. Removing the outer `S` constructors from both sides of P_1 allows us to replace the proof goal with a new goal, `S Z = Z`, which we will call P_2. The left-hand expression in P_2 is `S Z`, which is identical to the right-hand expression in P_1. Since P_2 is a descendant of P_1, it appears as if the left-hand expression from P_1 has been reduced to a point (in P_2) where it is identical to the right-hand expression from P_1. Appealing to the syntactic equality of the two expressions would yield a proof of P_1, but this is not actually valid reasoning because the reduction from P_1 to P_2 does not happen by regular evaluation. Removing the `S` constructors in the reduction from P_1 to P_2 creates a new block, so forbidding the use of syntactic equality between states from different blocks prevents invalid theorems like P_1 from being proven.

Symbolic Execution Termination Symbolic execution stops if the expression being evaluated reaches SWHNF, but some expressions will never reach SWHNF no matter how many evaluation steps they undergo. Because of this, we also stop symbolic execution if an expression is either a fully-applied non-symbolic function or a case statement with a scrutinee that is a fully-applied non-symbolic function. This guarantees termination because the only feature of λ_S that can prevent symbolic execution from reaching SWHNF is recursion. To enforce the productivity properties described in Section 4.2, and to ensure that we use coinduction soundly, we require that symbolic execution have taken at least one step on each side before terminating.

Verification Process Initially, $\overline{H}$ contains only one obligation: $[((e_1,\{\});(e_2,\{\}))]$, where (e_1,e_2) is the starting expression pair. During each iteration of the main loop, for each unresolved obligation $[\ldots,(\ldots,S^1_j;\ldots,S^2_k)]$, we apply reduction to S^1_j (assuming S^1_j is not in SWHNF already) to obtain a new set of states $\overline{S^1_{j+1}}$ such that $\forall S^1_{j+1} \in \overline{S^1_{j+1}}.S^1_j \hookrightarrow^*_{Y^2_k} S^1_{j+1}$. Then, for each $S^1_{j+1} = (e^1_{j+1},\ Y^1_{j+1}) \in \overline{S^1_{j+1}}$, S^2_{k+1} is reduced using $\hookrightarrow^*_{Y^1_{j+1}}$ to obtain a set of states $\overline{S^2_{k+1}}$, which gives us new obligations

$$\{[\ldots,(\ldots,S^1_j,S^1_{j+1};\ldots,S^2_k,S^2_{k+1})] | S^2_{k+1} \in \overline{S^2_{k+1}}\}.$$

If either of the most recent states *is* already in SWHNF, we simply reduce the other state to obtain n new states and append each new state to the appropriate side of the newest block in the obligation, producing n new obligations to take the place of the old one.

6.3 Tactics

After performing symbolic execution, we apply tactics to the obligations in an effort to discharge them or to produce counterexamples. Our proof rules and counterexample rules, as presented in Sections 4 and 5, expect two expressions that share a symbolic store. However, our implementation maintains separate symbolic stores for the left-hand and right-hand expressions in an obligation. We will begin by explaining *synchronization*, our process for joining the two sides' symbolic stores together when applying tactics, and briefly explaining our motivation and justification for this representation. Then we will enumerate the tactics that NEBULA uses in the main verification algorithm.

6.3.1 Synchronization. When we apply tactics, we *synchronize* the left-hand and right-hand states to be used for the tactic with each other.

Method If (e_1, Y_1) and (e_2, Y_2) are two states, then (e_1, Y) and (e_2, Y) are the *synchronized* versions of the states, where $Y = Y_1 \cup Y_2$. There is no risk of concretizations conflicting with each other when we take the union since we only ever synchronize pairs of states from the same obligation. If a symbolic variable s has already been concretized on one side of an obligation, the reduction rule LKDC-SYNC ensures that s cannot receive a conflicting concretization on the opposite side.

Justification Synchronizing the two sides of an obligation just before applying a tactic rather than synchronizing immediately at every opportunity allows us to decouple the evaluation sequences of an obligation's two sides from each other. Allowing staggered present-state and past-state combinations for tactics enables us to identify more opportunities to apply the tactics than we would find otherwise. The latest left-hand and right-hand expressions may not retain any meaningful connection over the course of multiple applications of symbolic execution. If the left-hand side and right-hand side both reach cycles that are usable for coinduction, the cycles may not start or end at the same time, and the two sides will not necessarily hit the same number of stopping points for symbolic execution between the start and end of their cycles.

6.3.2 Tactics. NEBULA uses tactics including syntactic equality and cycle counterexample detection, as outlined in Sections 4 and 5. For the most part, the implementations of these tactics are straightforward from the rules in those sections. However, the implementations of guarded and unguarded coinduction rely heavily on the structure of the obligations and blocks.

