JFP **35**, e19, 65 pages, 2025.

doi:10.1017/S0956796825100099

Checking equivalence in a non-strict language

JOHN CHARLES KOLESAR
Yale University, New Haven, CT 06520, USA
(*e-mail:* john.kolesar@yale.edu)

RUZICA PISKAC
Yale University, New Haven, CT 06520, USA
(*e-mail:* ruzica.piskac@yale.edu)

WILLIAM TRIEST HALLAHAN
Binghamton University, Binghamton, NY 13902, USA
(*e-mail:* whallahan@binghamton.edu)

Abstract

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.

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 (Peyton Jones, 1996; Peyton Jones *et al*., 2001; Benton, 2004). 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 (Schkufza *et al*., 2013; Smith & Albarghouthi, 2019; Campbell *et al.*, 2021), and automatic evaluation of students' submissions for programming assignments (Milovancevic *et al.*, 2021).

```
data Nat = S Nat | Z

Z     - _     = Z
x     - Z     = x
(S x) - (S y) = x - y
```

Fig. 1: Subtraction for natural numbers.

Non-strict languages allow for the use of conceptually infinite data structures. Infinite data structures have a number of uses, from memoization (Elliott, 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, shown in Figure 1. 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 automated equivalence checker for programs in a *non-strict functional language*. Existing approaches for fully automated equivalence checking (Dixon & Fleuriot, 2003; Claessen *et al.*, 2012; Sonnex *et al.*, 2012; Farina *et al.*, 2019) assume that all input values are finite, produce no errors, and always terminate when evaluated. 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 incompletely-defined 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 equivalences between programs in the non-strict functional language. This calculus also incorporates a sound approach for using auxiliary equivalence lemmas that allow a sub-expression to be rewritten as a different equivalent sub-expression. We show that, while lemma applications are actually *unsound* in general, we can use them soundly if we enforce certain restrictions.

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 on 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 all of the properties that are still true in a non-strict context (i.e. 28% of the entire suite), and we find counterexamples for every property that no longer holds (72% of the suite) as well. Furthermore, we evaluate NEBULA's ability to identify counterexamples involving non-termination and find that our tool can generate non-terminating counterexamples for 98% of the applicable benchmarks. Additionally, we describe an approach for accommodating error-free 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 we prove the soundness of the calculus.

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

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.

We include the proofs of important theorems for our formalism, but, for the sake of readability, we defer some of the less significant proofs to the Appendix.

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 2. 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 a b = min a b === a
prop33_rhs a b = a <= b

Z     === Z     = True
Z     === _     = False
(S _) === Z     = False
(S x) === (S y) = x === y

min Z     y     = Z
min (S x) Z     = Z
min (S x) (S y) = S (min x y)

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

Fig. 2: Zeno Theorem 33.

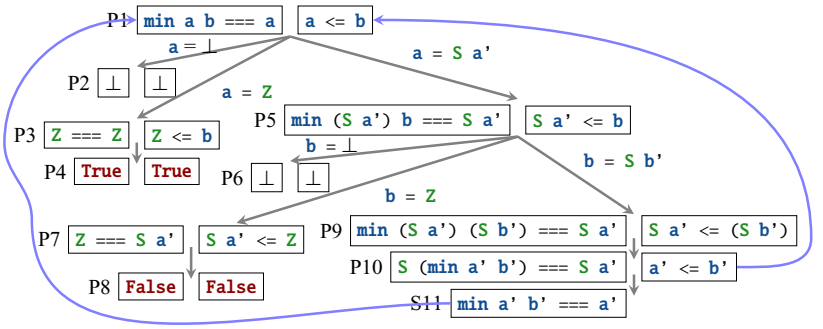

Fig. 3: Overview of how NEBULA proves `prop33`. Gray arrows denote symbolic execution, and blue arrows denote coinduction.

`prop33_lhs` and `prop33_rhs` on any inputs `a` and `b`, including inputs that are infinite or incompletely defined, will produce the same output.

Figure 3 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. To evaluate ===, we must evaluate `min a b` first, which we cannot do unless we know 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` as `Z`, in *P5* we concretize `a` as `S a'`, where `a'` is a fresh symbolic variable, and in *P2*, we concretize `a` as ⊥, 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 as ⊥ in *P6*, as `Z` in *P7*, and as `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 that 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`) have a similar structure. 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.

Our application of coinduction here may look similar to induction, but the two proof techniques are distinct. Informally, what distinguishes our proof from an inductive proof is the fact that it does not rely on the assumption that evaluation will reach a terminal base case eventually. Our conclusion that `min a b === a` and `a <= b` will behave equivalently on the path with the cycle holds even if the recursion loops forever. An inductive proof about recursive programs would not take the possibility of non-termination into account. We describe our use of coinduction in more detail in Section 4.2.

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 3, we presented only relevant steps in the proof, and we omitted 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.

```
prop01_lhs n xs = take n xs ++ drop n xs
prop01_rhs n xs = xs

data [a] = [] | a : [a]

(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

take Z _ = []
take _ [] = []
take (S x) (y:ys) = y : (take x ys)

drop Z xs = xs
drop _ [] = []
drop (S x) (_:xs) = drop x xs
```

Fig. 4: Zeno Theorem 1.

Example 2.2. Next, we consider the formula `prop01` from the Zeno evaluation suite (Sonnex *et al.*, 2012). In Figure 4, 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*. We say that a data structure is total if it is not $\perp$ and contains no occurrences of $\perp$ as sub-expressions. If `n` is allowed to be non-total, NEBULA finds a counterexample, with `n` as $\perp$ and `xs` as `Z:[]`. The expression `take` $\perp$ `(Z:[])` simplifies to $\perp$, and the expression $\perp$ `++ drop` $\perp$ `(Z:[])` also simplifies to $\perp$ because of its first argument. At the same time, the right-hand side is `Z:[]`, which is a total expression.

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. A more detailed explanation of the concept of totality appears in Section 6.6.

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 5.

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. To discharge *P12*, we must prove that `take n' xs' ++ drop (S n') (x:xs')` is equivalent to `xs'`.

It might look tempting to apply coinduction between *P12* and *P1*, but 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*.

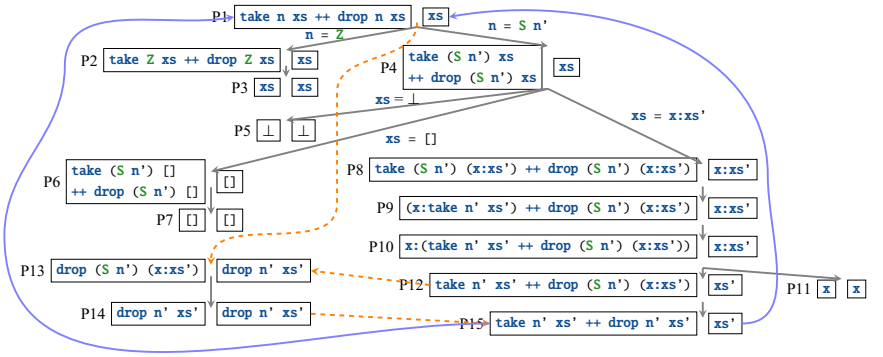

Fig. 5: Overview of how NEBULA proves **prop01**. Gray arrows denote symbolic execution, blue arrows denote coinduction, and orange dashed arrows denote lemma generation or usage.

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* 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 `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 final 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. To evaluate `m - m`, NEBULA must concretize `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 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 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 we can follow the same reduction steps that `m - m` took to reduce to `m' - m'` over again. More specifically, `m' - m'` can reduce to `m'' - m''`, where `m' = S m''`, and the process can repeat forever, resulting in non-termination. On the other hand, `Z` has already terminated. Mapping `m' - m'` to `m - m` requires us to replace `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`.

e	$::=$		**Expressions**
	$\mid$	x	variable
	$\mid$	s	symbolic variable
	$\mid$	$\lambda x \,.\, e$	lambda
	$\mid$	D	data constructor
	$\mid$	$e\, e$	application
	$\mid$	$\texttt{case}\, e\, \texttt{of}\, \{\vec{a}\}$	case
	$\mid$	$\bot^L$	bottom
a	$::=$	$D\, \vec{x} \to e$	**Alternatives**

Fig. 6: The language considered by NEBULA.

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 6 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*. 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 $=$ to check syntactic equality of expressions. We write $e' \preccurlyeq e$ to indicate that e' is a sub-expression of e. (We interpret the sub-expression relation as being reflexive.) The expression $e\,[e_2 \,/\, e_1]$ denotes e with each occurrence 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 abstraction or data constructor application. Correspondingly, symbolic execution reduces expressions to *Symbolic Weak Head Normal Form (SWHNF)*. SWHNF is defined as follows:

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

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

States Symbolic execution operates on *states* of the form $(e,\ Y)$, where 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. We write $\texttt{lookup}(s, Y)$ to denote the data constructor application that Y contains for s, and $Y\{s \to D\,\vec{s}\}$ denotes the symbolic store with s mapped to $D\,\vec{s}$ and all other mappings from Y preserved.

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 7 shows the reduction rules. The rules for lambdas and applications are standard. The VAR rule fetches the definitions of non-symbolic variables and functions (such as the definitions of **`min`** and <= in Example 2.1) from an implicit environment. These definitions may be recursive, but they cannot contain symbolic variables. We assume the existence of a function `lookupVar` that serves as an interface with the implicit environment, and we assume that every non-symbolic variable has a mapping in the environment. A case expression $\texttt{case}\ e\ \texttt{of}\ \{\vec{a}\}$ 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 rule LKDC selects the appropriate case statement branch to continue evaluation. If $s \notin Y$, evaluation is still possible for the state: 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.

$$\textsc{Var}\ \frac{}{(x,\ Y) \hookrightarrow (\texttt{lookupVar}(x),\ Y)} \qquad \textsc{App}\ \frac{(f,\ Y) \hookrightarrow (f',\ Y')}{(f\ e,\ Y) \hookrightarrow (f'\ e,\ Y')}$$

$$\textsc{App}\lambda\ \frac{}{((\lambda x\,.\,e')\ e,\ Y) \hookrightarrow (e'\ [e\,/\,x],\ Y)}$$

$$\textsc{CsEv}\ \frac{(e,\ Y) \hookrightarrow (e',\ Y')}{(\texttt{case}\ e\ \texttt{of}\ \{\vec{a}\},\ Y) \hookrightarrow (\texttt{case}\ e'\ \texttt{of}\ \{\vec{a}\},\ Y')}$$

$$\textsc{CsDC}\ \frac{}{(\texttt{case}\ D\,\vec{e}\ \texttt{of}\ \{D\,\vec{x} \to e_a; \ldots\},\ Y) \hookrightarrow (e_a\ [\vec{e}\,/\,\vec{x}],\ Y)}$$

$$\textsc{FrDC}\ \frac{s \notin Y \qquad \vec{s}\ \text{fresh}}{(\texttt{case}\ s\ \texttt{of}\ \{D\,\vec{x} \to e_a; \ldots\},\ Y) \hookrightarrow (e_a\ [\vec{s}\,/\,\vec{x}],\ Y\{s \to D\,\vec{s}\})}$$

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

$$\textsc{BtDC}\ \frac{L\ \ \texttt{fresh}}{(\texttt{case}\ s\ \texttt{of}\ \{D\,\vec{x} \to e_a; \ldots\},\ Y) \hookrightarrow (\bot^L,\ Y\{s \to \bot^L\})}$$

$$\textsc{BtApp}\ \frac{}{(\bot^L\ e,\ Y) \hookrightarrow (\bot^L,\ Y)} \qquad \textsc{BtCs}\ \frac{}{(\texttt{case}\ \bot^L\ \texttt{of}\ \{\vec{a}\},\ Y) \hookrightarrow (\bot^L,\ Y)}$$

Fig. 7: Reduction rules.

```
prop23_lhs a b = max a b
prop23_rhs a b = max b a

max x y = case x of
  Z -> y
  S x' -> case y of
    Z -> x
    S y' -> S (max x' y')
```

Fig. 8: Zeno Theorem 23.

The reduction rules BtApp and BtCs force any expression which must evaluate $\bot^L$ to reduce to $\bot^L$ itself. The rule 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.

Example 3.1. Consider the property **prop23** from the Zeno evaluation suite (Sonnex *et al.*, 2012), shown in Figure 8. The property asserts that the **max** function over natural numbers is commutative. If we ignore the labels on bottom values, then **prop23_lhs** and **prop23_rhs**

must be equivalent, and `prop23` is valid. However, the equivalence does not hold if we take labels into consideration. If we concretize `a` as $\perp^A$ and `b` as $\perp^B$, then `max a b` and `max b a` reduce to different values. The former reduces to $\perp^A$, and the latter reduces to $\perp^B$. The reason for the difference in the two sides' results is that `max a b` branches on the value of `a` before it branches on the value of `b`, whereas `max b a` does the opposite. Because the outer case statement evaluates to an error on each side, the value of the variable in the inner case statement is never used.

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 6.5.

Importantly, we do not have any reduction rules for expressions that are in SWHNF. There is no rule that allows us to derive $(D\,e,\ Y) \hookrightarrow (D\,e',\ Y')$ from $(e,\ Y) \hookrightarrow (e',\ Y')$. Likewise, there is no rule to derive $(\lambda x\,.\,e,\ Y) \hookrightarrow (\lambda x\,.\,e',\ Y')$ from $(e,\ Y) \hookrightarrow (e',\ Y')$. In a non-strict language, we cannot evaluate the arguments of a data constructor application without first eliminating the data constructor, and we cannot evaluate the body of a lambda abstraction without first applying the lambda to an argument. We want to capture the same behavior in our own formalism.

We assume that all case expressions are exhaustive for the types of their scrutinees, so evaluation cannot become stuck at a case expression that has no branch to match its scrutinee. Correspondingly, we assume that an individual case expression cannot have multiple branches for the same data constructor, so evaluation cannot branch nondeterministically at a case statement unless the scrutinee is a non-concretized symbolic variable. Additionally, we assume that scrutinees of case expressions cannot be function-typed. Since we also do not include any reduction rules that apply to SWHNF expressions, we can guarantee that an expression in our formalism has a reduction rule that applies to it if and only if the expression is not in SWHNF:

Theorem 1. *For any expression e and symbolic store Y, there exist an expression e' and symbolic store Y' such that $(e, Y) \hookrightarrow (e', Y')$ if and only if e is not in SWHNF.*

Proof See the Appendix. ■

Reduction Sequences For our proofs about reduction, we need to rely on the concept of *reduction sequences*. A *finite reduction sequence* $S_{\hookrightarrow} = S_1, \ldots S_k$ is a sequence of states such that $\forall i, 1 \leq i < k . S_i \hookrightarrow S_{i+1}$. Similarly, an *infinite reduction sequence* $S_{\hookrightarrow} = S_1, \ldots$ is an infinite sequence of states such that $\forall i, 1 \leq i . S_i \hookrightarrow S_{i+1}$. We use the term *reduction sequence* when the distinction between a finite and infinite reduction sequence is not significant. Also, we sometimes need to reason about the simultaneous reduction of two expressions, so we introduce the concept of *paired reduction sequences*. A paired reduction sequence is a sequence of triples of two expressions and one symbolic store, $S_{\hookrightarrow} = (e_1^1,\ e_1^2,\ Y_1), \ldots, (e_k^1,\ e_k^2,\ Y_k), \ldots$ such that

$$\begin{aligned}\forall i, 1 \leq i . ((e_i^1,\ Y_i) \hookrightarrow (e_{i+1}^1,\ Y_{i+1}) \wedge e_i^2 = e_{i+1}^2)\\ \vee ((e_i^2,\ Y_i) \hookrightarrow (e_{i+1}^2,\ Y_{i+1}) \wedge e_i^1 = e_{i+1}^1)\end{aligned}$$

holds for every entry in the sequence.

$$\sqsubseteq\text{-EVAL}\ \frac{\exists e'.(e_1,\ Y_1) \hookrightarrow^* (e',\ Y_1) \wedge (e',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)}{(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)}$$

$$\sqsubseteq\text{-SYM0}\ \frac{\exists e = \texttt{lookup}(s, Y_1) \qquad (e,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)}{(s,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)}$$

$$\sqsubseteq\text{-SYM1}\ \frac{\begin{array}{c}\exists e' = \texttt{lookup}(s, V), e''.(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1) \wedge (e_1,\ Y_1) \sqsubseteq_V (e'',\ Y_2) \\ \exists e = \texttt{lookup}(s, Y_2) \qquad\qquad (e_1,\ Y_1) \sqsubseteq_V (e,\ Y_2)\end{array}}{(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)}$$

$$\sqsubseteq\text{-SYM2}\ \frac{s \notin Y_2 \qquad \exists e = \texttt{lookup}(s, V), e'.(e,\ Y_1) \hookrightarrow^* (e',\ Y_1) \wedge (e_1,\ Y_1) \sqsubseteq_V (e',\ Y_2)}{(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)}$$

$$\sqsubseteq\text{-VAR}\ \frac{}{(x,\ Y_1) \sqsubseteq_V (x,\ Y_2)} \qquad \sqsubseteq\text{-LAM}\ \frac{(e_1,\ Y_1) \sqsubseteq_V (e_2[x_1/x_2],\ Y_2)}{(\lambda x_1\,.\,e_1,\ Y_1) \sqsubseteq_V (\lambda x_2\,.\,e_2,\ Y_2)}$$

$$\sqsubseteq\text{-CASE}\ \frac{\begin{array}{c}(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2) \\ \forall (D\,\vec{x_1} \to e_1^a) \in \vec{a}_1, (D\,\vec{x_2} \to e_2^a) \in \vec{a}_2.(e_1^a,\ Y_1) \sqsubseteq_V (e_2^a[\vec{x_1}/\vec{x_2}],\ Y_2)\end{array}}{(\texttt{case}\ e_1\ \texttt{of}\ \{\vec{a}_1\},\ Y_1) \sqsubseteq_V (\texttt{case}\ e_2\ \texttt{of}\ \{\vec{a}_2\},\ Y_2)}$$

$$\sqsubseteq\text{-DC}\ \frac{}{(D,\ Y_1) \sqsubseteq_V (D,\ Y_2)} \qquad \sqsubseteq\text{-APP}\ \frac{\begin{array}{c}(e_1,\ Y_1) \sqsubseteq_V (e_1',\ Y_2) \\ (e_2,\ Y_1) \sqsubseteq_V (e_2',\ Y_2)\end{array}}{(e_1\,e_2,\ Y_1) \sqsubseteq_V (e_1'\,e_2',\ Y_2)}$$

$$\sqsubseteq\text{-BT}\ \frac{}{(\bot^L,\ Y_1) \sqsubseteq_V (\bot^L,\ Y_2)}$$

Fig. 9: Approximation definition.