Coinduction Coinduction, as described in Section 4.2, allows us to discharge obligations directly. Consider two blocks within an obligation, which may or may not be distinct:

$$[\dots, (S^1_a, \dots, S^1_b; S^2_j, \dots, S^2_k), \dots, (S^1_c, \dots, S^1_d; S^2_m, \dots, S^2_n), \dots]$$

Let B be the first block, and let B' be the second block. Coinduction can be *unguarded* or *guarded.* For unguarded coinduction, B and B' are allowed to be the same block, but all four of the expressions in the present states and past states must not be in SWHNF. For guarded coinduction, the expressions from the present and past states can be in SWHNF, but B and B' must be distinct blocks.

Recall the rule RADD from Figure 10 for adding state pairs to a relation set R. We want to be able to apply RADD to any $1 \leq p_1 < d$ and $1 \leq p_2 < n$, to add $S^1_{p_1}, S^2_{p_2}$ to R. Then we could choose any $p_1 < q_1 \leq d$ or $p_2 < q_2 \leq n$ and attempt to use U-COIND (from Figure 10) to discharge either the state pair $(S^1_d, S^2_{q_2})$ or the state pair $(S^1_{q_1}, S^2_n)$. We synchronize the two present states with each other and the two past states with each other, so that (as the rules in Section 4.2 require) the present states share a symbolic store and the past states share a symbolic store. Note that we do not need to consider applying coinduction to $S^1_{q_1}$ and $S^2_{q_2}$ where both $q_1 \neq d$ and $q_2 \neq n$, because we have considered that possibility already in some past loop iteration. For guarded coinduction, the past

$$\textsc{LemCo}\;\frac{Y' = Y_1 \cup Y_2}{\begin{array}{c}(e_1,\ Y') \ \equiv\ (e_2\,[V(s)\,/\,s],\ Y') \\ \nvdash e_1 \lhd_{V,\,Y_1,\,Y_2} e_2\end{array}} \qquad \textsc{LemGen}\;\frac{\begin{array}{ccc} e_1' \in \text{scrutinees}(e_1) & & e_2' \in \text{scrutinees}(e_2) \\ e_1' = e_2' & s\ \texttt{fresh} & Y' = Y_1 \cup Y_2 \end{array}}{\begin{array}{c}(e_1\,[s\,/\,e_1'],\ Y') \ \equiv\ (e_2\,[s\,/\,e_2'],\ Y') \\ \nvdash e_1 \lhd_{V,\,Y_1,\,Y_2} e_2\end{array}}$$

Fig. 15. Rules for Lemma Introduction

states that we add to R need to have indices $1 \le p_1 \le b$ and $1 \le p_2 \le k$, and we use the rule G-Coind (also from Figure 10) instead. Everything else remains the same as it is for unguarded coinduction.

6.4 Lemmas

Lemmas allow us to modify expressions before applying $\subseteq$ and coinduction to them. Section 4.3 covers the rules and conditions that allow us to apply lemmas soundly. Here, we discuss both the practical implementation of the rules and the heuristics that we use to select potential lemmas.

Coinduction Lemmas We use lemmas to rewrite states into forms that are more amenable for $\subseteq$ and coinduction. Consequently, we generate potential lemmas in situations where $\subseteq$ fails to hold. If we have two states such that $(e_1,\ Y_1) \nsubseteq (e_2,\ Y_2)$, we may be able to generate a lemma that, once proven, allows us to rewrite one of the two states so that the approximation holds.

LemCo in Figure 15 shows how we produce possible lemmas from failed approximation attempts. Specifically, LemCo generates possible lemmas in situations where $\lhd_{V,\,Y_1,\,Y_2}$ fails to hold between two expressions, $\nvdash e_1 \lhd_{V,\,Y_1,\,Y_2} e_2$. We use these expressions to create the possible lemma

$$(e_1,\ Y') \ \equiv\ (e_2\,[V(x)\,/\,x],\ Y').$$

If we prove the lemma, we may be able to rewrite the first function application with it to create a situation where $\lhd_{V,\,Y_1,\,Y_2}$ holds. Note that, if we let V_I denote the identity mapping on variables, $(e_1,\ Y') \subseteq_{V_I} (e_1,\ Y')$. Consequently, once we prove the lemma, the rule LemmaLeft from Figure 11 can replace e_1 with $e_2\,[V(x)\,/\,x]$. We can see that $e_2\,[V(x)\,/\,x] \lhd_{V,\,Y_1,\,Y_2} e_2$, and so it is possible that $\lhd_{V,\,Y_1,\,Y_2}$ will hold for the entirety of the initial expression after the rewriting.

Recall the two lemma productivity properties from Section 4.3 that are sufficient for enforcing sound lemma usage. The first property requires that the expression receiving a substitution based on the lemma is an application of some function f. The second property requires that the function f not appear syntactically in the expression $e_2\,[V(x)\,/\,x]$ being added by the substitution, or in any functions directly or indirectly callable by $e_2\,[V(x)\,/\,x]$. Both requirements can be confirmed before applying a lemma with a simple syntactic check.