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 and is the same as S' otherwise. Different occurrences of the same symbolic variable within S' need to have the same replacement within S.

We formalize $\sqsubseteq_V$ in Figure 9. The relation $S \sqsubseteq_V S'$ 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 $\exists V.S \sqsubseteq_V S'$.

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 side of the relation. The rule $\sqsubseteq$-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 $(e_1,\ Y_1) \sqsubseteq_V (e',\ Y_2)$. The rule $\sqsubseteq$-SYM1 is similar to $\sqsubseteq$-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 rule $\sqsubseteq$-SYM0 handles concretized symbolic variables on the left-hand side: if there is a mapping for a symbolic variable in the left-hand symbolic store, we can use that mapping as a replacement for the symbolic variable itself. The final rule of interest is $\sqsubseteq$-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 $\sqsubseteq_V$.

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. Even though it is impossible to have a complete calculus of rules for proving approximation relations, our formalization of symbolic execution is complete in a different sense. Our formalization of $\sqsubseteq$ carefully ensures that symbolic execution explores all paths through a program. If a state S_2 approximates another state S_1, evaluation steps on S_1 cannot break the connection to S_2 without replacing it with a new connection to a state to which S_2 evaluates. Consequently, we can use approximation to verify properties of programs. We state this formally as Theorem 2:

Theorem 2 (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 $S_1' \sqsubseteq S_2'$.*

Proof Consider a state $S_1 = (e_1,\ Y_1)$ and a state $S_2 = (e_2,\ Y_2)$ such that $S_1 \sqsubseteq S_2$. We will show that if S_1 is reduced by a single application of $\hookrightarrow$, so that $S_1 \hookrightarrow S_1'$, then either (1) $S_1' \sqsubseteq S_2$ or (2) there exists a reduction $S_2 \hookrightarrow S_2'$ such that $S_1' \sqsubseteq S_2'$.

Use of Induction In many cases, we rely on induction on the size of the expressions in states S_1 and S_2. This results in new values $V', e_1', e_2', Y_1', Y_2'$, which we must use in the application of $\sqsubseteq$ to the larger expression. For most expressions, $(e_1',\ Y_1') \sqsubseteq_{V'} (e_2',\ Y_2')$ holding relies on the fact that the only rule that requires modifying V to add a variable is FRDC, which only results in mappings for *fresh* variables being added to V. (The rule BTDC modifies the symbolic store, but it does not affect V: V maps symbolic variables on the right-hand side of $\sqsubseteq$ to expressions on the left-hand side, and concretizing a symbolic variable as a bottom does not introduce any new symbolic variables on the right-hand side.) Thus, Lemma 4 from the Appendix ensures that $(e_1',\ Y_1) \sqsubseteq_{V'} (e_2',\ Y_2)$ continues to hold, except in the case where e_2' is or contains a symbolic variable. We will consider this special case in the following, when discussing symbolic variables.

In one case, we also apply this theorem inductively on a usage of $\sqsubseteq$ on the right-hand side of the definition of $\sqsubseteq$. To see why this is justified, notice that for $S_1 \sqsubseteq S_2$ to hold, this case may be used only a finite number of times (to a finite depth). Thus, in the base case, we have applied any other piece of $\sqsubseteq$'s definition.

Case Analysis We will now enumerate the cases in which $S_1 \sqsubseteq_V S_2$ holds and justify the theorem in each case.

Reduction on Left Suppose $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ holds because of $\sqsubseteq$-EVAL, whose premise is $\exists e'.(e_1,\ Y_1) \hookrightarrow^* (e',\ Y_1) \wedge (e',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$. There are two possibilities. The first possibility is that the number of reduction steps required to reduce $(e_1,\ Y_1)$ to $(e',\ Y_1)$ is 0 (that is, $e_1 = e'$). In this case, $(e',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ must hold because of some other piece

of the approximation definition, and we refer to the relevant piece of the proof to justify the theorem. The second possibility is that the number of reduction steps is nonzero. In this case, the reduction $(e_1,\ Y_1) \hookrightarrow^* (e',\ Y_1)$ must be deterministic, since Y_1 is not updated. Thus, when we reduce e_1 to some e_1' ($(e_1,\ Y_1) \hookrightarrow (e_1',\ Y_1)$), it must be the case that $\exists e''.(e_1',\ Y_1) \hookrightarrow^* (e'',\ Y_1) \wedge (e'',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$, and so $(e_1',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ by $\sqsubseteq$-EVAL.

Symbolic Variables If e_1 is a symbolic variable, it cannot be reduced, and so the theorem does not apply. There are two cases involving a symbolic variable on the right-hand side in which e_1 can be reduced.

First, consider the possibility that $(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)$ and $\exists e = \texttt{lookup}(s, Y_2)$. The rule from the definition of $\sqsubseteq_V$ being applied must be $\sqsubseteq$-SYM1, so it must be the case that $\exists e' = \texttt{lookup}(s, V), e''.(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1)$ and $(e_1,\ Y_1) \sqsubseteq_V (e'',\ Y_2)$ and $(e_1,\ Y_1) \sqsubseteq_V (e,\ Y_2)$. Because e comes from a symbolic store, it is in SWHNF, so it must be the case that e_1 either is already in SWHNF, or can be reduced to SWHNF deterministically. If e_1 is already in SWHNF, then it cannot be reduced, so the theorem holds vacuously. Consequently, let e_1' be the SWHNF expression distinct from e_1 to which e_1 can be reduced. There must be at least one step of reduction that e_1 can take, so let e_1^s be the expression such that $(e_1,\ Y_1) \hookrightarrow (e_1^s,\ Y_1)$. Since e is in SWHNF and cannot undergo reduction, applying our inductive hypothesis to $(e_1,\ Y_1) \sqsubseteq_V (e,\ Y_2)$ gives us $(e_1^s,\ Y_1) \sqsubseteq_V (e,\ Y_2)$. Likewise, applying our inductive hypothesis to $(e_1,\ Y_1) \sqsubseteq_V (e'',\ Y_2)$ gives us a disjunction: either $(e_1^s,\ Y_1) \sqsubseteq_V (e'',\ Y_2)$ or there exists some state $(e''',\ Y_2)$ to which $(e'',\ Y_2)$ steps such that $(e_1^s,\ Y_1) \sqsubseteq_V (e''',\ Y_2)$.

If $(e'',\ Y_2)$ does not need to undergo an evaluation step, then we have everything we need to apply $\sqsubseteq$-SYM1. With $(e_1^s,\ Y_1) \sqsubseteq_V (e'',\ Y_2)$ and $(e_1^s,\ Y_1) \sqsubseteq_V (e,\ Y_2)$ and the premises from before, we can conclude that $(e_1^s,\ Y_1) \sqsubseteq_V (s,\ Y_2)$. If $(e_1^s,\ Y_1) \sqsubseteq_V (e''',\ Y_2)$ holds instead, then we need to change some of the premises we use for $\sqsubseteq$-SYM1. (We know that the symbolic store does not change for the reduction $(e'',\ Y_2) \hookrightarrow (e''',\ Y_2)$ because no new mappings can be introduced for e_1 during its deterministic reduction to SWHNF. If it were possible to introduce a new mapping for Y_2, the original approximation $(e_1,\ Y_1) \sqsubseteq_V (e'',\ Y_2)$ would not have held.) We still know that $(e_1^s,\ Y_1) \sqsubseteq_V (e,\ Y_2)$ holds. Since the reduction $(e'',\ Y_2) \hookrightarrow (e''',\ Y_2)$ does not modify the symbolic store, we can say that $(e'',\ Y_1)$ steps to $(e''',\ Y_1)$. (We know that all of the relevant symbolic variables for the step $(e'',\ Y_2) \hookrightarrow (e''',\ Y_2)$ are present in Y_1 because of the determinism of the reduction for e_1.) Chaining this together with $(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1)$ gives us $(e',\ Y_1) \hookrightarrow^* (e''',\ Y_1)$, so we can apply $\sqsubseteq$-SYM1.

Second, consider the possibility that $(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)$ and $s \notin Y_2$. The $\sqsubseteq_V$ rule for this situation is $\sqsubseteq$-SYM2, so it must be the case that $\exists e' = \texttt{lookup}(s, V), e''.(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1)$, where $(e_1,\ Y_1) \sqsubseteq_V (e'',\ Y_2)$. Suppose that $(e_1,\ Y_1) \hookrightarrow (e_1',\ Y_1)$ for some Y_1. (We know that the evaluation step cannot change the symbolic store because $(e_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)$ and s is in SWHNF. As we noted for the previous part, this means that e_1 must reduce to SWHNF deterministically.) By induction (as justified in the note at the start of the proof), we know that $(e_1',\ Y_1) \sqsubseteq_V (e'',\ Y_2)$ holds or that there exists some state $(e''',\ Y_2)$ to which $(e'',\ Y_2)$ steps such that $(e_1',\ Y_1) \sqsubseteq_V (e''',\ Y_2)$.

For the first subcase, assume that $(e_1',\ Y_1) \sqsubseteq_V (e'',\ Y_2)$. We know from before that $(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1)$, so we have everything we need to apply $\sqsubseteq$-SYM2 to derive $(e_1',\ Y_1) \sqsubseteq_V (s,\ Y_2)$. For the second subcase, assume that $(e'',\ Y_2) \hookrightarrow (e''',\ Y_2)$ and that $(e_1',\ Y_1) \sqsubseteq_V (e''',\ Y_2)$. We know that the reduction step for e'' does not modify the symbolic

store because of the determinism of the reduction for e_1. As in the previous part, if it were possible to introduce a new mapping in the reduction of e'', the original approximation $(e_1,\ Y_1) \sqsubseteq_V (e'',\ Y_2)$ would not have held. Since the evaluation is deterministic and we know from earlier that $(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1)$, it follows that $(e',\ Y_1) \hookrightarrow^* (e''',\ Y_1)$. This gives us what we need to apply $\sqsubseteq$-SYM2 to derive $(e'_1,\ Y_1) \sqsubseteq_V (s,\ Y_2)$.

Variables The only rule that can reduce a variable is VAR, which looks up the variable's definition. The definition e of a non-symbolic variable cannot contain symbolic variables, so the theorem holds by induction over the structure of e.

Application Suppose $e_2 = e_2^1\ e_2^2$. The $\sqsubseteq$ rule that applies is $\sqsubseteq$-APP, so we know that $e_1 = e_1^1\ e_1^2$ and $e_1^1 \sqsubseteq_V e_2^1$ and $e_1^2 \sqsubseteq_V e_2^2$. If e_2 is already in SWHNF, the theorem holds trivially. Thus, we consider the two possible ways e_1 may be reduced:

- If APP is applied, then $(e_2^1,\ Y_2) \hookrightarrow (e_2^{1'},\ Y_2')$. The theorem holds by induction on e_1^1 and e_2^1.
- Now suppose the rule APPλ can be applied. Then e_2^1 has the form $\lambda x_2\,.\,e_2^b$ and e_1^1 has the form $\lambda x_1\,.\,e_1^b$, The rule $\sqsubseteq$-LAM gives us that $(e_1^b,\ Y_1) \sqsubseteq_V (e_2^b\,[x_1\,/\,x_2],\ Y_2)$. Then, by Lemma 3 from the Appendix, $(e_1^b\,[e_1^2\,/\,x_1],\ Y_1) \sqsubseteq_V (e_2^b\,[e_2^2\,/\,x_2],\ Y_2)$. Thus $(e_1^b\,[e_1^2\,/\,x_1],\ \{\}) \sqsubseteq_V (e_2^b\,[e_2^2\,/\,x_2],\ Y_2)$ and the theorem is satisfied.

Cases Suppose $e_2 = \texttt{case}\ e_2^b\ \texttt{of}\ \{\vec{a}_2\}$. Then the rule $\sqsubseteq$-CASE applies, and the left-hand expression must have the form $e_1 = \texttt{case}\ e_1^b\ \texttt{of}\ \{\vec{a}_1\}$, where there exists some V such that $(e_1^b,\ Y_1) \sqsubseteq_V (e_2^b,\ Y_2)$ and $\forall (D\,\vec{x}_1 \to e_1^a) \in a_1, (D\,\vec{x}_2 \to e_2^a) \in a_2.(e_1^a,\ Y_1) \sqsubseteq_V (e_2^a[\vec{x}_1/\vec{x}_2],\ Y_2)$. There are four rules that might be applicable to reduce the right-hand side. The rules CSEV and CSDC simply require an inductive argument on the application of $\hookrightarrow$, so we focus on FRDC and LKDC.

First consider FRDC. We assume some $s_1 \notin Y_1$, where $(\texttt{case}\ s_1\ \texttt{of}\ \{\vec{a}_1\},\ Y_1) \sqsubseteq_V (\texttt{case}\ s_2\ \texttt{of}\ \{\vec{a}_2\},\ Y_2)$. Suppose $s_2 \in Y_2$. Then, s_2 must be mapped to some $D\,e_1^a \ldots e_k^a$, and by the definition of $\sqsubseteq$, it must be that there exists some e' that deterministically reduces to both s_1 and $D\,e_1^a \ldots e_k^a$. This contradicts our assumption that $s_1 \notin Y_1$, so $s_2 \notin Y_2$. Thus, it is easy to see from the definition of $\sqsubseteq$ that if FRDC is used to reduce the left-hand state, it can be applied to the right-hand state to instantiate s_2 with the same constructor and preserve the approximation. A new V' must be constructed, which maps the fresh variables in the right-hand state to the corresponding fresh variables in the left-hand state.

Now consider LKDC. We assume there exists some $e = D\,s_1^a \ldots s_k^a = \texttt{lookup}(s_1, Y_1)$, where $(\texttt{case}\ s_1\ \texttt{of}\ \{\vec{a}_1\},\ Y_1) \sqsubseteq_V (\texttt{case}\ e_2^b\ \texttt{of}\ \{\vec{a}_2\},\ Y_2)$. By the definition of $\sqsubseteq$, it must be the case that $s_1 \sqsubseteq_V e_2^b$. The definition of $\sqsubseteq$ also gives us that e_2^b must either be a symbolic variable or a data constructor application. There are three possible ways we will preserve the mapping, depending on the right-hand state:

- First, suppose e_2^b is a symbolic variable s with no mapping in Y_2. In this case, FRDC can be applied to pick the same constructor that e has (or, if e is $\bot$, BTDC can be applied.) A new V' must be constructed, which maps the symbolic variables introduced by the rule to the corresponding arguments of e.
  Our use of induction and the possibility of s appearing at multiple places in e_2^b require that we justify that, for all e, assuming $(e,\ Y_1) \sqsubseteq (s,\ Y_2)$ held when $s \notin Y_2$, then $(e,\ Y_1') \sqsubseteq (s,\ Y_2\{s \to D\,s_1 \ldots s_n\})$ still holds after an application of FRDC or

BTDC on the right-hand side of the expression. The $\sqsubseteq$ rule that applies is $\sqsubseteq$-SYM2, so we know that there exists $e' = \texttt{lookup}(s, V)$ such that $(e',\ Y_1) \hookrightarrow^* (e,\ Y_1)$. In order for FRDC (or BTDC) to be applied on the right-hand side to map s to $D\, s_1 \ldots s_n$ (or $\bot$), it must hold that, in the scrutinee of the case statement, $(e_c,\ Y_1) \sqsubseteq_V (s,\ Y_2)$ is being checked. Thus, it must also be the case that $(e',\ Y_1) \hookrightarrow^* (e_c,\ Y_1)$ and that $(e_c,\ Y_1) \hookrightarrow^* (e'_c,\ Y_1)$, where $(e'_c,\ Y_1) \sqsubseteq_V (D\, s_1 \ldots s_n,\ Y_2)$ (or, in the case of BTDC, that $(e'_c,\ Y_1) \sqsubseteq_V (\bot,\ Y_2)$.) By Lemma 6 from the Appendix, it is then the case that there is only one possible reduction sequence for $(e,\ Y_1)$, specifically $(e,\ Y_1) \hookrightarrow^* (e_c,\ Y_1) \hookrightarrow (e'_c,\ Y'_1)$. Thus, $(e,\ Y'_1) \sqsubseteq (s,\ Y_2\{s \to D\, s_1 \ldots s_n\})$ holds (or a similar approximation holds in the case of an application of BTDC.)

- Now suppose e_2^b is a symbolic variable that Y_2 maps to $e' = D\, s_1^{a'} \ldots s_k^{a'}$. By the definition of $\sqsubseteq$-SYM1, it must be that $e \sqsubseteq_V e$'. Consequently, from the rule $\sqsubseteq$-DC, we know that $D\, s_1^a \ldots s_k^a \sqsubseteq_V D\, s_1^{a'} \ldots s_k^{a'}$, and thus $\forall 1 \leq i \leq k . s_i^a \sqsubseteq_V s_1^{a'}$. Then, we can apply LKDC on the right-hand side as well, and it is clear the approximation continues to hold on the reduced states, using the same V.
- Finally, suppose e_2^b is a data constructor application itself. Again, it is clear that we can apply LKDC on the right-hand side, and it is clear that the approximation continues to hold on the reduced states, using the same V.

Thus, the theorem is satisfied for case statements.

Lambdas, Constructors, Bottom Lambdas, data constructors, and bottoms are already in SWHNF, so they cannot be reduced. The theorem holds trivially in these cases. ■

Allowing arbitrary evaluation at various points is essential to ensure that Theorem 2 holds. The following example illustrates this:

Example 3.2. Consider the approximation

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

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:

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

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

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

would not hold for any V. The rule $\sqsubseteq$-SYM2 allows us to leave $V = \{s \to id\ D\}$ to preserve the approximation.

In Section 4.1, when we define our equivalence proof rules, we will include two rules for reasoning about reduction that are based on Theorem 2. Also, 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.

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):

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

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 $1 \leq i \leq k$, 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 unevaluated expressions 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:

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

The definitions of WHNF equivalence and general equivalence are mutually recursive because WHNF serves as a partial stopping point for execution in a non-strict functional language. We want to be able to reason about situations where a non-strict program runs forever but hits an infinite number of WHNF stopping points along its execution path. If the first two cases in the definition of WHNF equivalence relied on WHNF equivalence rather than general equivalence, we would lose that ability. We would only be able to establish equivalences in situations where two programs either terminate completely or diverge on a path that hits WHNF only finitely many times.

We treat bottom values with different labels as distinct because programmers might not want to treat errors from different sources as interchangeable. 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. Having two different failure-case concretization options for symbolic variables, with one representing termination with an error and the other representing non-termination, would be redundant and would not make our formalism any more expressive.

Although we view bottom values as representing either non-termination or termination with an error for symbolic variable concretization, we do not regard non-termination and

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

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

Fig. 10: Syntactic equivalence and equivalence based on splitting SWHNF expressions.

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

$$\textsc{Red-R}\ \frac{\forall (e_2',\ Y') s.t. (e_2,\ Y) \hookrightarrow (e_2',\ Y').R,\ Y', e_1 \equiv e_2'}{R,\ Y, e_1 \equiv e_2}$$

Fig. 11: Reduction equivalence rules.

termination with an error as equivalent in general. Bottom values from symbolic variables represent non-termination that comes from the arguments of a program. We want to distinguish this from non-termination that comes directly from the recursive structure of the program currently being examined. An example of the latter kind of non-termination is the cyclic evaluation of `m - m` in Example 2.3. Whenever we concretize one of the descendants of `m`, the concretization terminates with a data constructor, but the recursion of the subtraction function that consumes those data constructors never ends. In Section 5, we discuss our technique for detecting non-termination that comes from infinite recursion.

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 10 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 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 the task of checking whether e_1 and e_2 are equivalent to checking the equivalence of each matching argument pair (e_i^1, e_i^2). The rule LAM-EQUIV states that two lambdas are equivalent if their applications to a fresh symbolic value are equivalent. The rule BOT-EQUIV says that two bottoms are equivalent if they share a label. These rules follow easily from the definition of equivalence.

Reduction Rules Figure 11 shows RED-L and RED-R, which apply symbolic execution to the left and right states, respectively, being checked by the relation. The correctness of these rules follows from Theorem 2, 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

Because we aim to reason about potentially infinite expressions and non-terminating execution paths, the basis of NEBULA's approach to verification is coinduction. Coinduction is a proof technique based on the greatest fixed point of an equation. In contrast to an inductive construction, which creates a complex object from a base case and inductive steps, a coinductive proof starts by confirming 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 & Silva, 2017).

Here, we present a brief overview of the theoretical basis of coinduction. For full details, we refer readers to Gibbons & Hutton (2005), Gordon (1995), and Kozen & Silva (2017). The process of proving coinductively that two objects are equivalent involves the construction of a *bisimulation*. A bisimulation is a relation between structures, in which two structures are related only if they are still related after being reduced (Gibbons & Hutton, 2005). An individual bisimulation is defined in terms of a larger relation $\sim$, which is the greatest bisimulation. For our purposes, the greatest bisimulation $\sim$ is the greatest fixed point for state equivalence: it is the relation that relates all state pairs (S_1, S_2) such that either (1) S_1 and S_2 are equivalent when reduced to SWHNF or (2) 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 $\sim$. (This definition of equivalence is narrower than the definition that we provide at the start of the section: here we consider only non-terminating paths that lead to cycles, but the definition from the start of the section encompasses all forms of non-termination. Cycles are what we use as evidence that an execution path does not terminate. Either way, we cannot compute $\sim$ directly since program equivalence is undecidable in general, but we do not need to manipulate it directly.) The more specific bisimulation that we want to find between two programs is the union of $\sim$ with a set R of state pairs. To establish that R and $\sim$ form a valid bisimulation, we simply need to confirm that R is a consistent subset of $\sim$, or, equivalently, that $R \cup \sim$ is equal to $\sim$. The relation R is a subset of $\sim$ if every state pair (S_1, S_2) meets condition (1) or (2). If we know that R is a subset of $\sim$, then all state pairs related by R must uphold semantic equivalence. More importantly, R is consistent if, for every state pair (S_1, S_2) in R, the other state pairs that serve as evidence for the inclusion of (S_1, S_2) in R are also related by R (AbdelGawad, 2019). If S_1 and S_2 satisfy condition (1), then the state pairs that serve as evidence are the ones that confirm that their corresponding sub-expressions are equivalent. For that situation, membership of the sub-expression state pairs in R can be established either by appealing to conditions (1) and (2) recursively or by some other technique such as syntactic equality. Alternatively, if S_1 and S_2 satisfy condition (2), the evidence for their inclusion in R is the pair (S_1', S_2') of states that approximate S_1 and S_2, respectively.

$$\textsc{RAdd}\ \frac{R \cup (e_1, e_2, Y),\ Y, e_1 \equiv e_2}{R,\ Y, e_1 \equiv e_2}$$

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

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

Fig. 12: Unguarded and guarded coinduction.

We formalize our use of coinduction as the rules RAdd, U-Coind, and G-Coind in Figure 12. The rule 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. The rule U-Coind allows NEBULA to discharge a pair of expressions (e_1, e_2) and a corresponding symbolic store Y if $\neg\texttt{SWHNF}(e_1^R)$, $\neg\texttt{SWHNF}(e_2^R)$, and there is a mapping V such that $(e_1,\ Y) \sqsubseteq_V (e_1^R,\ Y^R)$ and $(e_2,\ Y) \sqsubseteq_V (e_2^R,\ Y^R)$. The rule 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 2. 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 10 and 11:

Definition 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 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.

We say that a proof tree is *productive* if it is both U-productive and G-productive. A proof tree must be productive 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.

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

$$\textsc{LemmaLeft}\ \frac{\{\},\ Y^L, e_1^L \equiv e_2^L \qquad \exists e_1' \preccurlyeq e_1 . e_1' = f\ e_1^a\ \ldots\ e_k^a \\ \exists e_1'' \preccurlyeq e_1' . (e_1'',\ Y) \sqsubseteq_V (e_1^L,\ Y^L) \qquad e_2^V = e_2^L\,[V(s)\,/\,s] \\ \neg \texttt{calls}(e_2^V, f, \{\}) \qquad R,\ Y, e_1\,[e_2^V\,/\,e_1''] \equiv e_2}{R,\ Y, e_1 \equiv e_2}$$

$$\textsc{LemmaRight}\ \frac{\{\},\ Y, e_1^L \equiv e_2^L \qquad \exists e_2' \preccurlyeq e_2 . e_2' = f\ e_1^a\ \ldots\ e_k^a \\ \exists e_2'' \preccurlyeq e_2' . (e_2'',\ Y) \sqsubseteq_V (e_2^L,\ Y^L) \qquad e_1^V = e_1^L\,[V(s)\,/\,s] \\ \neg \texttt{calls}(e_1^V, f, \{\}) \qquad R,\ Y, e_1 \equiv e_2\,[e_1^V\,/\,e_2'']}{R,\ Y, e_1 \equiv e_2}$$

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

Fig. 13: Proof rules for lemmas.

to guide an expression into a form more amenable to $\sqsubseteq$ and coinduction. In Figure 13, 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. Let e_1 and e_2 be the two main expressions that we want to prove equivalent, and let Y be their corresponding symbolic store. The first step in applying LemmaLeft is to prove some lemma $\{\},\ Y^L, e_1^L \equiv e_2^L$. To link the lemma to our main equivalence proof, we need to find some expression $e_1'' \preccurlyeq e_1$ and mapping V such that $(e_1'',\ Y) \sqsubseteq_V (e_1^L,\ Y^L)$. The sub-expression e_1'' is the part of e_1 that will receive the lemma substitution. We can substitute the mapping V into e_2^L, creating $e_2^V = e_2^L\,[V(s)\,/\,s]$, the new sub-expression that we will use as a replacement for e_1''. Then we simply need to prove the equivalence $R,\ Y, e_1\,[e_2^V\,/\,e_1'] \equiv e_2$ for some state pair set R, and we can conclude $R,\ Y, e_1 \equiv e_2$ in the end.

For soundness, LemmaLeft requires two other *lemma productivity properties* to hold. First, we require the substitution to happen within a sub-expression e_1' of e_1 that is in *function application form*: simply put, e_1' must be a function application $f\ e_1^a \ldots e_k^a$, where f is a non-symbolic variable. Note that e_1' is allowed to be e_1 itself. Second, we require that f, the function being applied in e_1', is *not* syntactically included in e_2^V or in any functions invoked by e_2^V, either directly or indirectly. For an expression e, $\texttt{calls}(e, f, \{\})$ indicates whether e has any direct or indirect invocations of f:

$$\texttt{calls}(e, f, X) = \begin{cases} f = x \vee \texttt{calls}(\texttt{lookupVar}(x), f, \{x\} \cup X) & e = x, f \notin X \\ \texttt{calls}(e', f, X) & e = \lambda x\,.\,e' \\ \texttt{calls}(e_1, f, X) \vee \texttt{calls}(e_2, f, X) & e = e_1\ e_2 \\ \texttt{calls}(e', f, X) \vee & \\ \quad \exists (D\,\vec{x} \to e'') \in \vec{a} . \texttt{calls}(e'', f, X) & e = \texttt{case}\ e'\ \texttt{of}\ \{\vec{a}\} \\ \text{False} & \text{otherwise} \end{cases}$$

$$\textsc{LemmaLeftUS}\ \frac{\{\},\ Y^L, e_1^L \equiv e_2^L \qquad \exists e_1' \preccurlyeq e_1 . e_1' = f\ e_1^a\ \ldots\ e_k^a \quad \exists e_1'' \preccurlyeq e_1' . (e_1'',\ Y) \sqsubseteq_V (e_1^L,\ Y^L) \qquad e_2^V = e_2^L\,[V(s)\,/\,s] \quad R,\ Y, e_1\,[e_2^V\,/\,e_1''] \equiv e_2}{R,\ Y, e_1 \equiv e_2}$$

(a) Unsound proof rule.

$$f = [] \qquad g = g \qquad w\,h\,c = \texttt{case}\ h\ \texttt{of}\ \{[] \to c; \ldots\}$$

(b) Problematic expressions.

$$\textsc{LemmaLeftUS}\ \frac{\textsc{U-Coind}\ \dfrac{\vdots}{\{\},\ Y, wff \equiv wf\,(wff)} \quad \textsc{U-Coind}\ \dfrac{wf\,(wf\,f) \sqsubseteq_{\{\}} wf\,(wf\,f) \quad g \sqsubseteq_{\{\}} g}{R,\ Y, wf\,(wf\,f) \equiv g} \quad e_1' = wff \quad (wff,\ Y) \sqsubseteq_{\{\}} (wff,\ Y) \quad e_2^V = wf\,(wf\,f)}{R,\ Y, wff \equiv g}$$

$$\textsc{RAdd}\ \frac{\textsc{Red-L/R}\ \dfrac{\vdots}{R = \{(wf\,(wf\,f), g, Y)\},\ Y, wf\,(wf\,f) \equiv g}}{\{\},\ Y, wf\,(wf\,f) \equiv g}$$

(c) An incorrect proof tree.

Fig. 14: An unsound use of lemmas that causes coinduction to prove incorrectly that a terminating program and a non-terminating program are equivalent.

The function is an exhaustive traversal over e that unfolds non-symbolic variable definitions as it encounters them. In $\texttt{calls}(e, f, X)$, the third argument X serves to prevent the function from entering an infinite loop. The set should be empty in the initial call, and it records the variables whose definitions have been unfolded so far. If we ever see a variable that we have unfolded previously, we do not unfold it again. We can ignore repeated unfoldings because any AST nodes in a repeated unfolding are covered elsewhere in the traversal. Note that there is no need to take symbolic variable mappings into account because symbolic variables cannot be concretized as functions. For more details on our handling of symbolic functions, see Section 6.5.

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.

Example 4.1. Figure 14 shows an example of a situation where LemmaLeftUS, a flawed version of LemmaLeft that does not enforce the second lemma productivity property, leads to a faulty derivation of an untrue equivalence. Figure 14(a) shows the rule itself, which takes all of the premises that LemmaLeft takes except $\neg(e_2^V \preccurlyeq f)$. Figure 14(b) shows the definitions of three expressions that we can use together with LemmaLeftUS to violate

soundness. We can see from the definitions of f and w that $wf\,(wff)$ is not in SWHNF itself, but it reduces to wff and then to f, which is an empty list and therefore a terminal expression in SWHNF. On the other hand, the variable g is a non-terminating expression that will never reach SWHNF because it is defined in terms of itself.

Figure 14(c) shows the faulty derivation, where we reach the incorrect conclusion that $wf\,(wff)$ and g are equivalent. At the start, we can apply U-Coind to derive $R,\ Y, wf\,(wff) \equiv g$ because $R = \{(wf\,(wff), g, Y)\}$. After that, we apply LemmaLeftUS. Between the application of LemmaLeftUS and the final application of RAdd, there are two steps: one application of Red-L and one application of Red-R. The application of Red-L converts the previous rule's conclusion of $R,\ Y, wff \equiv g$ into $R,\ Y, wf\,(wff) \equiv g$ since, as we said before, $wf\,(wff)$ reduces to wff in one step. The application of Red-R leaves the conclusion $R,\ Y, wf\,(wff) \equiv g$ unchanged since g reduces to itself, but we need the step in the proof tree to uphold U-productivity for U-Coind. At the end, we apply RAdd to reach the conclusion that the terminating expression $wf\,(wff)$ is equivalent to the non-terminating expression g, which is a violation of soundness.

Why do our lemma productivity 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.

In Example 2.2, the lemma substitution that we use upholds both of our lemma productivity properties. The larger expression that receives the substitution is an application of `++`, and the new sub-expression that we insert is `drop n' xs'`. In the definition of `drop`, there are no function applications other than the recursive application of `drop` itself, so we meet the requirement of not introducing any new occurrences of `++`.

The rule LemmaRight resembles LemmaLeft but substitutes on the right-hand side of the state pair. Its soundness requirements are dual to the requirements for LemmaLeft.

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 $(R,\ Y, e_1 \equiv e_2)$ from the existence of some e_1^L, e_2^L, and Y^L such that $(\{},\ Y^L, e_1^L \equiv e_2^L)$, $(e_1,\ Y) \sqsubseteq_V (e_1^L,\ Y^L)$, and $(e_2,\ Y) \sqsubseteq_V (e_2^L,\ Y^L)$. The justification for the rule is straightforward. Since $(e_1,\ Y) \sqsubseteq_V (e_1^L,\ Y^L)$ and $(e_2,\ Y) \sqsubseteq_V (e_2^L,\ Y^L)$, it must be the case that $(e_1^L,\ Y^L)$ and $(e_2^L,\ Y^L)$ are *generalizations* of $(e_1,\ Y)$ and $(e_2,\ Y)$. That is, $(e_1^L,\ Y^L)$ and $(e_2^L,\ Y^L)$ must over-approximate the behavior of $(e_1,\ Y)$ and $(e_2,\ Y)$. Consequently, if $(e_1^L,\ Y^L)$ and $(e_2^L,\ Y^L)$ are equivalent, so are $(e_1,\ Y)$ and $(e_2,\ Y)$.

Lemma Application Locations Our lemma rules, as we present them here, can perform substitutions at any location within an expression. However, in practice, we usually do not need to consider all parts of an expression as candidates for lemma substitutions. In Section 6.4.1, we discuss the heuristics that we use for lemmas to restrict our attention

to the locations where lemma substitutions are most likely to be useful. We discuss our lemma heuristics further in Section 7.3.6, where we examine their effect on our evaluation results.

4.4 Soundness

We define the soundness of an equivalence checker as follows:

Definition 3 (Soundness)**.** *A set of proof rules is sound if a productive proof tree using those rules, and with the conclusion* $\{\}, Y, e_1 \equiv e_2$*, can be constructed only if* e_1 *and* e_2 *are equivalent.*

Soundness establishes that our proof rules for equivalence align with the semantic definition of equivalence from the beginning of the section. We formally state the soundness of the coinduction rules, in combination with the rules from the prior sections, as the following theorem:

Theorem 3 (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*), the coinduction rules (*RAdd*,* U-Coind*, and* G-Coind*), and the lemma rules (*LemmaLeft*,* LemmaRight*, and* LemmaOver*) are sound when used in a productive proof tree.*

Proof Consider a proof tree with a root of $\{\}, Y, e_1 \equiv e_2$. Soundness of the syntactic equality rule, the SWHNF equivalence rules, and the reduction rules is straightforward to prove, so we focus on the coinduction and lemma rules.

We consider branches beginning at the root of the proof tree and ending with U-Coind, G-Coind, or LemmaOver.

Ending with U-Coind Consider a branch ending at a leaf that is discharged with U-Coind:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\exists (e_1^P, e_2^P, Y^P) \in R, V\,.\ (e_1^C,\ Y^C) \sqsubseteq_V (e_1^P,\ Y^P) \wedge (e_2^C,\ Y^C) \sqsubseteq_V (e_2^P,\ Y^P)}\ \text{U-Coind}}{R \cup (S_1^C, S_2^C),\ Y^C, e_1^C \equiv e_2^C}}{\vdots}}{R \cup (e_1^P, e_2^P, Y^P),\ Y^P, e_1^P \equiv e_2^P}}{\neg \texttt{SWHNF}(e_1^P) \quad \neg \texttt{SWHNF}(e_2^P)}\ \text{RAdd}}{R,\ Y, e_1^P \equiv e_2^P}}{\vdots}}{\{\},\ Y, e_1 \equiv e_2}$$

Note that, since the proof tree is productive, there must be at least one application of each of Red-L and Red-R between RAdd and U-Coind.