Generalization Lemmas The generalization tactic generates potential lemmas that, if proven, can be used to discharge a pair of states $S_1 = (e_1, Y_1)$ and $S_2 = (e_2, Y_2)$ from opposite sides of the same block. To generate these potential lemmas, we define a function to accumulate a non-exhaustive set of the scrutinees of (possibly nested) case statements on either side:

$$\text{scrutinees}(e) = \begin{cases} \{e'\} \cup \text{scrutinees}(e') & e = \mathsf{case}\ e'\ \mathsf{of}\ \{\vec{a}\} \\ \{\} & \text{otherwise} \end{cases}$$

If an expression in scrutinees(e_1) is syntactically equal to an expression in scrutinees(e_2), then we create a potential lemma where the matching scrutinees in e_1 and e_2 are replaced with the same fresh symbolic variable. The rule LemGen in Figure 15 formalizes this. If we prove the lemma, we can use it to discharge the original obligation by applying the LemmaOver rule from Figure 11.

$$\textsc{HgLookup}\ \frac{s' = \texttt{lookup}(s\ e, Y)}{(s\ e,\ Y) \hookrightarrow (s',\ Y)} \qquad \textsc{HgFresh}\ \frac{s\ e \notin Y \qquad s' \text{ fresh}}{(s\ e,\ Y) \hookrightarrow (s',\ Y\{s\ e \to s'\})}$$

Fig. 16. Evaluation for Symbolic Functions

Lemma Implementation Augmenting Algorithm 1 to support lemmas requires a few changes. Every potential lemma receives a fresh name L to differentiate it from other potential lemmas. We add lemma obligations to $\overline{H}$, but we tag every obligation for a potential lemma with the potential lemma's name. We know that we have finished proving a lemma L when every obligation in $\overline{H}$ with L as its tag has been discharged.

We also tag each potential lemma with a *generating state pair* (S_m^i, S_n^i), which is the pair of states that caused us to generate the potential lemma when $\subseteq$ failed to hold. If we succeed in proving the lemma, we retry the coinduction tactic—with the new lemma in hand—on all obligations that include the states S_m^i and S_n^i, with all appropriate state pairs from the other side. We discharge all obligations for which coinduction succeeds with the new lemma.

Before we add any new potential lemma to the list of potential lemmas to prove, we perform a few checks to avoid redundant work. If the new potential lemma is implied by a lemma that has already been proven, is equivalent to a potential lemma that has been proposed but not proven yet, or implies a previously-proposed potential lemma that has been disproven, we discard the potential lemma instead of attempting to prove it. Here, we mean that one potential lemma L implies another potential lemma L' if the generating state pair of L approximates the generating state pair of L' according to $\subseteq$. L and L' count as equivalent if the approximation works in both directions.

Lemmas for Syntactic Equality In addition to allowing lemma usage with coinduction, we also generate potential lemmas from failed attempts at proving syntactic equality, and we apply lemmas when checking for syntactic equality. The changes to syntactic equality match the changes to coinduction closely: potential lemmas are generated from the sub-expressions that cause a syntactic equality check to fail, and, if the lemma is proven, we attempt syntactic equality again on the generating state pair.

6.5 Symbolic Functions

Our implementation supports symbolic function variables, although our earlier formalism does not. The reduction rules for symbolic function applications appear in Figure 16. As symbolic execution proceeds, we record symbolic function applications that we have encountered in the symbolic store, just as we record concretizations of ordinary symbolic variables. If a symbolic function application we are evaluating is syntactically identical to one encountered previously, we apply HgLookup to introduce the same variable that we used before. Otherwise, we apply HgFresh to introduce a new symbolic variable. For simplicity, we check only for syntactic equality between symbolic function applications rather than performing a more thorough equivalence check.

Our verification process remains sound when we introduce symbolic functions, as the symbolic variable that replaces a symbolic function application can assume any value of its type, including $\perp^L$. This means that our handling of symbolic functions can only make proof goals more general.

Although verification remains sound when we support symbolic functions, symbolic functions do introduce the possibility of spurious counterexamples. Expressions can be equivalent even if they are not syntactically identical, so NEBULA may assign two equivalent applications of a symbolic function to two distinct symbolic variables. If the two variables receive different concretizations, the choice of concretizations will represent an impossible situation. NEBULA cannot detect the inconsistency, and it may derive a spurious counterexample from the branch. Nevertheless, spurious counterexamples are rare in practice. In our evaluation, NEBULA never rejected any theorem, valid or invalid, because of a spurious counterexample.