Suppose that there is, in fact, some reduction of $(e_1^C,\ e_2^C,\ Y)$ that demonstrates that $e_1^C \not\equiv e_2^C$. Then, by the completeness of symbolic execution (Theorem 2), there exists some reduction of $(e_1^P,\ e_2^P,\ Y)$ that demonstrates that $e_1^P \not\equiv e_2^P$. Since $e_1^P \not\equiv e_2^P$, it must be that there exists a reduction of $(e_1^P,\ e_2^P,\ Y)$ where one of the expressions reaches SWHNF and the other never does, or where both expressions reach non-equivalent SWHNF expressions. We consider each case:

- *Only one expression terminates* Without loss of generality, suppose that e_1^P reaches a SWHNF expression e_1^F and that e_2^P does not terminate. Letting $p(S) = \texttt{SWHNF}(S)$ and $q(S) = \mathit{True}$, if no lemma is applied in the proof tree, Lemma 13 from the Appendix tells us that there exists a reduction sequence $S'_{\hookrightarrow} = (e_1^{1'},\ Y_1'), (e_2^{1'},\ Y_2'), \ldots$ which reduces e_1^P to some SWHNF expression such that

$$\forall i. e_{i-1}^{1'} \hookrightarrow e_i^{1'} \implies (e_j^{1'},\ Y_j') \not\sqsubseteq (e_i^{1'},\ Y_i'). \tag{4.1}$$

  If a lemma is applied by LEMMALEFT or LEMMARIGHT in a proof tree, it must be applied to some expression $f\ e \ldots e$ in function application form. Lemma 14 from the Appendix tells us that (4.1) is true even in the case of such a lemma application, along the branch of the proof tree corresponding to a minimal number of f applications. (Lemma 14 depends on Lemma 10, which requires the number of f applications to be minimal.) In either the case with lemmas or the case without lemmas, the relevant reduction sequence must correspond to some branch of the proof tree, and along it, we will never be able to apply U-COIND to discharge the state. Thus, we will not be able to form a finite proof tree, and our rules are sound in this case.
- *Both expressions terminate* Now suppose that e_1^P reduces to a SWHNF expression e_1^F, that e_2^P reduces to a SWHNF expression e_2^F, and that $e_1^F \not\equiv e_2^F$. The expressions e_1^F and e_2^F must be data constructor applications, lambdas, or bottoms. Similarly to the previous case, letting $p(S) = S \sqsubseteq e_1^F$ and $q(S) = S \sqsubseteq e_2^F$ allows us to use Corollary 3 to guarantee that there exists a paired reduction sequence $S'_{\hookrightarrow} = (e_1^1,\ e_1^2,\ Y_1'), (e_2^1,\ e_2^2,\ Y_2'), \ldots$ which reduces e_1^P and e_2^P to SWHNF expressions that approximate e_1^F and e_2^F, respectively, such that

$$\forall i. e_{i-1}^{1'} \hookrightarrow e_i^{1'} \implies (e_j^{1'},\ Y_j') \not\sqsubseteq (e_i^{1'},\ Y_i')$$

  and

$$\forall i. e_{i-1}^{2'} \hookrightarrow e_i^{2'} \implies (e_j^{2'},\ Y_j') \not\sqsubseteq (e_i^{2'},\ Y_i').$$

  Similarly to the case in which only one expression terminates, Lemma 14 tells us that this is true along some reduction sequence in the proof tree even if a lemma is applied with LEMMALEFT or LEMMARIGHT.
  We subdivide further to consider each of the three possible ways that the expressions could reach SWHNF:

  – e_1^F **and** e_2^F **are data constructor applications** If the data constructors being applied are different, the proof tree will not be able to be completed, and we will not be able to prove the equivalence of e_1^F and e_2^F soundly. If the data constructors are the same, DC-EQUIV must be applied to check the equivalence of each

corresponding argument between e_1^F and e_2^F. We can see then, by an inductive argument on the size of the proof tree, that the proof for one of the corresponding argument pairs must fail.

- **e_1^F and e_2^F are lambdas** We proceed with LAM-EQUIV, which checks the equivalence of both lambdas applied to the same fresh symbolic literal. Again, by an inductive argument on the size of the proof tree, the proof of the equivalence of these applications will fail.
- **e_1^F and e_2^F are labeled bottoms** If the labels are different, we will not be able to apply BOT-EQUIV to complete the proof tree.

Ending with G-COIND Now consider a proof tree with a root of $(\{\}, Y, e_1 \equiv e_2)$, with a branch that ends with G-COIND:

$$\dfrac{\dfrac{\dfrac{\vdots}{\dfrac{\dfrac{\exists (e_1^P, e_2^P, Y^P) \in R, V\,.\quad (e_1^C,\ Y^C) \sqsubseteq_V (e_1^P,\ Y^P) \quad \wedge (e_2^C,\ Y^C) \sqsubseteq_V (e_2^P,\ Y^P)}{R \cup (S_1^C, S_2^C),\ Y^C, e_1^C \equiv e_2^C}\ \text{G-COIND}}{}}{R \cup (e_1^P, e_2^P, Y^P),\ Y^P, e_1^P \equiv e_2^P}}{R,\ Y, e_1^P \equiv e_2^P}\ \text{RADD}$$

$$\dfrac{\vdots}{\{\},\ Y, e_1 \equiv e_2}$$

Note that, if there is at least one application each of RED-L and RED-R between the applications of RADD and G-COIND, we could have applied U-COIND instead, and soundness follows by the same argument. Thus, assume there is no application of RED-L (without loss of generality, we could assume instead that there is no application of RED-R) between RADD and G-COIND. To satisfy the productivity requirement, there must have been an application of either DC-EQUIV or LAM-EQUIV. This means that e_1^P is already in SWHNF.

Suppose that there is, in fact, some reduction of $(e_1^C,\ e_2^C,\ Y)$ that demonstrates that $e_1^C \not\equiv e_2^C$. Then, by Theorem 2, there must exist some reduction of $(e_1^P,\ e_2^P,\ Y)$ that demonstrates that $e_1^P \not\equiv e_2^P$. Since $e_1^P \not\equiv e_2^P$, it must be the case that there exists a reduction of e_2^P that does not reach SWHNF or that reaches an application of a constructor distinct from the constructor being applied in e_1^P. Soundness then follows from Lemma 13, as it does in the U-COIND case.

Ending with LEMMAOVER Now consider the case where we end a branch with the LEMMAOVER rule:

$$\dfrac{\dfrac{\{\},\ Y^L, e_1^L \equiv e_2^L \qquad (e_1,\ Y) \sqsubseteq_V (e_1^L,\ Y^L) \qquad (e_2,\ Y) \sqsubseteq_V (e_2^L,\ Y^L)}{R,\ Y, e_1 \equiv e_2}\ \text{LEMMAOVER}}{\vdots}$$

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

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

$$\textsc{CyL}\ \frac{\begin{array}{cc}\texttt{SWHNF}(e_2) & e \in \{e_1\} \cup \text{targets}(e_1) \\ (e,\ Y) \hookrightarrow^* (e_1',\ Y') & e' \in \{e_1'\} \cup \text{targets}(e_1') \quad (e,\ Y) \sqsubseteq (e',\ Y')\end{array}}{R,\ Y, e_1\ \not\equiv\ e_2}$$

$$\textsc{CyR}\ \frac{\begin{array}{cc}\texttt{SWHNF}(e_1) & e \in \{e_2\} \cup \text{targets}(e_2) \\ (e,\ Y) \hookrightarrow^* (e_2',\ Y') & e' \in \{e_2'\} \cup \text{targets}(e_2') \quad (e,\ Y) \sqsubseteq (e',\ Y')\end{array}}{R,\ Y, e_1\ \not\equiv\ e_2}$$

Fig. 15: Counterexample rules.

To use LemmaOver, we must prove that $S_1^l \equiv S_2^l$. Then, we can discharge $S_1 \equiv S_2^l$ if there exists some V such that $(e_1,\ Y) \sqsubseteq_V S_1^l$ and $(e_2,\ Y) \sqsubseteq_V S_2^l$.

■

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 15, 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 15. The cycle detection rules check if one expression has a non-terminating path, while the other 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 $(e,\ Y) \hookrightarrow^* (e_1',\ Y')$, where e is either e_1 itself or a *target expression* within e_1. There also needs to exist some e' such that $(e,\ Y) \sqsubseteq (e',\ Y')$, where e' is either e_1' itself or a target expression within e_1'. For any expression $\hat{e}$, the target expressions of $\hat{e}$ are a non-exhaustive set of sub-expressions of $\hat{e}$ that are guaranteed to undergo evaluation before $\hat{e}$ reaches SWHNF. We use a helper function to define the set of target expressions:

$$\text{targets}(e) = \begin{cases} \{e'\} \cup \text{targets}(e') & e = \texttt{case}\ e'\ \texttt{of}\ \{\vec{a}\} \\ \{e_1\} \cup \text{targets}(e_1) & e = e_1\ e_2 \\ \{\} & \text{otherwise} \end{cases}$$

If the premises of CYL are satisfied and e' is e'_1 itself, then, by Theorem 2, there is an infinite reduction sequence beginning with $(e,\ Y)$. Intuitively, the premises $(e,\ Y) \hookrightarrow^* (e'_1,\ Y')$ and $(e,\ Y) \sqsubseteq (e'_1,\ Y')$ mean that $(e,\ Y)$ can evaluate to a state that is at least as general as itself. Since $(e'_1,\ Y')$ is at least as general as $(e,\ Y)$, $(e'_1,\ Y')$ must have an execution path corresponding to any execution path that $(e,\ Y)$ has. From $(e'_1,\ Y')$, we can follow the path corresponding to $(e,\ Y) \hookrightarrow^* (e'_1,\ Y')$ 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.

If e' is a target expression of e'_1 instead, then we can still construct an infinite reduction sequence for $(e,\ Y)$, but the construction is less direct. As in the previous case, we have that $(e,\ Y) \hookrightarrow^* (e'_1,\ Y')$. Because e' is a target expression of e'_1, any execution path that $(e'_1,\ Y')$ has must reduce e' to SWHNF before reducing anything else. Since $(e,\ Y) \sqsubseteq (e',\ Y')$, we can follow the execution path for e' corresponding to $(e,\ Y) \hookrightarrow^* (e'_1,\ Y')$, and this will give us another more deeply nested target expression that approximates $(e,\ Y)$ when paired with the current state of its expression. We can keep repeating this process indefinitely to produce an infinite execution path.

All that remains to be done is to link our infinite reduction sequence for e to an infinite reduction sequence for e_1. If e is e_1, no conversion is necessary. If e is not e_1, we can convert the infinite execution path for e into an infinite execution path for e_1 in the same way that we converted the infinite path for e' into an infinite path for e'_1 before. Because e is a target expression within e_1, any execution path that $(e_1,\ Y)$ has must reduce e to SWHNF before reducing anything else, and the approach from before works again. Since an infinite reduction sequence for e_1 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.

Example 5.1. In Example 2.3 from Section 2, we showed a cycle counterexample where $e = e_1$ and $e' = e'_1$ for the application of CYL. For a different example that uses target expressions, consider Theorem 4 from the Zeno suite (Sonnex *et al.*, 2012). The theorem appears in Figure 16. The right-hand side is in SWHNF, but the left-hand side has an infinite execution path in the event that `n` is infinite. Evaluation starts by reducing the outermost case statement:

```
case n === n of
  False -> count n xs
  True -> S (count n xs)
```

If we concretize `n` as `S n'`, the expression reduces further to this:

```
case n' === n' of
  False -> count (S n') xs
  True -> S (count (S n') xs)
```

```
count :: Nat -> [Nat] -> Nat
count x [] = Z
count x (y:ys) =
  case x === y of
    False -> count x ys
    True -> S (count x ys)

forall n xs . count n (n : xs) = S (count n xs)
```

Fig. 16: The `count` function and Zeno Theorem 4.

At this point, we can apply CYL. We do not have an approximation relation between the two whole expressions because the new expression has `n'` in some places where the old expression had `n` and has `S n'` in others. However, `n' === n'` is at least as general as `n === n`. The sub-expression `n === n` is a target expression of the old expression, and `n === n` reduces to `n' === n'` along the path we are considering, so we can use the target expressions for CYL.

Note that this application of CYL is valid only because `n === n` reduces directly to `n' === n'`. If there were more evaluation steps between the old expression and the new expression that altered the outer case statement and then brought it back to a form resembling the old expression, we would not be able to apply CYL. In the premises for CYL, we enforce this requirement by phrasing the reduction premise in terms of the target expression being considered, not the outermost expression.

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 9 is not computable. To implement the equivalence checking algorithm, we use a simpler approximation relation $\subseteq$, defined in Figure 17, that implies the theoretical version of approximation. The relation $\sqsubseteq$ 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.

Within our algorithm, 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 17 compute a mapping V such that $S_1 \subseteq_V S_2$ (alternatively, $S_1 \sqsubseteq_V S_2$) holds. 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 $\lhd$-SYMV1 and $\lhd$-SYMV2. The rule $\lhd$-SYMV1 applies when e_2 is

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

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

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

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

$$\lhd\text{-DC}\ \frac{}{V \vdash D \lhd_{V,Y_1,Y_2} D} \qquad \lhd\text{-App}\ \frac{V' \vdash e_1 \lhd_{V,Y_1,Y_2} e_1' \quad V'' \vdash e_2 \lhd_{V',Y_1,Y_2} e_2'}{V'' \vdash e_1\ e_2 \lhd_{V,Y_1,Y_2} e_1'\ e_2'}$$

$$\lhd\text{-Bt}\ \frac{}{V \vdash \bot^L \lhd_{V,Y_1,Y_2} \bot^L} \qquad \subseteq\text{-Link}\ \frac{V \vdash e_1 \lhd_{\{\},Y_1,Y_2} e_2}{(e_1,\ Y_1) \subseteq_V (e_2,\ Y_2)}$$

Fig. 17: Computable approximation.

a symbolic variable not mapped by the current V, and it adds e_1 as the mapping for e_2, producing the conclusion $V \cup \{s \to e\} \vdash e_1 \lhd_{V,Y_1,Y_2} s$. 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 $e_1 = \texttt{lookup}(s, V)$.

As we state in Section 3 and demonstrate with Example 3.2, the use of evaluation in the definition of $\sqsubseteq$ is essential to establish Theorem 2, 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 4. *If $S_1 \subseteq S_2$, then $S_1 \sqsubseteq S_2$.*

Proof We need to prove that, for any states $S_1 = (e_1, Y_1)$ and $S_2 = (e_2, Y_2)$ such that $e_1 \subseteq^E_{V,Y_1,Y_2} e_2$ for some mapping V, there exists some mapping V' such that $(e_1,\ Y_1) \sqsubseteq_{V'} (e_2,\ Y_2)$. We will show this by case analysis and induction on the definition of computable approximation.

For this proof, we can treat the relations $(e_1,\ Y_1) \sqsubseteq_{V'} (e_2,\ Y_2)$ and $V' \vdash e_1 \lhd_{V,Y_1,Y_2} e_2$ as interchangeable for any combination of expressions, symbolic stores, and variable mappings. The former has one symbolic variable mapping, but the latter has two. In the rules for computable approximation, we use the auxiliary mapping V to construct the main mapping V' gradually while traversing a pair of expressions. The rule $\lhd$-SymV1 adds the mappings

that we need as we encounter them, and the requirement that $s \notin V$ for that rule prevents us from adding inconsistent mappings. We can always convert the auxiliary relation into $\subseteq$ if we start the traversal with an empty auxiliary mapping.

If we ignore the distinction between the two relations for computable approximation, then the proof is trivial for most rules because there is an exact analogue among the rules for non-computable approximation. The main difficulty comes from the three rules for handling symbolic variables in computable approximation that do not have exact analogues. (The rule $\lhd$-SymLkL corresponds to $\sqsubseteq$-Sym0.)

The rule $\lhd$-SymV1 is subsumed by $\sqsubseteq$-Sym2. If we have $s \notin Y_2$, $s \notin V$, and $V \cup \{s \to e\} \vdash e \lhd_{V,Y_1,Y_2} s$, then let $V'' = V \cup \{s \to e\}$. By definition, $e = \texttt{lookup}(s, V'')$. It holds trivially that $(e,\ Y_1) \hookrightarrow^* (e,\ Y_1)$, and we also know that $(e,\ Y_1) \sqsubseteq_{V''} (e,\ Y_2)$ since any expression approximates itself. Together with our premise $s \notin Y_2$, we have everything we need to apply $\sqsubseteq$-Sym2 to conclude $(e,\ Y_1) \sqsubseteq_{V''} (s,\ Y_2)$.

The rule $\lhd$-SymV2 is also subsumed by $\sqsubseteq$-Sym2. If we have $s \notin Y_2$, $e = \texttt{lookup}(s, V)$, and $V \vdash e \lhd_{V,Y_1,Y_2} s$, then we have $(e,\ Y_1) \hookrightarrow^* (e,\ Y_1)$ and $(e,\ Y_1) \sqsubseteq_V (e,\ Y_2)$ trivially. Again, this gives us what we need to apply $\sqsubseteq$-Sym2 to reach the conclusion $(e,\ Y_1) \sqsubseteq_V (s,\ Y_2)$.

The most complex case of the proof is the one for $\lhd$-SymLkR, and we will dedicate the remainder of the proof to it. The rule $\lhd$-SymLkR derives $V' \vdash e_1 \lhd_{V,Y_1,Y_2} s$ from the premises $\exists e = \texttt{lookup}(s, Y_2)$ and $V' \vdash e_1 \lhd_{V,Y_1,Y_2} e$. We will break our proof for this rule into two sub-cases.

Concretized Symbolic Variables on Both Sides Suppose that e_1 is a symbolic variable s'. We know from our premises that s has a concretization e in Y_2. Moreover, e is not a symbolic variable because a symbolic variable cannot be concretized as another symbolic variable. We also have as a premise that $V' \vdash s' \lhd_{V,Y_1,Y_2} e$. To derive $V' \vdash s' \lhd_{V,Y_1,Y_2} e$ in the first place, we would have needed to use the rule $\lhd$-SymLkL. No other rule's conclusion has a symbolic variable on the left-hand side and something that is not a symbolic variable on the right-hand side. That application of $\lhd$-SymLkL would have needed to have $e_1' = \texttt{lookup}(s', Y_1)$ and $V' \vdash e_1' \lhd_{V,Y_1,Y_2} e$ as premises. By our inductive hypothesis, this implies that there exists some mapping V'' such that $(e_1',\ Y_1) \sqsubseteq_{V''} (s,\ Y_2)$. We said that Y_1 maps s' to e_1', so we can apply $\sqsubseteq$-Sym0 to conclude $(s',\ Y_1) \sqsubseteq_{V''} (s,\ Y_2)$, which was our goal for this sub-case.

Concretized Symbolic Variable on the Right Only Now assume that e_1 is not a symbolic variable. For this sub-case, we still have the same premises and conclusion: $V' \vdash e_1 \lhd_{V,Y_1,Y_2} s$, $\exists e = \texttt{lookup}(s, Y_2)$, and $V' \vdash e_1 \lhd_{V,Y_1,Y_2} e$. The rule $\sqsubseteq$-Sym1 gives us that $(e_1,\ Y_1) \sqsubseteq_{V''} (s,\ Y_2)$ if three conditions hold, where V'' is a new mapping, $e' = \texttt{lookup}(s, V'')$, and e'' is some other expression:

1. $(e',\ Y_1) \hookrightarrow^* (e'',\ Y_1)$.
2. $(e_1,\ Y_1) \sqsubseteq_{V''} (e'',\ Y_2)$.
3. $(e_1,\ Y_1) \sqsubseteq_{V''} (e,\ Y_2)$.

Let e' and e'' both be equal to e. These definitions make condition (1) hold trivially because $e' = e''$ and $\hookrightarrow^*$ is reflexive. Before we define V'' and confirm the other two

conditions, we will perform some more case analysis on e. Because e is drawn from a symbolic store, it must be a data constructor application or labeled bottom.

If $e = \bot^L$ for some label L, then we know that e_1 is $\bot^L$ as well since $V' \vdash e_1 \lhd_{V,Y_1,Y_2} e$. We assumed that e_1 is not a symbolic variable, so there is no other way for the relation to hold. The rule $\sqsubseteq$-BT gives us that $(\bot^L,\ Y_1) \sqsubseteq_{V''} (\bot^L,\ Y_2)$ regardless of the value of V''. This gives us conditions (2) and (3) immediately because e_1, e'', and e are all equal to $\bot^L$ in this situation. Let V'' be the mapping that only maps s to e' in order to uphold the requirement that $e' = \texttt{lookup}(s, V'')$.

If $e = D\,\vec{e^d}$, where $\vec{e^d}$ is a vector of n arguments for D, then e_1 must be $D\,\vec{e^c}$ for some other vector $\vec{e^c}$ of the same length n. Since $V' \vdash e_1 \lhd_{V,Y_1,Y_2} e$ and we can ignore the distinction between the two relations for computable approximation, it must hold that $(e_1,\ Y_1) \subseteq_{V'} (e,\ Y_2)$ and that $(e_i^c,\ Y_1) \subseteq_{V'} (e_i^d,\ Y_2)$ for every $i \in \{1, ..., n\}$. As an inductive hypothesis, we can assume that, for every such i, there is a mapping V_i such that $(e_i^c,\ Y_1) \sqsubseteq_{V_i} (e_i^d,\ Y_2)$. For every index i, the set of symbolic variables in e_i^d must be disjoint from the set of symbolic variables in any other argument in $\vec{e^d}$ because $D\,\vec{e^d}$ is a concretization of s. We also know that s does not appear in $\vec{e^d}$ because symbolic variable concretizations cannot be cyclic. This means that we can define V'' as $\bigcup_{i=0}^{n} V_i$, where V_0 is the mapping that simply maps s to e', without worrying about overlapping mappings. (Assume that $V_1, ..., V_n$ only contain mappings that are actually used for their respective approximations. We include V_0 in the union in order to uphold the requirement that $e' = \texttt{lookup}(s, V'')$.) Adding irrelevant symbolic variable mappings does not interfere with an approximation, so we know that $(e_i^c,\ Y_1) \sqsubseteq_{V''} (e_i^d,\ Y_2)$ for every $i \in \{1, ..., n\}$. It follows from $\sqsubseteq$-DC and $\sqsubseteq$-APP that $(D\,\vec{e^c},\ Y_1) \sqsubseteq_{V''} (D\,\vec{e^d},\ Y_2)$. In other words, $(e_1,\ Y_1) \sqsubseteq_{V''} (e,\ Y_2)$. This is precisely what we wanted to confirm for condition (3), and it gives us condition (2) as well because $e'' = e$.

■

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 10 is applied. Blocks allow us to enforce the productivity

$\overline{H} \leftarrow \{[((e_1, \{\}); (e_2, \{\}))]\}$;
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.

properties for both guarded and unguarded coinduction. The verification algorithm deals mainly with obligations rather than dealing with state pairs directly 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.

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 18. 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,

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

Fig. 18: Symbolic store synchronization.

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_j^1 and S_k^2 are the final states in a block B_i, the expressions inside S_j^1 and S_k^2 must be either data constructor applications or lambdas. If the expressions are data constructor applications, then the expressions in the starting states S_{j+1}^1 and S_{k+1}^2 in B_{i+1} are corresponding arguments from the applications. If S_j^1 and S_k^2 are lambdas, then the expressions in S_{j+1}^1 and S_{k+1}^2 are applications of those lambdas 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 e is either a fully-applied non-symbolic function or a case statement such that an element of targets(e) is a fully-applied non-symbolic function, reusing the definition of targets(e) from Section 5. 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 $[\ldots, (\ldots, S_j^1; \ldots, S_k^2)]$, we apply reduction to S_j^1 (assuming S_j^1 is not in SWHNF already) to obtain a new set of states $\overline{S_{j+1}^1}$ such that $\forall S_{j+1}^1 \in \overline{S_{j+1}^1}. S_j^1 \hookrightarrow^*_{Y_k^2} S_{j+1}^1$. Then, for each $S_{j+1}^1 = (e_{j+1}^1,\ Y_{j+1}^1) \in \overline{S_{j+1}^1}$, S_{k+1}^2 is reduced using $\hookrightarrow^*_{Y_{j+1}^1}$ to obtain a set of states $\overline{S_{k+1}^2}$, which gives us new obligations

$$\{[\ldots, (\ldots, S_j^1, S_{j+1}^1; \ldots, S_k^2, S_{k+1}^2)] | S_{k+1}^2 \in \overline{S_{k+1}^2}\}.$$

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. Example 2.1 is an instance of this.

6.3.2 Implementations of 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:

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

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 12 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 12) 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 states that we add to R need to have indices $1 \leq p_1 \leq b$ and $1 \leq p_2 \leq k$, and we use the rule G-COIND (also from Figure 12) 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 to $\subseteq$ and coinduction. Consequently, we generate potential lemmas in situations where $\subseteq$ fails to hold. If we have two states such that $(e_1,\ Y_1) \not\subseteq (e_2,\ Y_2)$, 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 19 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, $\not\vdash e_1 \lhd_{V,Y_1,Y_2} e_2$. We use these expressions to create a new possible lemma:

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

$$\textsc{LemCo}\ \frac{Y' = Y_1 \cup Y_2}{(e_1,\ Y') \equiv (e_2\,[V(s)\,/\,s],\ Y') \not\vdash e_1 \lhd_{V,Y_1,Y_2} e_2}$$

$$\textsc{LemGen}\ \frac{\begin{array}{c} e_1' \in \text{targets}(e_1) \qquad e_2' \in \text{targets}(e_2) \\ e_1' = e_2' \quad s\ \texttt{fresh} \quad Y' = Y_1 \cup Y_2 \end{array}}{(e_1\,[s\,/\,e_1'],\ Y') \equiv (e_2\,[s\,/\,e_2'],\ Y') \not\vdash e_1 \lhd_{V,Y_1,Y_2} e_2}$$

Fig. 19: Rules for lemma introduction.

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 13 can replace e_1 with $e_2\,[V(s)\,/\,s]$. We can see that $e_2\,[V(s)\,/\,s] \lhd_{V,Y_1,Y_2} e_2$, 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 $e_2\,[V(s)\,/\,s]$ being added by the substitution, or in any functions directly or indirectly callable by $e_2\,[V(s)\,/\,s]$. 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 examine the set of target expressions from either side, reusing our helper function from Section 5. If an expression in targets(e_1) is syntactically equal to an expression in targets(e_2), then we create a potential lemma where the matching target expressions in e_1 and e_2 are replaced with the same fresh symbolic variable. The rule LemGen in Figure 19 formalizes this. If we prove the lemma, we can use it to discharge the original obligation by applying the LemmaOver rule from Figure 13.

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 $\subseteq$. Also, L and L' count as equivalent if the approximation works in both directions.

6.4.1 Heuristics for potential lemma generation

In our implementation, we use a heuristic to limit the generation of possible lemmas. If $(e_1,\ Y_1)$ and $(e_2,\ Y_2)$ are two states being tested for approximation, and e'_1 and e'_2 are corresponding sub-expressions of e_1 and e_2 such that $\vdash e'_1 \lhd_{V,Y_1,Y_2} e'_2$ fails to hold, then we use LemCo to produce a possible lemma from e'_1 and e'_2 only if e'_1 and e'_2 do not lie in the *active region* of e_1 and e_2. We define a function active(e) to indicate the parts of an expression that lie in the active region:

$$\text{active}(e) = \begin{cases} \{e\} \cup \text{active}(e_1) & e = e_1\ e_2 \\ \{e\} \cup \text{active}(e') & e = \texttt{case}\ e'\ \texttt{of}\ \{\vec{a}\} \\ \{e\} \cup \text{active}(e') & e = \lambda x\,.\,e' \\ \{e\} & \text{otherwise} \end{cases}$$

Intuitively, active(e) is a set of AST nodes within e that are guaranteed to undergo evaluation if symbolic execution continues indefinitely without reaching a terminal expression. The set is not necessarily the whole set of AST nodes that are guaranteed to undergo evaluation, since determining reachability is undecidable in general. However, the function captures some guarantees that hold universally for our language. Importantly, the output of active(e) should be viewed as a set of AST locations, not a set of expressions. If the same sub-expression e' appears multiple times in an expression e, both in the active region and outside it, the copies of e' that are not in active(e) can be used to generate possible lemmas.

Our definition of the active region is very similar to our definition of target expressions from Section 5, but active(e) includes the bodies of lambdas and targets(e) does not. The reason for the difference is that the definition of the active region depends on the assumption that e has an infinite reduction sequence. The definition of target expressions does not incorporate that assumption.

For an application $e_1\ e_2$, the active region includes e_1 but not e_2. Recall the reduction rules in Figure 7. An infinite reduction sequence is not guaranteed to evaluate e_2: the sequence could apply App repeatedly, constantly reducing e_1 while never touching e_2. However, an infinite reduction sequence must manipulate e_1, either by applying App or by applying Appλ to substitute e_2 into e_1. The rule BtApp effectively leaves e_1 untouched, but it cannot appear in an infinite reduction sequence because the end result of BtApp is a terminal expression.

For case statements, we include the scrutinee in the active region but not the expressions in the branches. The rationale is the same as it is for applications. An infinite reduction

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

Fig. 20: Evaluation for symbolic functions.

sequence can reduce the scrutinee forever with CsEv and never reach any of the branches, but there is no way for it to avoid reducing the scrutinee. We cannot have an infinite reduction sequence where the scrutinee reduces to a bottom value because the evaluation would terminate one step afterward with BtCs.

Even though lambdas are in SWHNF, we count the body of a lambda as being inside the active region if the lambda itself is in the active region. When a lambda is applied to an argument, the body receives some substitutions and becomes the main expression to be evaluated afterward. If the lambda itself is in the active region, then an infinite reduction sequence will need to reduce the lambda eventually by applying it, and the body of the lambda will need to undergo evaluation at that point.

Why do we choose not to generate potential lemmas from sub-expressions in the active region? It is not a requirement for soundness, but it helps to prevent NEBULA from cluttering its search space with unnecessary potential lemmas. Recall the proof of `prop01` from Example 2.2. We needed to apply a lemma to convert `drop (S n') (x:xs')` into `drop n' xs'` because the application of `drop` is not in the active region of `take n' xs' ++ drop (S n') (x:xs')`. On the execution path where `n'` and `xs'` are concretized with more successors and list elements endlessly, the application of `drop` will never be reduced, so normal evaluation will never cause it to align with the original application `drop n xs`. On the other hand, there is no need to apply a lemma to the application of `take`: the application lies in the active region, so ordinary symbolic execution reduces the application of `take` on its own to give us the alignment we want. In general, the same principle establishes that lemma substitutions in the active region are not as useful for NEBULA as lemma substitutions outside the active region are. There are exceptions, as we discuss in Section 7.3.6, but in practice the performance improvement that comes from narrowing the space of potential lemmas outweighs the benefit of generating potential lemmas from sub-expressions in the active region.

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 20. 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 the sake of simplicity, we check only for syntactic equality between symbolic function applications rather than performing a more thorough equivalence check.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs) =
  case p x of
    True -> x : (takeWhile p xs)
    _ -> []

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:xs) =
  case p x of
    True -> dropWhile p xs
    _ -> x:xs

forall p xs . takeWhile p xs ++ dropWhile p xs = xs
```

Fig. 21: Zeno Theorem 43.

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.

6.5.1 Symbolic function consistency

The possibility of spurious counterexamples has repercussions for our handling of one-sided cycle detection. An approximation for one-sided cycle detection does not represent a real counterexample if it maps expressions with differently concretized symbolic function mappings to each other. NEBULA can encounter spurious counterexamples with this problem if we do not impose any safeguards for one-sided cycle detection:

Example 6.2. Consider Theorem 43 from the Zeno test suite (Sonnex *et al*., 2012), shown in Figure 21. In the theorem, `p` is a symbolic function whose output type is `Bool`. The function application `takeWhile p xs` returns the entries at the front of `xs` up to but not including the first entry for which `p` returns `False`. Conversely, `dropWhile p xs` returns all the entries of `xs` except the ones returned by `takeWhile p xs`. Consider the symbolic execution branch where, at the start, `xs` is concretized as `x:ys`. We examine the left-hand side's evaluation:

```
takeWhile p xs ++ dropWhile p xs
takeWhile p (x:ys) ++ dropWhile p (x:ys)
```

Within `takeWhile`, `p` is applied to `x`, so we map `p x` to a fresh symbolic variable `b`. The case statement inside `takeWhile` will then branch on the value of `b`. On the branch where `b` is concretized as `True`, we have this:

```
(x:takeWhile p ys) ++ dropWhile p (x:ys)
x:(takeWhile p ys ++ dropWhile p (x:ys))
```

At this point, the leading `x` drops from both sides. (The right-hand side is simply `x:ys`.) We evaluate further, taking the branch where `ys` is concretized as `y:zs`:

```
takeWhile p ys ++ dropWhile p (x:ys)
takeWhile p (y:zs) ++ dropWhile p (x:y:zs)
```

We apply `p` to `y`, and we map `p y` to a fresh symbolic variable `b'`. Now `takeWhile` branches on the value of `b'`. Consider the branch where `b'` is concretized as `False`, causing `takeWhile` to return `[]`. This makes the application of `dropWhile` reachable:

```
[] ++ dropWhile p (x:y:zs)
dropWhile p (x:y:zs)
dropWhile p (y:zs)
```

Because `p x` evaluates to `True`, `dropWhile` drops `x` from the front of the list. Once evaluation reaches this point, CYL appears to be applicable. If we ignore symbolic function mappings, then `dropWhile p (x:y:zs)` approximates `dropWhile p (y:zs)`, with `y` mapping to `x` and `zs` mapping to `y:zs`. The right-hand side is `y:zs` at this point, which is in SWHNF, so we have what looks like a one-sided cycle. However, this is not a genuine counterexample because `x` is not really a more specific version of `y` in all aspects of its behavior. During evaluation on this branch, we decided that `p x` is `True` and that `p y` is `False`. The concrete values `True` and `False` are distinct, so replacing `y` with `x` does not preserve the program's non-symbolic behavior perfectly.

We can avoid spurious cyclic counterexamples with this problem by enforcing an extra *symbolic function consistency* requirement on any approximation mapping V for a one-sided cycle. Let Y_1 be the symbolic store for the present state S_1, and let Y_2 be the symbolic store for the past state S_2. Let e_1 be a symbolic function application mapped to the variable s_1 in Y_1. Let e_2 be another symbolic function application mapped to s_2 in Y_2. Let V be a mapping for an approximation between S_1 and S_2. If any expressions with symbolic function mappings from the symbolic store in the past align with expressions that have symbolic function mappings in the symbolic store in the present, then the mappings for the two expressions also need to align:

$$\forall (e_1, s_1) \in Y_1, (e_2, s_2) \in Y_2 . (e_1,\ Y_1) \subseteq_V (e_2,\ Y_2) \Rightarrow (s_1,\ Y_1) \subseteq_V (s_2,\ Y_2)$$

Importantly, the mapping V for symbolic function consistency is fixed to be the same mapping from the main approximation. If an approximation that holds between two symbolic function applications requires a different set of mappings than the main approximation does, then we do not need to worry about it.

Note that our enforcement of symbolic function consistency is only a helpful heuristic for rejecting certain spurious counterexamples, not a requirement for the soundness of verification. There is no need to check symbolic function consistency when applying regular tactics for verification. To see why, let e_1 and e_2 be two symbolic function applications such that $(e_1,\ Y_1) \sqsubseteq (e_2,\ Y_2)$ for some symbolic stores Y_1 and Y_2, but e_1 and e_2 are not syntactically equal. Let s_1 and s_2 be the fresh symbolic variables used for e_1 and e_2, respectively. Suppose that, as the verification algorithm runs, we produce and discharge an obligation H where s_1 and s_2 have inconsistent concretizations. Discharging H does not make the verification algorithm unsound: the concretizations for s_1 and s_2 represent an impossible situation, so the equivalence for H holds vacuously. Additionally, in order to verify the theorem fully, we will still need to cover all of the cases where s_1 and s_2 have consistent concretizations. Taking impossible concretizations into consideration may prevent NEBULA from verifying certain theorems, but it does not allow NEBULA to disregard any symbolic execution paths that really need to be validated.

6.6 Total variables

Our implementation allows 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 we under-approximate the requirements for totality:

$$\text{totalExpr}(e) = \begin{cases} s \text{ marked as total} & e = s \\ \text{True} & e = D \\ \text{totalExpr}(e_1) \wedge \text{totalExpr}(e_2) & e = e_1\ e_2 \\ \text{False} & \text{otherwise} \end{cases}$$

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. When a total symbolic variable is concretized, any new symbolic variables in the concretization are marked as total in turn.

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 f to arguments $e_1^a \ldots e_k^a$ that are all total according to $\text{totalExpr}(e_i^a)$, we mark the fresh variable that we use as a substitute for the application as total.

Our definition of totality, as used to describe both functions and non-function expressions, is standard in the field of functional programming (Danielsson *et al.*, 2006; Claessen *et al.*, 2012; Breitner *et al.*, 2018; Spector-Zabusky *et al.*, 2018; Hallahan *et al.*, 2019; Mastorou *et al.*, 2022). This meaning is closely linked to the mathematical concept of a *total function*. In mathematics, a total function has a mapping for every element in its

domain, or set of allowed input values. In functional programming, when deciding whether a function is total, we consider a non-function expression to be part of the domain only if it contains no bottoms or diverging sub-expressions. Correspondingly, we consider a function expression to be part of the domain only if that function expression is itself total. Note that a total function may still produce $\perp$ or diverge when given arguments (including function arguments) that themselves contain $\perp$ or diverge because such expressions are not part of the considered domain for totality. Further, in these situations, the function itself is not the source of the bottom or non-termination: it simply retransmits the bottom or non-termination that comes from some other part of the program.

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 equivalences between Haskell expressions automatically. NEBULA is open source. It is available as part of the G2 symbolic execution engine at https://github.com/BillHallahan/G2 and at https://zenodo.org/records/7083308. 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.

7.1 Benchmark construction

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.

7.1.1 Totality and finiteness requirements

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 Example 2.3 in Section 2 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
```

```
walkList [] a = a
walkList (_:xs) a = walkList xs a

walkNatList xs a = case xs of
  [] -> a
  y:ys -> walkNat y (walkNatList ys a)
```

Fig. 22: The `walkList` and `walkNatList` functions.