6.6 Total Variables

In our implementation, we allow users to mark specific symbolic variables as total. Total symbolic variables and their descendants cannot be concretized as bottoms. To support total symbolic variables soundly, an additional condition needs to hold for approximations between states. If the approximation mapping V maps the symbolic variable s to an expression e, and s has been marked as a total variable, then e needs to be total as well for the approximation to be valid. Checking totality for expressions in general is undecidable, so the only expressions that we count as total for approximations are data constructors, symbolic variables that have been marked as total, and applications of expressions that are total by the same definition.

Totality works differently for symbolic functions than it does for symbolic variables of algebraic datatypes. We never concretize symbolic functions, so, for our purposes, a total function is one that always maps total inputs to total outputs. During symbolic execution, if we encounter an application of a total symbolic function to arguments that are all total according to our definition from before, we mark the fresh variable that we use as a substitute for the application as total.

7 EVALUATION

We implemented our techniques for equivalence checking with coinduction and symbolic execution in a practical tool, NEBULA. NEBULA is written in Haskell, and it checks equivalence of Haskell expressions automatically. NEBULA is open source. It is available as part of the G2 symbolic execution engine at https://github.com/BillHallahan/G2 or as a virtual machine image at [Kolesar et al. 2022].

In our evaluation of NEBULA, we seek to answer two main questions. (1) When given theorems that hold in a non-strict context, does NEBULA succeed in proving their correctness? (2) When given theorems that hold only in a strict context, does NEBULA succeed in both (a) finding counterexamples in general and (b) finding non-terminating counterexamples for theorems that have them?

We base our evaluation on the 85 theorems from the IsaPlanner suite [Johansson et al. 2010], as they are formulated in the Zeno codebase [Sonnex et al. 2012]. For our main evaluation, we simply run NEBULA on the original formulations of the theorems. Many of the theorems do not hold in a non-strict setting, so we use the true ones for question (1) and the false ones for question (2). As a further assessment of question (1), we also run NEBULA on modified versions of the invalid theorems that hold even when evaluation is non-strict. We group the invalid theorems into two categories. Some of the theorems do not handle errors properly, and requiring some of their arguments to be total makes the theorems true. For other theorems, the possibility of one side diverging while the other terminates is a problem. In these cases, we force one or more of the theorem's arguments to be finite to make the theorem true. If a theorem needs both totality requirements and finiteness requirements to be true in a non-strict setting, we include it in the second category.

Test Suite We give NEBULA its inputs in the form of *rewrite rules*. Rewrite rules are constructs that allow a programmer to express domain-specific optimizations to the GHC Haskell compiler [Peyton Jones et al. 2001]. A rewrite rule consists of a number of universally quantified variables, a pattern for expressions to be replaced, and a pattern for replacement expressions. The two expressions are defined in terms of the universally quantified variables. GHC does type-check rewrite rules, but it does not check that the rules preserve a program's behavior otherwise. We designed NEBULA to take its inputs in the form of rewrite rules to allow for easy rewrite rule verification.

The process for converting theorems into rewrite rules is simple. In the Zeno code, every theorem is a function with a return type of `Bool`. If the outermost layer of a theorem's function body is an equality check between two sub-expressions, then we represent the theorem as a rewrite rule that asserts the equality of the two sub-expressions. Otherwise, we represent the theorem as a rewrite rule that asserts that the theorem's whole expression is equal to `True`. In either case, the universally quantified variables for the rewrite rule are the arguments of the original theorem's function.

Requirements for the Theorems Every theorem in our suite is true under the assumption that all arguments are total and finite. However, most of the theorems no longer hold in their original formulations in a non-strict context. We run nebula on every unmodified theorem to see whether it can verify the ones that remain true and find counterexamples for the ones that become false. To assess nebula's verification abilities further, we also run it on modified versions of the invalid theorems. The modified theorems include extra requirements to make them true in a non-strict context. Some of the modified versions of the theorems require certain variables to be total. Others remove infinite concretizations of specific variables from consideration by forcing the evaluation of one or both sides not to terminate when given an infinite input.

We can require the arguments of a rewrite rule to be total, as outlined in Section 6.6, by designating them as total in the settings of nebula. To force finiteness for an argument, we use type-specific *walk* functions. A walk function for an algebraic datatype τ_w takes two arguments, one of type τ_w and one polymorphic argument of type τ_p. The walk function traverses over some portion of the τ_w argument. The traversal ensures that the function application will raise an error if that portion of the argument is non-total or will fail to terminate if that portion of the argument is infinite. Once the traversal finishes, the walk function returns its τ_p argument.

We add walk functions manually to the theorems that need them. When a variable needs to be finite, we wrap the main expression on one or both sides of a rewrite rule with an application of the corresponding walk function. For example, consider the rewrite rule `prop10`:

```
forall m . m - m = Z
```

Recall from Section 2, Example 2.3, that this rule is false if `m` is infinite, i.e. `m = S m`. Now consider an altered version of `prop10` that includes a walk function on the right-hand side:

```
walkNat Z a = a
walkNat (S x) a = walkNat x a
```

```
forall m . m - m = walkNat m Z
```

The left-hand side still diverges if `m` is infinite, but now the right-hand side diverges as well. Further, there is no need to make `m` total now: both `m - m` and `walkNat m Z` force `m` to be evaluated fully, so if `m` is non-total, both expressions will terminate with the same bottom value.

We utilize three different walk functions in our evaluation. The function `walkNat` applies to natural numbers. The function `walkList` forces the spine of a list to be total and finite but does not impose any restrictions on the contents of the list. The function `walkNatList` forces the spine of a natural number list to be total and finite and also applies `walkNat` to every entry within the list. For the sake of simplicity, we do not consider any finer distinctions for finiteness, even though finer distinctions are possible. In cases where the minimum conditions necessary for a theorem to hold are not expressible in our system, we over-approximate the conditions.

7.1 Results

We give each theorem a time limit of 3 minutes. We ran nebula on a 2.4 GHz Intel Core i9 laptop. Table 1 summarizes the results of our evaluation.

We report a positive answer for question (1): nebula can prove theorems that hold in a non-strict context. Of the 85 unmodified theorems, 24 are true in a non-strict context. nebula proves the correctness of 22 of the 24 correct theorems (92%) and hits the time limit for the other two.

Table 1. Evaluation results. # Thms indicates the number of theorems in a category. # V indicates the number of theorems in the category that were verified. # C indicates the number of theorems that NEBULA marked as untrue by finding counterexamples. # TO indicates the number of timeouts in a category. Avg. V Time is the average time that NEBULA takes to verify the theorems that it proved in a category. Avg. C Time is the average time that NEBULA takes to find a counterexample for the theorems in a category that it rejected.

Category	# Thms	# V	# C	# TO	Avg. V Time (s)	Avg. C Time (s)
Unmodified Theorems	85	22	61	2	11.3	15.1
Modified (No Finite Variables)	18	11	0	7	16.7	N/A
Modified (Finite Variables)	56	12	0	44	6.0	N/A
Cycle Counterexamples	44	0	32	12	N/A	5.5

As an additional assessment of question (1), we also run NEBULA on the theorems modified with totality requirements and finiteness requirements. There are 17 theorems that can be made true with totality requirements and no finiteness requirements. For one of the theorems, namely theorem 23, there are two different possible minimal totality requirements. We can view the two different modified versions of theorem 23 as distinct theorems, bringing the count to 18 for this category. With the minimum totality requirements in place, NEBULA proves 11 of the theorems (61%) and hits the time limit on the remaining 7. There are also 44 theorems that are only true when certain variables are required to be finite. 12 of the 44 theorems have two distinct combinations of minimal totality and finiteness requirements, so we effectively have 56 theorems in this category. NEBULA verifies 12 of the theorems (21%) and hits the time limit on the rest.

We also report a positive answer for both parts of research question (2). For part (a), we can see that NEBULA succeeds at finding counterexamples in general because it produces a genuine counterexample for every single one of the 61 unmodified untrue theorems.

For part (b) of question (2), we have NEBULA attempt to find cycle counterexamples for the 44 unmodified theorems that need finite variables to be true. The suite of unmodified theorems does not suffice for testing this: all of the theorems with non-terminating counterexamples also have terminating counterexamples that involve bottom values. To test NEBULA's ability to detect cycle counterexamples, we required totality for all of the theorems' arguments but did not impose any finiteness requirements. Requiring all of the arguments to be total makes non-cyclic counterexamples impossible. Under these conditions, NEBULA finds genuine cycle counterexamples for 32 of the 44 theorems (73%) and hits the time limit for the other 12.

7.2 Discussion of Results

Finite-Variable Benchmarks NEBULA performs well on the unmodified benchmarks and the totality-requiring benchmarks, but it performs relatively poorly on the finiteness-requiring benchmarks. We do not consider this a major cause for concern. Walk functions are abnormal constructs that do not resemble the code that a programmer would typically write in a non-strict language, and we include them specifically to counteract the non-strict behavior of Haskell.

NEBULA's relatively low success rate on the finiteness-requiring benchmarks stems primarily from its reliance on coinduction as its primary proof tactic. In general, coinduction is not the best fit for verifying properties involving functions that reach SWHNF only on finite inputs. An induction-based proof technique would likely be more appropriate in such a situation. This is the reason why many of the modified benchmarks with finite variables fail: the walk functions used in the modified versions of the theorems terminate only on finite inputs. In particular, NEBULA fails to verify any modified theorem where a list of natural numbers needs to have only finite entries. It

```
height :: Tree a -> Nat
height Leaf = Z
height (Node l x r) = S (max (height l) (height r))
```

Fig. 17. The `height` Function

also fails to verify any modified theorem that includes walk functions for two or more variables. Several of the failing theorems among the unmodified theorems and the modified theorems with only total variables face similar issues. For instance, NEBULA does not verify any valid theorem involving the `rev` and `sort` functions for lists: both functions can traverse the whole spine of their input list before reaching SWHNF.