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 other two walk functions appear in Figure 22. 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.

If a theorem requires multiple variables to be finite, we need to have multiple nested walk function applications. Whenever multiple variables need walk function applications for a single theorem, the order that we use for the walk function application nesting is the same as the order that the original theorem's arguments follow. On both sides, the walk function applications for earlier arguments appear outside the walk function applications for later arguments. We impose our walk-function ordering requirement for the sake of simplicity. Allowing for more variation in the order of walk function applications would cause the number of options for minimal finiteness requirements to grow significantly without any evident benefit for demonstrating the capabilities of NEBULA. Furthermore, if we allowed different walk-function application orders between the two sides of a theorem, simple counterexamples would be possible for any combination with differing orders between the two sides. Let `a` and `b` be two symbolic variables of type `Nat` and consider the expressions `walkNat a (walkNat b Z)` and `walkNat b (walkNat a Z)`. If we define `a` as $\bot^L$ and `b` as `S b`, then `walkNat a (walkNat b Z)` evaluates to $\bot^L$ and `walkNat b (walkNat a Z)` fails to terminate. We can circumvent the problem by requiring `a` and `b` to be total, but we still do not derive any clear benefit from permitting variation in walk-function application orders.

For some Zeno theorems, there are two distinct minimal combinations of restrictions on finiteness and totality that make the theorem true. In situations where multiple minimal combinations of requirements exist, we treat the versions of the theorem with the two combinations of requirements as if they were distinct theorems. No theorem in the Zeno suite has more than two minimal alternatives that are expressible in our system of requirements for totality and finiteness.

Table 1: Evaluation results.

Category	# Thms	# V	# C	# TO	V Time	C Time
Unmodified theorems	85	24	61	0	7.6	5.1
Modified (no finite variables)	17	13	0	4	6.7	N/A
Modified (finite variables)	58	14	0	44	7.7	N/A
Cycle counterexamples	45	0	44	1	N/A	5.7

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.
V Time is the average time that NEBULA takes to verify the theorems that it proved in a category, in seconds.
C Time is the average time that NEBULA takes to find a counterexample for the theorems in a category that it rejected, also in seconds.

7.2 Results

For the evaluation, we ran NEBULA on a Linux desktop with a 3.7 GHz Intel Xeon W-2145 processor, and we gave each theorem a time limit of 3 minutes. To compile NEBULA, we used version 8.10.7 of GHC, which is the same version that we used for the original evaluation. 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 all 24.

As an additional assessment of question (1), we also run NEBULA on the theorems modified with totality requirements and finiteness requirements. There are 16 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 17 for this category. With the minimum totality requirements in place, NEBULA proves 13 of the theorems (76%) and hits the time limit on the remaining 4. There are also 45 theorems that are only true when certain variables are required to be finite. 13 of the 45 theorems have two distinct combinations of minimal totality and finiteness requirements, so we effectively have 58 theorems in this category. NEBULA verifies 14 of the theorems (24%) 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 45 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 44 of the 45 theorems (98%) and hits the time limit for only one.

7.3 Discussion of results

7.3.1 Differences from original evaluation

Overall, the results of our evaluation are better than the results in our original paper on NEBULA. For the original suite of unmodified benchmarks, we had 22 verifications, 61 counterexamples, and 2 timeouts. For the original suite of modified benchmarks with no finite variables, we had 11 verifications and 7 timeouts. For the original suite of modified benchmarks with finite variables, we had 12 verifications and 44 timeouts. For the original cycle counterexample detection suite, we had 32 counterexamples and 12 timeouts. Overall, we added 6 new verifications and 12 new cycle counterexamples. Every theorem that was verified in the original evaluation is still verified, and every theorem that was rejected with a genuine counterexample in the original evaluation is still rejected with a genuine counterexample.

In the original publication, one of the benchmarks was misclassified, namely theorem 58. It was classified as requiring total variables but no finite variables, but it actually requires finiteness restrictions to be valid in a non-strict setting, and there are two distinct ways to make it valid with finiteness requirements. Because of this, our new evaluation has one fewer theorem for the modified benchmarks with no finite variables, two new theorems for the modified benchmarks with finite variables, and one new theorem for the cycle counterexample detection suite. Theorem 58 is not one of the 11 modified theorems without finite variables that the original version of NEBULA could verify, so the verifications and counterexamples from the original evaluation are all still valid.

Multiple factors contributed to our improved success rates. After the original publication, we improved NEBULA's ability to prove equivalences by loosening the requirements for lemma substitutions. The original version of LEMMALEFT required e'_1, the sub-expression of e_1 in function application form, to be e_1 itself. Likewise, the original version of LEMMARIGHT required e'_2 to be e_2 itself. Additionally, we allow NEBULA to apply two lemma substitutions to a single expression simultaneously, whereas it could apply only one at a time in the original evaluation. In principle, we could allow NEBULA to apply arbitrarily many lemma substitutions at once, but capping the number at two keeps the search space manageable.

We improved our ability to detect cycle counterexamples as well. The current version of NEBULA can look inside target expressions for cycles. The original version of CYL required e', the sub-expression of e'_1 that loops back to e, to be e'_1 itself. Additionally, the original version of CYL required e, the sub-expression of e_1 that starts the looping path, to be e_1 itself. Similarly, the original version of CYR required e' and e to be e'_2 and e_2, respectively.

7.3.2 Finite-variable benchmarks

NEBULA performs well on the unmodified benchmarks, the totality-requiring benchmarks, and the cycle counterexample 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.

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

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

Fig. 23: The `height` function.

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. 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 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.

7.3.3 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 23, 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.

7.3.4 Cycle counterexamples

In our original evaluation, NEBULA found cycle counterexamples for 32 of the 44 theorems that had them. In our new evaluation, we find them for 44 of 45. The one remaining theorem for which NEBULA cannot find a cycle counterexample is Theorem 52, shown in Figure 24. The theorem contains applications of the `rev` and `count` functions. The definition of `rev` also appears in the figure, and the definition of `count` appears in Figure 16. The combination of `count` with `rev` makes cycle counterexample detection impossible because the case statement at the outermost level of `count` branches on the output of `rev` on the right-hand side of each theorem. As we mentioned in Section 7.3.2, `rev` can traverse its whole input list before reaching SWHNF, and in fact it is guaranteed to traverse its whole input list before reaching SWHNF. Consequently, on the symbolic execution branch where `xs` is infinite, the outermost case statement in `count` needs to keep unfolding increasingly

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]

forall n xs . count n xs = count n (rev xs)
```

Fig. 24: The `rev` function and Zeno Theorem 52.

deep layers of nested case statements from `rev` that never eliminate themselves, which prevents our techniques for cycle detection from being used. Even though we can look for cycles inside target expressions, we cannot find an exact match. If we abstract away the case statements, the structure of nested function applications that symbolic execution will encounter on the right-hand side looks like this across successive steps:

```
count n ((rev ...) ++ [x])
count n (((rev ...) ++ [y]) ++ [x])
count n ((((rev ...) ++ [z]) ++ [y]) ++ [x])
```

The ASTs do not align between steps in the way that we would need them to align for a cycle counterexample. The non-symbolic components of the ASTs are structurally distinct at every step. The location of `rev` within the AST changes every time: each iteration adds another `++` application on top of it. More importantly, the `rev` applications are never among the target expressions. The `rev` applications are all arguments of `++` applications, and the target expressions for a function application are inside the function being applied, not the argument that the function receives.

Theorem 53, one of the theorems for which NEBULA finds a cycle counterexample, is nearly identical to Theorem 52 but contains `sort` instead of `rev`. NEBULA can find a cycle counterexample for Theorem 53 because `sort` compares the `Nat` entries in its input list with <= as it traverses the list. (The definition of <= appears in Figure 2 as part of Example 2.1.) If we have two adjacent cyclically defined infinite `Nat` entries in the list, `sort` will become stuck comparing the two entries with <= forever. This creates an opportunity for a one-sided cycle that does not exist for Theorem 52 because `rev` does not manipulate the entries in its input list. The cycle is similar to the one that we discuss in Example 16.

7.3.5 Impact of the time limit

We believe that the 3-minute time limit for the evaluation does not inhibit NEBULA's performance in a significant way. Usually, when NEBULA can prove an equivalence, it finds the cyclic pattern that it needs for coinduction rather quickly. This was true for the original evaluation, and it continues to hold for our updated evaluation. NEBULA's average times for proving equivalences and finding counterexamples in our updated evaluation are all under 10 seconds. Generally, the differences between the theorems that NEBULA verifies and the theorems that NEBULA fails to verify are more qualitative than quantitative: the mere size of the expressions that need to be evaluated is not a significant factor. 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.

```
forall n xs . count n (n : xs) = walkNat n (S (count n xs))
```

Fig. 25: Modified Zeno Theorem 4.

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.

Nevertheless, there are rare situations where NEBULA takes a long time to find a proof. Theorem 55 is one of the two unmodified theorems that NEBULA could not verify in the original evaulation, and its running time is a major outlier. It takes 62.8 seconds to verify, which is the longest individual non-timeout running time that we observed in our evaluation by a wide margin. The next-longest non-timeout running time that we observed is 19.4 seconds, which comes from one of the altered benchmarks with no finite variables that NEBULA could not verify in the original evaluation.

7.3.6 Impact of heuristics

We know that one of our heuristics that we impose for the sake of efficiency limits NEBULA's effectiveness in certain situations. Recall from Section 6.4.1 that NEBULA does not generate potential lemmas for LEMCO from the active region of an expression. There is an additional theorem that we can verify if we disable this heuristic. The theorem is the finiteness-requiring version of Theorem 4 from the Zeno evaluation suite (Sonnex *et al.*, 2012), shown in Figure 25. On the left-hand side, symbolic execution follows the same path that it does in the ordinary version of Theorem 4 that we discussed in Example 5.1:

```
case n === n of
  False -> count n xs
  True -> S (count n xs)
```

Evaluation continues along the same path to another similar state when we concretize `n` as `S n'`:

```
case n' === n' of
  False -> count (S n') xs
  True -> S (count (S n') xs)
```

Because we have a walk function on the right-hand side, we can find an equivalence proof at this point rather than a cycle counterexample. If we allow lemma generation in the active region, then NEBULA can create and prove a lemma that asserts that `n' === n'` and `S n' === S n'` are equivalent. Replacing the scrutinee in the newer state with `S n' === S n'` will allow NEBULA to find an approximation between the two states. The lemma substitution in the active region is a necessary part of the proof: we cannot achieve the same result by applying two lemma substitutions to the branches of the case statement instead. The potential lemma that NEBULA would generate for the branches asserts the equivalence of `count (S n') xs` and `count n' xs`, which is not true in general

because the main recursion of `count` happens on the list being searched, not the number whose occurrences are being counted.

Although disabling the heuristic allows NEBULA to prove the altered version of Theorem 4, there is a downside to allowing potential lemma generation in the active region. In certain situations, increasing the number of potential lemmas that NEBULA needs to consider creates a state explosion problem that prevents it from making progress toward a proof. We ran NEBULA on our test suite with lemma generation allowed in the active region and all other settings kept the same as they are in the main evaluation. With the change in place, NEBULA verifies only 22 of the correct unmodified theorems, 10 of the modified theorems with no finite variables, and 8 of the modified theorems with finite variables. In total, NEBULA hits the time limit for 12 theorems that it can verify with its normal settings. The altered version of Theorem 4 is the only new theorem that becomes verifiable with the heuristic disabled, so we choose to leave our limitations on potential lemma generation in place.

Interestingly, even though NEBULA does not use lemmas for counterexample detection, disabling the heuristic also has an effect on our outcomes for counterexample detection. For the suite of unmodified theorems, NEBULA still finds all 61 counterexamples, but it detects only 43 counterexamples for the cycle counterexample benchmarks instead of 44. The two timeouts are Theorems 52 and 53, which we mentioned in Section 7.3.4. When NEBULA runs on an invalid theorem, it still searches for a proof as it searches for counterexamples, so it makes sense that the heuristic change would affect counterexample detection as well.

7.3.7 Impact of lemmas

Lemma substitutions are an important part of NEBULA's approach to equivalence verification, even though the process of searching for usable lemmas impedes NEBULA's performance when we allow potential lemma generation in the active region. To assess the significance of lemmas in NEBULA's algorithm, we ran NEBULA on our test suite with lemma substitutions disabled and all other settings kept the same as they are for the main evaluation. Without the ability to use lemma substitutions, NEBULA verifies only 20 of the correct unmodified theorems, 8 of the modified theorems with no finite variables, and 7 of the modified theorems with finite variables. All of the verifiable theorems are ones that are also verifiable with lemma substitutions enabled. Also, NEBULA still detects all of the counterexamples that it does normally. In total, NEBULA verifies 16 fewer theorems with lemmas disabled than it does with lemma substitutions enabled, so the effectiveness of NEBULA does depend to a significant extent on its ability to use lemma substitutions.

8 Related work

Coinduction NEBULA relies on coinduction, a well-established proof technique (Gordon, 1995; Rutten, 2000; Gibbons & Hutton, 2005; Sangiorgi, 2009; Kozen & Silva, 2017). 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 & Wand, 2006; Sangiorgi *et al*., 2007). Unlike our approach for NEBULA, the formalizations in Koutavas & 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 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 & Roşu, 2007; Roşu & 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) describe 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 the translation of Haskell code into Coq code, allowing users to verify properties of their Haskell code within Coq. While Breitner *et al*. (2018) discuss 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 & Moskal (2014) describe 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.

Even without cycle counterexamples, we can find a counterexample for each of the invalid theorems by using bottom values. However, this does not entail that cycle counterexamples are unnecessary or that we could emulate the full functionality of NEBULA simply by allowing a strict equivalence checker to concretize variables as bottoms. As we mentioned at the start of Section 4, bottom values and non-termination are not semantically interchangeable. Cycle counterexamples have a different meaning than terminating counterexamples do, and existing equivalence checkers would not be able to detect cycle counterexamples even if we extended them with the ability to reason about bottom values.

Cyclist differs from Zeno, HipSpec, and IsaPlanner in that it relies on *cyclic proofs*. A cyclic proof is an infinite proof tree that follows a looping pattern along every infinite branch (Brotherston, 2005). There is some resemblance between the infinite looping paths in a cyclic proof and the infinite execution paths that loop back on themselves that we consider for NEBULA, but NEBULA's proofs do not qualify as cyclic proofs. We take advantage of cyclic behavior in non-strict functional programs' execution paths, but the proofs about those execution paths are not themselves cyclic. Our proof trees are always finite.

Functional Symbolic Execution NEBULA is not the first tool to perform symbolic execution for a non-strict functional language. Both our formalism and our implementation utilize techniques introduced by G2, a symbolic execution engine for non-strict functional programs (Hallahan *et al.*, 2019). Other prior work in the domain of functional symbolic execution has targeted languages such as Racket (Torlak & Bodik, 2014) and Erlang (Vidal, 2015).

The concept of narrowing in functional logic programming resembles our technique for concretizing symbolic variables during symbolic execution. Narrowing is the process of replacing variables of unspecified value with specific values (Alpuente *et al*., 1996, 2005; Antoy, 2001; López-Fraguas & Sánchez-Hernández, 2002). Moreover, narrowing serves the same purpose in functional logic programming that concretization serves in our formalism: gradual exploration of the possible values of an algebraic data type (Antoy & Hanus, 2010). The specific version of symbolic variable concretization employed by NEBULA comes from G2 (Hallahan *et al.*, 2019).

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 nonlinear integer programs, as opposed to the data-structure-heavy programs that NEBULA targets. Nguyễn *et al*. (2019) use symbolic execution and the size-change principle (Lee *et al*., 2001) to prove termination of functional programs but, unlike NEBULA, cannot prove non-termination.

Symbolic Functions Nguyễn & Van Horn (2015) handle symbolic functions during symbolic execution by using templates to concretize function definitions gradually. It is possible that techniques from Nguyễn & 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. From a theoretical perspective, there are no major impediments to running NEBULA on rewrite rules from Hackage, but the implementation of NEBULA as it exists currently is not capable of running on them. Preparing NEBULA for the task would require additional engineering: the versions of GHC that real-world Haskell packages require can vary, and real-world packages typically have dependencies on other supporting packages.

Acknowledgments

We thank the anonymous OOPSLA 2022 reviewers for their feedback on earlier versions of this paper. We thank Dorel Lucanu for answering our questions about CIRC. The original version of this work was supported by the National Science Foundation under Grant Numbers CCF-2131476, CNS-1565208, CCF-2219995, and CCF-2318974.

Supplementary material

The supplementary material for this article can be found at https://doi.org/10.1017/S0956796825100099.

Conflicts of Interest

The authors' conflicts of interest include Timos Antonopoulos (Yale University), Eric Campbell (Cornell University), Ben Chaimberg (Yale University), Matthew Elacqua (Yale University), Ferhat Erata (Yale University), Nate Foster (Cornell University), William Harris (Amazon Web Services), Samuel Judson (Yale University), Viktor Kuncak (EPFL,

Switzerland), Tancrède Lepoint (Amazon Web Services), Daniel Luick (Yale University), Ning Luo (Northwestern University), James Parker (Galois), Mark Santolucito (Barnard College), Martin Schäf (Amazon Web Services), Scott Shapiro (Yale University), Zhong Shao (Yale University), Robert Soulé (Yale University), Eran Tromer (Boston University), Willem Visser (Amazon Web Services), Xiao Wang (Northwestern University), Thomas Wies (New York University), Ennan Zhai (Alibaba), and Jialu Zhang (University of Waterloo).

Author Homepages

- John Charles Kolesar: https://johnckolesar.github.io
- Ruzica Piskac: https://www.cs.yale.edu/homes/piskac/
- William Triest Hallahan: https://billhallahan.github.io

References

AbdelGawad, M. A. (2019) Induction, coinduction, and fixed points: Intuitions and tutorial. arXiv preprint arXiv:1903.05127.

Alpuente, M., Falaschi, M. & Vidal, G. (1996) Narrowing-driven partial evaluation of functional logic programs. In Programming Languages and Systems–ESOP'96: 6th European Symposium on Programming Linköping, Sweden, April 22–24, 1996 Proceedings 6. Springer, pp. 45–61.

Alpuente, M., Lucas, S., Vidal, G. & Hanus, M. (2005) Specialization of functional logic programs based on needed narrowing. *Theory Practice Logic Program*. **5**(3), 273–303.

Antoy, S. (2001) Evaluation strategies for functional logic programming. *Electron. Notes Theoret. Comput. Sci*. **57**, 1–16.

Antoy, S. & Hanus, M. (2010) Functional logic programming. *Commun. ACM*. **53**(4), 74–85.

Benton, N. (2004) Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Not*. **39**(1), 14–25.

Breitner, J. (2018) hs-to-coq supports coinduction. Available at: https://mobile.twitter.com/nomeata/status/977257104120664064.

Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J. & Weirich, S. (2018) Ready, set, verify! applying hs-to-coq to real-world haskell code (experience report). *Proc. ACM Program. Lang*. **2**(ICFP), 1–16.

Brotherston, J. (2005) Cyclic proofs for first-order logic with inductive definitions. In International Conference on Automated Reasoning with Analytic Tableaux and Related Methods. Springer, pp. 78–92.

Brotherston, J., Gorogiannis, N. & Petersen, R. L. (2012) A generic cyclic theorem prover. In Asian Symposium on Programming Languages and Systems. Springer, pp. 350–367.

Burnim, J., Jalbert, N., Stergiou, C. & Sen, K. (2009) Looper: Lightweight detection of infinite loops at runtime. In 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 161–169.

Campbell, E. H., Hallahan, W. T., Srikumar, P., Cascone, C., Liu, J., Ramamurthy, V., Hojjat, H., Piskac, R., Soulé, R. & Foster, N. (2021) Avenir: Managing data plane diversity with control plane synthesis. In NSDI, pp. 133–153.

Carbin, M., Misailovic, S., Kling, M. & Rinard, M. C. (2011) Detecting and escaping infinite loops with jolt. *European Conference on Object-Oriented Programming*. Springer, pp. 609–633.

Claessen, K., Johansson, M., Rosén, D. & Smallbone, N. (2012) Hipspec: Automating inductive proofs of program properties. In ATx/WInG@ IJCAR. Citeseer, pp. 16–25.

Claessen, K., Johansson, M., Rosén, D. & Smallbone, N. (2013) Automating inductive proofs using theory exploration. In International Conference on Automated Deduction. Springer, pp. 392–406.

Cook, B., Fuhs, C., Nimkar, K. & O'Hearn, P. (2014) Disproving termination with overapproximation. In 2014 Formal Methods in Computer-Aided Design (FMCAD). IEEE, pp. 67–74.
Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J. (2006) Fast and loose reasoning is morally correct. *ACM SIGPLAN Not*. **41**(1), 206–217.
Dixon, L. & Fleuriot, J. (2003) Isaplanner: A prototype proof planner in isabelle. In International Conference on Automated Deduction. Springer, pp. 279–283.
Elliott, C. (2010) Non-strict memoization. Available at: http://conal.net/blog/posts/nonstrict-memoization.
Farina, G. P., Chong, S. & Gaboardi, M. (2019) Relational symbolic execution. In Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, pp. 1–14.
Farmer, A., Sculthorpe, N. & Gill, A. (2015) Reasoning with the hermit: tool support for equational reasoning on ghc core programs. *ACM SIGPLAN Not*. **50**(12), 23–34.
Gibbons, J. & Hutton, G. (2005) Proof methods for corecursive programs. *Fundam. Inf*. **66**(4), 353–366.
Gordon, A. D. (1995) A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, pp. 78–95.
Gupta, A., Henzinger, T. A., Majumdar, R., Rybalchenko, A. & Xu, R.-G. (2008) Proving non-termination. In Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 147–158.
Hallahan, W. T., Xue, A., Bland, M. T., Jhala, R. & Piskac, R. (2019) Lazy counterfactual symbolic execution. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 411–424.
Johansson, M., Dixon, L. & Bundy, A. (2010) Case-analysis for rippling and inductive proof. In International Conference on Interactive Theorem Proving. Springer, pp. 291–306.
Kling, M., Misailovic, S., Carbin, M. & Rinard, M. (2012) Bolt: On-demand infinite loop escape in unmodified binaries. *ACM SIGPLAN Not*. **47**(10), 431–450.
Koutavas, V. & Wand, M. (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, pp. 141–152.
Kozen, D. & Silva, A. (2017) Practical coinduction. *Math. Struct. Comput. Sci*. **27**(7), 1132–1152.
Le, T. C., Antonopoulos, T., Fathololumi, P., Koskinen, E. & Nguyen, T. (2020) Dynamite: Dynamic termination and non-termination proofs. *Proc. ACM Program. Lang*. **4**(OOPSLA), 1–30.
Lee, C. S., Jones, N. D. & Ben-Amram, A. M. (2001) The size-change principle for program termination. In Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 81–92.
Leino, K. R. M. & Moskal, M. (2014) Co-induction simply. In International Symposium on Formal Methods. Springer, pp. 382–398.
López-Fraguas, F. J. & Sánchez-Hernández, J. (2002) Narrowing failure in functional logic programming. In International Symposium on Functional and Logic Programming. Springer, pp. 212–227.
Lucanu, D. & Roşu, G. (2007) Circ: A circular coinductive prover. In International Conference on Algebra and Coalgebra in Computer Science. Springer, pp. 372–378.
Mastorou, L., Papaspyrou, N. & Vazou, N. (2022) Coinduction inductively: Mechanizing coinductive proofs in liquid haskell. Haskell Symposium.
Milovancevic, D., Giunta, J. & Kuncak, V. (2021) *On Proving and Disproving Equivalence of Functional Programming Assignments*. Technical report.
Nguyễn, P. C., Gilray, T., Tobin-Hochstadt, S. & Van Horn, D. (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, pp. 845–859.
Nguyễn, P. C. & Van Horn, D. (2015) Relatively complete counterexamples for higher-order programs. *ACM SIGPLAN Not*. **50**(6), 446–456.

Person, S., Dwyer, M. B., Elbaum, S. & Păsăreanu, C. S. (2008) Differential symbolic execution. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 226–237.
Peyton Jones, S., Tolmach, A. & Hoare, T. (2001) Playing by the rules: rewriting as a practical optimisation technique in ghc. In Haskell Workshop, pp. 203–233.
Peyton Jones, S. L. (1996) Compiling haskell by program transformation: A report from the trenches. In European Symposium on Programming. Springer, pp. 18–44.
Roşu, G. & Lucanu, D. (2009) Circular coinduction: A proof theoretical foundation. In International Conference on Algebra and Coalgebra in Computer Science. Springer, pp. 127–144.
Rutten, J. J. (2000) Universal coalgebra: A theory of systems. *Theoret. Comput. Sci*. **249**(1), 3–80.
Sangiorgi, D. (2009) On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst. (TOPLAS)*. **31**(4), 1–41.
Sangiorgi, D., Kobayashi, N. & Sumii, E. (2007) Environmental bisimulations for higher-order languages. In 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007). IEEE, pp. 293–302.
Schkufza, E., Sharma, R. & Aiken, A. (2013) Stochastic superoptimization. *ACM SIGARCH Comput. Archit. News* **41**(1), 305–316.
Schuts, M., Hooman, J. & Vaandrager, F. (2016) Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In International Conference on Integrated Formal Methods. Springer, pp. 311–325.
Smith, C. & Albarghouthi, A. (2019) Program synthesis with equivalence reduction. In International Conference on Verification, Model Checking, and Abstract Interpretation. Springer, pp. 24–47.
Sonnex, W., Drossopoulou, S. & Eisenbach, S. (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, pp. 407–421.
Spector-Zabusky, A., Breitner, J., Rizkallah, C. & Weirich, S. (2018) Total haskell is reasonable coq. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 14–27.
Torlak, E. & Bodik, R. (2014) A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Not*. **49**(6), 530–541.
Vidal, G. (2015) Towards symbolic execution in erlang. In Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24–27, 2014. Revised Selected Papers 9. Springer, pp. 351–360.

Appendix A

A.1. Symbolic weak head normal form

Theorem 1. *For any expression e and symbolic store Y, there exist an expression e′ and symbolic store Y′ such that* $(e, Y) \hookrightarrow (e', Y')$ *if and only if e is not in SWHNF.*

Proof For the first direction, we need to show that any expression that can take a step is not in SWHNF. Start from the assumption that $(e, Y) \hookrightarrow (e', Y')$ for some e' and Y'. To confirm that e is not in SWHNF, we can perform case analysis and induction on our reduction rules. Most of the reduction rules apply to expressions that, by definition, are never in SWHNF. The rule VAR applies to non-symbolic variables. The rules APPλ and BTAPP apply to applications where the left-hand side is not a data constructor. The rules CSEV, CSDC, FRDC, LKDC, BTDC, and BTCS all apply to case statements. In the extended version of our formalism where we allow symbolic functions, the rules HGLOOKUP and HGFRESH apply to applications where the left-hand side is a symbolic variable.

The only remaining rule to consider is App. With it, we can conclude $(e, Y) \hookrightarrow (e', Y')$ if e is $e_1\ e_2$, e' is $e_1'\ e_2$, and $(e_1, Y) \hookrightarrow (e_1', Y')$. In this situation, since e is an application, the only way for it to be in SWHNF is for e_1 to be a data constructor D. The constructor D on its own is in SWHNF, so our inductive hypothesis gives us that it cannot undergo any reduction steps. This contradicts our earlier assumption that $(e_1, Y) \hookrightarrow (e_1', Y')$, so e must not be in SWHNF, and the first direction holds.

For the second direction, we need to show that any expression that is not in SWHNF can take a step. Assume that e is not in SWHNF. We will apply an inductive argument again. There are multiple cases to consider. If e is a non-symbolic variable, then Var applies to it because we assume that all non-symbolic variables have mappings in our implicit environment. If e is an application $e_1\ e_2$ where the left-hand side is not a data constructor, we have multiple sub-cases to consider depending on what e_1 is

- If e_1 is not in SWHNF, we can apply App since our inductive hypothesis gives us that e_1 can reduce to some other expression.
- If e_1 is a lambda, we can apply Appλ.
- If e_1 is a bottom, we can apply BtApp.
- In the ordinary version of our formalism, e_1 cannot be a symbolic variable s, but the result still holds in the extended version of our formalism where we allow symbolic variables to be function-typed. We can apply either HgLookup or HgFresh depending on whether $s\ e_2$ has a mapping in Y.

If e is a case statement $\texttt{case}\ e_1\ \texttt{of}\ \{\vec{a}\}$, we again have multiple sub-cases to consider depending on what e_1 is

- If e_1 is not in SWHNF, we can apply CsEv since our inductive hypothesis gives us that e_1 can reduce to some other expression.
- If e_1 is a symbolic variable that has a mapping in Y, we can apply LkDC.
- If e_1 is a symbolic variable that does not have a mapping in Y, we can apply FrDC.
- If e_1 is a data constructor application, we can apply CsDC since we assume that all case statements are exhaustive.
- If e_1 is a bottom, we can apply BtCs.
- Lastly, e_1 cannot be a lambda because that would violate our assumption that scrutinees are never function-typed.

There are no more options for non-SWHNF expressions, so the second direction holds. ■

A.2. Approximation lemmas and theorems