Inadequate Proof Tactics Walk functions are a major obstacle for NEBULA, but some recursive functions that do reach SWHNF on infinite inputs also present difficulties. For example, the `height` function on binary trees, shown in Figure 17, is not well-suited for NEBULA's proof tactics. Because `height` interleaves applications of `max` with recursive applications of itself, symbolic execution adds an extra `max` application to the expression with every layer of recursion, and this prevents any use of the coinduction tactic. The development of techniques for reasoning about functions like `height` coinductively is an interesting opportunity for future work.

Impact of the Time Limit We believe that the 3-minute time limit for the evaluation does not inhibit NEBULA's performance in any significant way. Usually, when NEBULA can prove an equivalence, it finds the cyclic pattern that it needs for coinduction rather quickly. NEBULA's average times for proving equivalences and finding counterexamples in our evaluation are all under 20 seconds. When NEBULA reaches the time limit for a theorem, what typically happens is that the evaluation of one or both expressions proceeds down an infinite path with no obvious cyclic pattern. As evaluation continues, the proof obligation for that path will keep branching into more obligations that NEBULA has no way of discharging. This state explosion prevents NEBULA from making any real progress toward verifying the equivalence. Because NEBULA behaves in this way in situations where it reaches the time limit, giving NEBULA additional time to run is unlikely to improve its verification coverage in most cases.

8 RELATED WORK

Coinduction NEBULA relies on coinduction, a well-established proof technique [Gibbons and Hutton 2005; Gordon 1995; Kozen and Silva 2017; Rutten 2000; Sangiorgi 2009]. Our primary contribution is the development of a calculus to combine coinduction with symbolic execution, along with the use of that calculus to automate coinductive reasoning for a functional language.

Other researchers have examined the possibility of using coinduction to verify programs' equivalence previously [Koutavas and Wand 2006; Sangiorgi et al. 2007]. Unlike our approach for NEBULA, the formalizations in [Koutavas and Wand 2006] and [Sangiorgi et al. 2007] do not take infinite or non-total inputs into consideration. More importantly, the two papers only provide theoretical frameworks for for proving programs' equivalence by coinduction, not an automated algorithm for generating proofs like the one that we introduce.

Interactive Tools Interactive tools allow a user to prove properties of programs manually or semi-automatically. An interactive setup has the advantage that it might allow the prover to verify larger or more complex properties, but proving each property requires more manual effort.

CIRC [Lucanu and Roşu 2007; Roşu and Lucanu 2009] generates coinductive proofs for values and properties specified in Maude, a logic language. In contrast, NEBULA targets the functional language Haskell. For CIRC's purposes, expressions do not have complete definitions that specify an unambiguous evaluation order for all possible inputs. Instead, CIRC relies on axioms that allow it to make certain substitutions for expressions. While CIRC supports some simple automation, it requires much more manual effort to prove properties than NEBULA requires. For example, CIRC cannot apply case analysis automatically to decompose a property into several subproperties, whereas NEBULA applies case analysis automatically every time it concretizes a symbolic variable.

HERMIT [Farmer et al. 2015] is an interactive verification tool for Haskell programs that accounts for the possibility of bottom expressions. The design of HERMIT is quite different from the design of NEBULA: like CIRC, HERMIT relies on guidance from users in order to find proofs. Users can guide HERMIT to a proof through the tool's interactive REPL.

[Mastorou et al. 2022] describes a method for using the LiquidHaskell verifier to prove coinductive properties. The outlined techniques rely on a *guardedness property* which states that values are produced, and thus, in contrast to our approach with NEBULA, they cannot be used to prove equivalence of non-terminating expressions. The approach also relies on user-written proofs to guide the verifier.

Hs-to-coq [Breitner et al. 2018] automates translation of Haskell code into Coq code, allowing users to verify properties of their Haskell code within Coq. While [Breitner et al. 2018] discusses only inductive proofs, hs-to-coq has been extended to support verification of coinductive properties [Breitner 2018]. However, this verification is not automated: it requires manually-written Coq proofs.

[Leino and Moskal 2014] describes the integration of features supporting coinduction into the modular verifier Dafny. Dafny requires user-provided annotations to specify function and loop behavior, unlike NEBULA, which aims to prove equivalences automatically.

Functional Automated Inductive Proofs Zeno [Sonnex et al. 2012], HipSpec [Claessen et al. 2013], Cyclist [Brotherston et al. 2012], and IsaPlanner [Johansson et al. 2010] are automated theorem provers targeting properties of functional programs. These tools assume strict semantics and, correspondingly, total and finite data structures. Zeno and HipSpec accept Haskell programs as input, but both fail to reason about Haskell in a completely accurate way because they ignore infinite and non-total inputs, unlike NEBULA. Our evaluation highlights the difference. It uses the same benchmarks as Zeno, HipSpec, and IsaPlanner, but only 28% of these theorems are true under non-strict semantics, whereas all of them are true under strict evaluation.

Imperative Symbolic Execution RelSym [Farina et al. 2019] is a symbolic execution engine for proving relational properties of imperative programs. RelSym depends on user-provided invariants in order to reason about loops. Differential symbolic execution [Person et al. 2008] is a technique for detecting behavioral differences that arise from changes to a program. It exploits optimizations based on the assumption that the old and new versions of the program are mostly similar.

(Non)Termination Checking Looper [Burnim et al. 2009], TNT [Gupta et al. 2008], Jolt [Carbin et al. 2011], and Bolt [Kling et al. 2012] detect non-termination of imperative programs. Like NEBULA, these tools rely on finding program states that are, in some sense, repetitions of earlier states. [Le et al. 2020] and [Cook et al. 2014] detect both program termination and non-termination. Both focus on non-linear integer programs, as opposed to the data-structure-heavy programs that NEBULA targets. [Nguyễn et al. 2019] uses symbolic execution and the size-change principle [Lee et al. 2001] to prove termination of functional programs but, unlike NEBULA, does not prove non-termination.

Symbolic Functions [Nguyễn and Van Horn 2015] handles symbolic functions during symbolic execution by using templates to concretize function definitions gradually. It is possible that techniques from [Nguyễn and Van Horn 2015] could complement NEBULA by allowing us to guarantee the correctness of apparent counterexamples. However, our current approach of over-approximation allows us to consider fewer states when we aim to confirm an equivalence.

9 CONCLUSION

We have presented NEBULA, the first fully automated expression equivalence checker designed with non-strictness in mind. We used NEBULA both to verify correct theorems and to find counterexamples for incorrect theorems that hold in a strict setting. We have evaluated our tool in practical settings with promising results.

We view the verification of rewrite rules in production Haskell code as a potential application for NEBULA. Rewrite rules see significant use on Hackage, the main repository of open-source libraries for the Haskell community. In our preliminary survey, we have found that there are over 5000 rewrite rules across more than 300 libraries on Hackage. Consequently, our tool has the potential to assist Haskell programmers with the verification and debugging of rewrite rules. We plan to explore this possibility further in future work.

10 DATA AVAILABILITY STATEMENT

The artifact for NEBULA is available at [Kolesar et al. 2022]. The artifact contains all of the code necessary to reproduce the results presented in Section 7, along with instructions for running the evaluation suite.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback on earlier versions of this paper. We thank Dorel Lucanu for answering our questions about CIRC. This work was supported by the National Science Foundation under Grant Numbers CCF-2131476 and CNS-1565208.

REFERENCES

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices* 39, 1 (2004), 14–25. https://doi.org/10.1145/982962.964003

Joachim Breitner. 2018. *hs-to-coq supports coinduction.* https://mobile.twitter.com/nomeata/status/977257104120664064

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, set, verify! applying hs-to-coq to real-world Haskell code (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–16. https://doi.org/10.1145/3236784

James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. 2012. A generic cyclic theorem prover. In *Asian Symposium on Programming Languages and Systems.* Springer, 350–367. https://doi.org/10.1007/978-3-642-35182-2_25

Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. 2009. Looper: Lightweight detection of infinite loops at runtime. In *2009 IEEE/ACM International Conference on Automated Software Engineering.* IEEE, 161–169. https://doi.org/10.1109/ASE.2009.87

Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. 2021. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *NSDI.* 133–153.

Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and escaping infinite loops with Jolt. In *European Conference on Object-Oriented Programming.* Springer, 609–633. https://doi.org/10.1007/978-3-642-22655-7_28

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2012. HipSpec: Automating Inductive Proofs of Program Properties. In *ATx/WInG@ IJCAR.* Citeseer, 16–25. https://doi.org/10.29007/3qwr

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction.* Springer, 392–406. https://doi.org/10.1007/978-3-642-38574-2_27

Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Disproving termination with overapproximation. In *2014 Formal Methods in Computer-Aided Design (FMCAD).* IEEE, 67–74. https://doi.org/10.1109/FMCAD.2014.6987597

Lucas Dixon and Jacques Fleuriot. 2003. IsaPlanner: A prototype proof planner in Isabelle. In *International Conference on Automated Deduction.* Springer, 279–283. https://doi.org/10.1007/978-3-540-45085-6_22

Conal Elliot. 2010. Non-strict memoization. http://conal.net/blog/posts/nonstrict-memoization

Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2019. Relational symbolic execution. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 1–14. https://doi.org/10.1145/3354166.3354175

Andrew Farmer, Neil Sculthorpe, and Andy Gill. 2015. Reasoning with the HERMIT: tool support for equational reasoning on GHC core programs. *ACM SIGPLAN Notices* 50, 12 (2015), 23–34. https://doi.org/10.1145/2887747.2804303

Jeremy Gibbons and Graham Hutton. 2005. Proof methods for corecursive programs. *Fundamenta Informaticae* 66, 4 (2005), 353–366. https://dl.acm.org/doi/abs/10.5555/1227189.1227192

Andrew D. Gordon. 1995. A tutorial on co-induction and functional programming. *Functional Programming, Glasgow 1994* (1995), 78–95. https://doi.org/10.1007/978-1-4471-3573-9_6

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 147–158. https://doi.org/10.1145/1328438.1328459

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 411–424. https://doi.org/10.1145/3314221.3314618

Moa Johansson, Lucas Dixon, and Alan Bundy. 2010. Case-analysis for rippling and inductive proof. In *International Conference on Interactive Theorem Proving*. Springer, 291–306. https://doi.org/10.1007/978-3-642-14052-5_21

Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. 2012. Bolt: on-demand infinite loop escape in unmodified binaries. *ACM SIGPLAN Notices* 47, 10 (2012), 431–450. https://doi.org/10.1145/2398857.2384648

John Kolesar, Ruzica Piskac, and William Hallahan. 2022. Checking Equivalence in a Non-strict Language: Artifact. Zenodo. https://doi.org/10.5281/zenodo.7083308

Vasileios Koutavas and Mitchell Wand. 2006. Small bisimulations for reasoning about higher-order imperative programs. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 141–152. https://doi.org/10.1145/1111037.1111050

Dexter Kozen and Alexandra Silva. 2017. Practical coinduction. *Mathematical Structures in Computer Science* 27, 7 (2017), 1132–1152. https://doi.org/10.1017/S0960129515000493

Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428257

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 81–92. https://doi.org/10.1145/360204.360210

K. Rustan M. Leino and Michał Moskal. 2014. Co-induction simply. In *International Symposium on Formal Methods*. Springer, 382–398. https://doi.org/10.1007/978-3-319-06410-9_27

Dorel Lucanu and Grigore Roşu. 2007. CIRC: A circular coinductive prover. In *International Conference on Algebra and Coalgebra in Computer Science*. Springer, 372–378. https://doi.org/10.1007/978-3-540-73859-6_25

Lykourgos Mastorou, Nikolaos Papaspyrou, and Niki Vazou. 2022. Coinduction Inductively: Mechanizing Coinductive Proofs in Liquid Haskell. In *Haskell Symposium*. https://nikivazou.github.io/static/Haskell22/coinduction.pdf

Dragana Milovancevic, Julie Giunta, and Viktor Kuncak. 2021. *On Proving and Disproving Equivalence of Functional Programming Assignments*. Technical Report.

Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 845–859. https://doi.org/10.1145/3314221.3314643

Phúc C. Nguyễn and David Van Horn. 2015. Relatively complete counterexamples for higher-order programs. *ACM SIGPLAN Notices* 50, 6 (2015), 446–456. https://doi.org/10.1145/2813885.2737971

Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 226–237. https://doi.org/10.1145/1453101.1453131

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, Vol. 1. 203–233.

Simon L. Peyton Jones. 1996. Compiling Haskell by program transformation: A report from the trenches. In *European Symposium on Programming*. Springer, 18–44. https://doi.org/10.1007/3-540-61055-3_27

Grigore Roşu and Dorel Lucanu. 2009. Circular coinduction: A proof theoretical foundation. In *International Conference on Algebra and Coalgebra in Computer Science*. Springer, 127–144. https://doi.org/10.1007/978-3-642-03741-2_10

Jan J.M.M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theoretical computer science* 249, 1 (2000), 3–80. https://doi.org/10.1016/S0304-3975(00)00056-6

Davide Sangiorgi. 2009. On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 4 (2009), 1–41. https://doi.org/10.1145/1516507.1516510

Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2007. Environmental bisimulations for higher-order languages. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE, 293–302. https://doi.org/10.1109/LICS.2007.17

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316. https://doi.org/10.1145/2490301.2451150

Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. 2016. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *International Conference on Integrated Formal Methods*. Springer, 311–325. https://doi.org/10.1007/978-3-319-33693-0_20

Calvin Smith and Aws Albarghouthi. 2019. Program synthesis with equivalence reduction. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 24–47. https://doi.org/10.1007/978-3-030-11245-5_2

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 407–421. https://doi.org/10.1007/978-3-642-28756-5_28