Lemma 1 ($\sqsubseteq$ preserved by inlining)**.** *If* $(e_1, Y_1) \sqsubseteq_V (s, Y_2)$ *and* $e_2 = \texttt{lookup}(s, Y_2)$*, then* $(e_1, Y_1) \sqsubseteq_V (e_2, Y_2)$*.*

Proof Follows immediately from the definitions of $\sqsubseteq$-Sym1 and $\sqsubseteq$-Eval. ■

Corollary 1. *If* $(e_1, Y_1) \sqsubseteq_V (e_2, Y_2)$ *and* e_2' *is* e_2 *with all symbolic variable concretizations from* Y_2 *inlined, then* $(e_1, Y_1) \sqsubseteq_V (e_2', Y_2)$*.*

Lemma 2 ($\sqsubseteq$ transitive). *If* $S_1 \sqsubseteq S_2$ *and* $S_2 \sqsubseteq S_3$ *then* $S_1 \sqsubseteq S_3$.

Proof To start, note that we can assume that there is no overlap between the symbolic variables in S_2 and the symbolic variables in S_3. If there is any overlap, we can simply give fresh names to all of the symbolic variables in S_2 to eliminate the overlap while preserving the approximations between S_1 and S_2 and between S_2 and S_3. The result that we derive at the end is still the one that we wanted originally, namely that $S_1 \sqsubseteq S_3$, since S_1 and S_3 do not change.

Let $(e_1, Y_1) = S_1$, let $(e_2, Y_2) = S_2$, and let $(e_3, Y_3) = S_3$. Let V be the mapping such that $S_1 \sqsubseteq_V S_2$, and let V' be the mapping such that $S_2 \sqsubseteq_{V'} S_3$. For the new approximation, we will need a new mapping V''. For each mapping $(s, e) \in V'$, let e' be e with all of the symbolic variables from Y_2 inlined and let $\texttt{lookup}(s, V'') = e'$. Also, for any mapping $(s, e) \in V$, let V'' map s to e. There are no common symbolic variables between S_2 and S_3, so, if any symbolic variable s is mapped by both V and V', at least one of the two mappings must be irrelevant. V'' should contain a mapping for s based on the mapping in V or V' that is actually needed for one of the two original approximations. We will prove that $S_1 \sqsubseteq_{V''} S_3$ by induction on the relation $S_2 \sqsubseteq_{V'} S_3$.

Deterministic Evaluation on the Left This is not one of the main cases, but we are covering it here so that, in the subsequent cases, we can ignore the possibility that $\sqsubseteq$-EVAL is the main rule used for the approximation $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$. Assume for this case that it is the main rule. This means that there exists some e_1' such that $(e_1, Y_1) \hookrightarrow^* (e_1', Y_1)$ and $(e_1',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$. If we can use the facts that $(e_1',\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ and $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ to derive that $(e_1',\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$, it follows immediately that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$ by $\sqsubseteq$-EVAL. This means that, in the following cases, we can ignore the possibility that $\sqsubseteq$-EVAL is used as the main rule for the approximation between e_1 and e_2.

Deterministic Evaluation in the Middle Suppose that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-EVAL, meaning that there exists some e_2' such that $(e_2, Y_2) \hookrightarrow^* (e_2', Y_2)$ and $(e_2',\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$. Because $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$, Lemma 8 gives us that there exists some e_1' such that $(e_1, Y_1) \hookrightarrow^* (e_1', Y_1)$ and $(e_1',\ Y_1) \sqsubseteq_V (e_2',\ Y_2)$. At this point, the inductive hypothesis lets us derive that $(e_1',\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$. Since $(e_1, Y_1) \hookrightarrow^* (e_1', Y_1)$, it follows from $\sqsubseteq$-EVAL now that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$.

Concretized Symbolic Variable in the Middle Assume that the formula $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-SYM0, which means that e_2 is a symbolic variable s_2, that Y_2 maps s_2 to some e_2', and that $(e_2',\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$. Note that we can rewrite $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ as $(e_1,\ Y_1) \sqsubseteq_V (s_2,\ Y_2)$. We know that $\sqsubseteq$-EVAL is not the rule being applied to establish this result, and we also know that s_2 has a mapping e_2' in Y_2, so the only rule that could have given us $(e_1,\ Y_1) \sqsubseteq_V (s_2,\ Y_2)$ is $\sqsubseteq$-SYM1. From the premises of that rule, we have $(e_1,\ Y_1) \sqsubseteq_V (e_2',\ Y_2)$. At this point, we can apply the inductive hypothesis to $(e_1,\ Y_1) \sqsubseteq_V (e_2',\ Y_2)$ and $(e_2',\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ to conclude that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$ holds.

Concretized Symbolic Variable on the Right Assume that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-SYM1, meaning that e_3 is a symbolic variable s_3 that Y_3 maps to some e_3'. In this case, we know that there exists some $e' = \texttt{lookup}(s_3, V')$ and some other expression e'' such that $(e', Y_2) \hookrightarrow^* (e'', Y_2)$, $(e_2,\ Y_2) \sqsubseteq_{V'} (e'',\ Y_3)$, and $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3',\ Y_3)$. We want to find

some expressions $e_1' = \texttt{lookup}(s_3, V'')$ and e_1'' such that $(e_1', Y_1) \hookrightarrow^* (e_1'', Y_1)$, $(e_1,\ Y_1) \sqsubseteq_{V''} (e_1'',\ Y_3)$, and $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3',\ Y_3)$ so we can apply $\sqsubseteq$-SYM1.

We already have a definition of e_1' from V'': e_1' is e' with all of the symbolic variables from Y_2 inlined. Let e_1'' be e'' with all of the symbolic variables from Y_2 inlined. We know that $(e', Y_2) \hookrightarrow^* (e'', Y_2)$, so it must be the case that $(e_1', \{\}) \hookrightarrow^* (e_1'', \{\})$. All of the symbolic variables that are used in the evaluation from e' to e'' are inlined for e_1', so there is no need to use concretizations from the symbolic store in the evaluation from e_1' to e_1''. It follows that $(e_1', Y_1) \hookrightarrow^* (e_1'', Y_1)$ because adding unused concretizations to a state does not interfere with its evaluation.

Also, since $(e_2,\ Y_2) \sqsubseteq_{V'} (e'',\ Y_3)$, Corollary 1 gives us that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_1'',\ Y_3)$ as well. We can apply the inductive hypothesis to this to derive that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_1'',\ Y_3)$. Likewise, applying the inductive hypothesis to $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3',\ Y_3)$ gives us that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3',\ Y_3)$. All of these conclusions together allow us to apply $\sqsubseteq$-SYM1.

Non-Concretized Symbolic Variable on the Right Now assume that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-SYM2, meaning that e_3 is a symbolic variable s_3, $s_3 \notin Y_3$, and there exist some $e' = \texttt{lookup}(s_3, V')$ and e'' such that $(e', Y_2) \hookrightarrow^* (e'', Y_2)$ and $(e_2,\ Y_2) \sqsubseteq_{V'} (e'',\ Y_3)$. We want to find some expressions $e_1' = \texttt{lookup}(s_3, V'')$ and e_1'' such that $(e_1', Y_1) \hookrightarrow^* (e_1'', Y_1)$ and $(e_1,\ Y_1) \sqsubseteq_{V''} (e_1'',\ Y_3)$.

We already have a definition of e_1' from V'': e_1' is e' with all of the symbolic variables from Y_2 inlined. Let e_1'' be e'' with all of the symbolic variables from Y_2 inlined. We know that $(e', Y_2) \hookrightarrow^* (e'', Y_2)$, so it must be the case that $(e_1', \{\}) \hookrightarrow^* (e_1'', \{\})$. All of the symbolic variables that are used in the evaluation from e' to e'' are inlined for e_1', so there is no need to use concretizations from the symbolic store in the evaluation from e_1' to e_1''. It follows that $(e_1', Y_1) \hookrightarrow^* (e_1'', Y_1)$ because adding unused concretizations to a state does not interfere with its evaluation.

Also, since $(e_2,\ Y_2) \sqsubseteq_{V'} (e'',\ Y_3)$, Corollary 1 gives us that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_1'',\ Y_3)$. We can apply the inductive hypothesis to this to derive that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_1'',\ Y_3)$. This gives us everything that we need to apply $\sqsubseteq$-SYM2.

Non-Symbolic Variables Now assume that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-VAR, meaning that $e_2 = e_3 = x$. Symbolic variables cannot map to non-symbolic variables, and we covered $\sqsubseteq$-EVAL for the approximation between e_1 and e_2 already, so, in order for $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ to hold, it must be the case that $e_1 = x$ as well. This lets us apply $\sqsubseteq$-VAR to derive immediately that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$.

Lambdas Suppose that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-LAM, meaning that $e_2 = \lambda x_2 \,.\, e_2'$, $e_3 = \lambda x_3 \,.\, e_3'$, and $(e_2'[x'/x_2],\ Y_2) \sqsubseteq_{V'} (e_3'[x'/x_3],\ Y_3)$ for some fresh variable x'. In this case, e_1 must be a lambda $\lambda x_1 \,.\, e_1'$ as well because we covered $\sqsubseteq$-EVAL already. The fact that $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ implies that $(e_1'[x/x_1],\ Y_1) \sqsubseteq_V (e_2'[x/x_2],\ Y_2)$ for some other fresh variable x. We can use Lemma 3 to derive that $(e_2'[x'/x_2][x/x'],\ Y_2) \sqsubseteq_{V'} (e_3'[x'/x_3][x/x'],\ Y_3)$. We can simplify this to $(e_2'[x/x_2],\ Y_2) \sqsubseteq_{V'} (e_3'[x/x_3],\ Y_3)$ because x' is fresh and therefore cannot appear in e_2' or e_3'. Since we know now that $(e_1'[x/x_1],\ Y_1) \sqsubseteq_V (e_2'[x/x_2],\ Y_2)$ and $(e_2'[x/x_2],\ Y_2) \sqsubseteq_{V'} (e_3'[x/x_3],\ Y_3)$, we can apply the inductive hypothesis to derive that $(e_1'[x/x_1],\ Y_1) \sqsubseteq_{V''} (e_3'[x/x_3],\ Y_3)$. Then we can apply $\sqsubseteq$-LAM to establish that $(\lambda x_1 \,.\, e_1',\ Y_1) \sqsubseteq_{V''} (\lambda x_3 \,.\, e_3',\ Y_3)$, which was our goal.

Data Constructors Assume that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-DC, meaning that $e_2 = e_3 = D$. Since we can ignore the possibility that $\sqsubseteq$-EVAL applies between e_1 and e_2, $\sqsubseteq$-DC must apply between e_1 and e_2. This means that $e_1 = D$, so we can apply $\sqsubseteq$-DC to e_1 and e_3 to derive that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$.

Applications The next possibility to consider is that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-APP, meaning that $e_2 = e_2'\ e_2''$, $e_3 = e_3'\ e_3''$, $(e_2',\ Y_2) \sqsubseteq_{V'} (e_3',\ Y_3)$, and $(e_2'',\ Y_2) \sqsubseteq_{V'} (e_3'',\ Y_3)$. e_1 must be an application $e_1'\ e_1''$ in order for the approximation between e_1 and e_2 to hold, since the approximation between the two does not use $\sqsubseteq$-EVAL as the main rule. Our assumption that $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ implies that $(e_1',\ Y_1) \sqsubseteq_V (e_2',\ Y_2)$ and $(e_1'',\ Y_1) \sqsubseteq_V (e_2'',\ Y_2)$. We can apply the inductive hypothesis twice over now to derive that $(e_1',\ Y_1) \sqsubseteq_{V''} (e_3',\ Y_3)$ and $(e_1'',\ Y_1) \sqsubseteq_{V''} (e_3'',\ Y_3)$. Next, we can apply $\sqsubseteq$-APP to conclude from these that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$, which is what we wanted to show.

Case Expressions Now assume that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-CASE, meaning that $e_2 = \texttt{case}\ e_2'\ \texttt{of}\ \{\vec{a}_2\}$ and $e_3 = \texttt{case}\ e_3'\ \texttt{of}\ \{\vec{a}_3\}$, where $(e_2',\ Y_2) \sqsubseteq_{V'} (e_3',\ Y_3)$ and, for any aligning branch pair of $(D\,\vec{x}_2 \to e_2^a) \in a_2$ and $(D\,\vec{x}_3 \to e_3^a) \in a_3$, we know that $(e_2^a,\ Y_2) \sqsubseteq_{V'} (e_3^a[\vec{x}_2/\vec{x}_3],\ Y_3)$. Recall that we can ignore $\sqsubseteq$-EVAL for the approximation between e_1 and e_2. Since e_2 approximates e_1, it must be the case that $e_1 = \texttt{case}\ e_1'\ \texttt{of}\ \{\vec{a}_1\}$, that $(e_1',\ Y_1) \sqsubseteq_V (e_2',\ Y_2)$, and that, for any aligning branch pair of $(D\,\vec{x}_1 \to e_1^a) \in a_1$ and $(D\,\vec{x}_2 \to e_2^a) \in a_2$, $(e_1^a,\ Y_1) \sqsubseteq_V (e_2^a[\vec{x}_1/\vec{x}_2],\ Y_2)$. Because $(e_1',\ Y_1) \sqsubseteq_V (e_2',\ Y_2)$ and $(e_2',\ Y_2) \sqsubseteq_{V'} (e_3',\ Y_3)$, we know from the inductive hypothesis that $(e_1',\ Y_1) \sqsubseteq_{V''} (e_3',\ Y_3)$.

Let $(D\,\vec{x}_1 \to e_1^a)$ be an alternative in $\vec{a}_1$. Because case statement branches are exhaustive, we know that there is an alternative $(D\,\vec{x}_2 \to e_2^a)$ from $\vec{a}_2$ such that $(e_1^a,\ Y_1) \sqsubseteq_V (e_2^a[\vec{x}_1/\vec{x}_2],\ Y_2)$. For this same alternative, we also know that there is an alternative $(D\,\vec{x}_3 \to e_3^a) \in \vec{a}_3$ such that $(e_2^a,\ Y_2) \sqsubseteq_{V'} (e_3^a[\vec{x}_2/\vec{x}_3],\ Y_3)$. We can rewrite the approximations for alternatives equivalently as $(e_2^a[\vec{x'}/\vec{x}_2],\ Y_2) \sqsubseteq_{V'} (e_3^a[\vec{x'}/\vec{x}_3],\ Y_3)$ and $(e_1^a[\vec{x}/\vec{x}_1],\ Y_1) \sqsubseteq_V (e_2^a[\vec{x}/\vec{x}_2],\ Y_2)$ for fresh variable vectors $\vec{x}$ and $\vec{x'}$. Because $\vec{x'}$ is fresh, Lemma 3 lets us rewrite the first of these approximations again as $(e_2^a[\vec{x}/\vec{x}_2],\ Y_2) \sqsubseteq_{V'} (e_3^a[\vec{x}/\vec{x}_3],\ Y_3)$. (Replacing $\vec{x}_2$ or $\vec{x}_3$ with $\vec{x'}$ and then replacing $\vec{x'}$ with $\vec{x}$ is equivalent to replacing $\vec{x}_2$ or $\vec{x}_3$ with $\vec{x}$ directly since $\vec{x'}$ does not appear in e_2^a or e_3^a.) At this point, we can apply the inductive hypothesis again. Chaining the two approximations together gives us that $(e_1^a[\vec{x}/\vec{x}_1],\ Y_1) \sqsubseteq_{V''} (e_3^a[\vec{x}/\vec{x}_3],\ Y_3)$. We can eliminate the fresh variable, converting the approximation into $(e_1^a,\ Y_1) \sqsubseteq_{V''} (e_3^a[\vec{x}_1/\vec{x}_3],\ Y_3)$. An approximation of this form must hold for any alternative in a_1, and the branches must be in a corresponding order for all three case statements, so we know now that all of the requirements for $(\texttt{case}\ e_1'\ \texttt{of}\ \{\vec{a}_1\},\ Y_1) \sqsubseteq_{V''} (\texttt{case}\ e_3'\ \texttt{of}\ \{\vec{a}_3\},\ Y_3)$ hold.

Bottoms Now suppose that $(e_2,\ Y_2) \sqsubseteq_{V'} (e_3,\ Y_3)$ comes from $\sqsubseteq$-BT, meaning that $e_2 = e_3 = \bot^L$ for some label L. Since $\sqsubseteq$-EVAL is not used between e_1 and e_2, the only way that $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ can hold is for e_1 to be $\bot^L$ as well. This means that we can apply $\sqsubseteq$-BT on e_1 and e_3 to derive that $(e_1,\ Y_1) \sqsubseteq_{V''} (e_3,\ Y_3)$. ■

Lemma 3 ($\sqsubseteq_V$ substitution)**.** *Given expressions e_1, e_2, symbolic stores Y_1 and Y_2 involving some variable x, expressions e_1', e_2', and some V such that $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$ and $(e_1',\ Y_1) \sqsubseteq_V (e_2',\ Y_2)$ then $(e_1\ [e_1'\ /\ x],\ Y_1) \sqsubseteq_V (e_2\ [e_2'\ /\ x],\ Y_2)$.*

Proof Case analysis and induction on definition of $\sqsubseteq_V$. ■

Lemma 4. *Suppose* $(e_1,\ Y_1) \sqsubseteq_V (e_2,\ Y_2)$. *For any e and any s that does not appear in* e_2, *it is the case that* $(e_1,\ Y_1) \sqsubseteq_{V\{s \to e\}} (e_2,\ Y_2)$.

Proof Case analysis and induction on definition of $\sqsubseteq_V$. ■

Lemma 5. *If* $S_1 \hookrightarrow S_2$, *and there exists some e such that* $\mathtt{SWHNF}(e)$ *and* $S_1 \hookrightarrow^* (e, \_)$, *then* $S_2 \not\sqsubseteq S_1$.

Proof Case analysis based on the expression in S_1 and induction on $\hookrightarrow$ in the possible reductions. The only difficulty is that a variable may reduce to itself. However, in this case, the state will never reach SWHNF. ■

Lemma 6. *If* $(e_1,\ Y_1) \hookrightarrow^* (e_k,\ Y_k)$ *in k steps, then* $(e_1,\ Y_k) \hookrightarrow^* (e_k,\ Y_k)$. *Further, the next k steps in the reduction of* $(e_1,\ Y_k)$ *are deterministic.*

Proof Case analysis of the reduction rules. The only rule which may be applied nondeterministically is FrDC, since a symbolic variable that is not mapped in Y may be replaced by any constructor of the appropriate type. The reduction of $(e_1,\ Y_k) \hookrightarrow^* (e_k,\ Y_k)$ will proceed exactly as the reduction of $(e_1,\ Y_1) \hookrightarrow^* (e_k,\ Y_k)$ except that any applications of FrDC will be substituted for applications of LkDC, which will return the constructor application inserted into Y by FrDC. ■

Lemma 7. *If* S_1 *can be reduced to* S_2 *in k steps, and there is some* S_1' *such that* $S_1 \sqsubseteq S_1'$, *then there is some* S_2' *such that* S_1' *can be reduced to* S_2' *in* k' *steps, where* $k' \leq k$.

Proof Follows from Theorem 2. ■

Lemma 8 (Symbolic Execution Determinism)**.** *Let* $S_1 = (e_1,\ Y_1)$ *and* $S_2 = (e_1,\ Y_2)$ *be states such that* $S_1 \sqsubseteq_V S_2$. *If* $S_2 \hookrightarrow S_2'$ *where* $S_2' = (e_2',\ Y_2)$, *then there exists* $S_1' = (e_1',\ Y_1)$ *such that* $S_1 \hookrightarrow^* S_1'$ *and* $S_1' \sqsubseteq_V S_2'$.

Proof We proceed by case analysis on the expression e_2.

Variable If $(x,\ Y_1) \sqsubseteq_V (x,\ Y_2)$, both sides can only be reduced by Var. Thus, the theorem trivially holds.

Application If $(e_1^1\ e_1^2,\ Y_1) \sqsubseteq_V (e_2^1\ e_2^2,\ Y_2)$, reduction may proceed on the right by App or Appλ. In either case, the same rule must be applicable on the left, preserving the approximation by induction on the size of the expression.

Case Suppose $(\mathtt{case}\ e_1^b\ \mathtt{of}\ \{\vec{a_1}\},\ Y_1) \sqsubseteq_V (\mathtt{case}\ e_2^b\ \mathtt{of}\ \{\vec{a_2}\},\ Y_2)$. If the rule CsEv or CsDC is applicable on the right-hand side, the same rule must be applicable on the left-hand side, and the lemma holds by induction on the size of the expression. FrDC cannot be applied on the right-hand side, because it is nondeterministic. If LkDC is applicable on the right-hand side, then e_2^b is some s_2, such that there is an $e = \mathtt{lookup}(s_2, Y_2)$. By the definition of $\sqsubseteq$, $(e_1^b,\ Y_1) \sqsubseteq_V (e,\ Y_2)$. It must be the case that $(e_1^b,\ Y_1) \hookrightarrow^* (e_1^{b'},\ Y_1)$ such

that $(e_1^{b'},\ Y_1) \sqsubseteq_V (e,\ Y_2)$. Thus, both states may continue evaluation along corresponding alternative expressions, preserving the approximation.

Symbolic Variables, Lambdas, Constructors, Bottom Symbolic variables, lambdas, data constructors, and bottoms are already in SWHNF, so they cannot be reduced deterministically. The theorem holds trivially in these cases.

■

A.3. Reduction sequence lemmas

For our lemmas about reduction sequences, we need to introduce some new terminology: a *non-approximating reduction sequence* is a reduction sequence in which no state is approximated by a past state, that is, $\forall i < j . S_j \not\sqsubseteq S_i$.

Reduction Sequences and Approximation To establish the soundness of coinduction, we rely on the following lemma, which relates reduction sequences and approximation:

Lemma 9. *Let p be a predicate on states such that $S_1 \sqsubseteq S_2 \wedge p(S_1) \implies p(S_2)$. If there exists a reduction sequence $S_{\hookrightarrow} = S_1, \ldots, S_n$ and $p(S_n)$, then there exists some non-approximating reduction sequence $S'_{\hookrightarrow} = S'_1, S'_2, \ldots, S'_{n'}$, where $S'_1 = S_1$ and $p(S'_{n'})$.*

Proof We proceed by induction on the length n of the reduction sequence $S_{\hookrightarrow}$.

Base Case - n = 2 By Lemma 5.

Inductive Step—Assume for $n \leq k$, Show for $n = k+1$ If, for all $1 \leq i < j \leq k+1$, it is the case that $S_j \not\sqsubseteq S_i$, then we are done. Otherwise, let i and j be two indices such that $1 \leq i < j \leq k+1$ and $S_j \sqsubseteq S_i$. S_j reduces to S_{k+1} in $k+1-j$ steps. Then, by Lemma 7, S_i can be reduced to some state $S'_{k+1} = (e'_{k+1},\ Y'_{k+1})$, such that $S_{k+1} \sqsubseteq S'_{k+1}$, in at most $k+1-j$ steps. Since $S_{k+1} \sqsubseteq S'_{k+1} \wedge p(S_{k+1})$, it must also be the case that $p(S'_{k+1})$ holds. Since S_1 can be reduced to S_i in $i-1$ steps and S_i can be reduced to S'_{k+1} in $k+1-j$ steps, S_1 can be reduced to S'_{k+1} in $i-1+k+1-j+1$ steps (where the extra "+1" comes from the reduction between states S_i and S_j). Since $i-1+k+1-j+1 = k+1-(j-i) \leq k$, this lemma follows from the inductive hypothesis. ■

Corollary 2. *If there exists a reduction sequence $S_{\hookrightarrow} = S_1, \ldots, S_n = (e_n, Y_n)$ and $\texttt{SWHNF}(e_n)$, then there exists some non-approximating reduction sequence $S'_{\hookrightarrow} = S'_1 = S_1, S'_2, \ldots, S'_{n'} = (e_{n'}, Y_{n'})$, where $\texttt{SWHNF}(e_{n'})$.*

Corollary 3. *If there exists a reduction sequence $S_{\hookrightarrow} = S_1, \ldots, S_n$ and $S_A \sqsubseteq S_n$, then there exists some non-approximating reduction sequence $S'_{\hookrightarrow} = S'_1 = S_1, S'_2, \ldots, S'_{n'}$, where $S_A \sqsubseteq S_{n'}$.*

Proof Consider Lemma 9 with $p(S) = S_A \sqsubseteq S$, which satisfies $S_1 \sqsubseteq S_2 \wedge p(S_1) \implies p(S_2)$ by the transitivity of $\sqsubseteq$ (Lemma 2). ■

Lemma 10. *Let p be a predicate on states such that $S_1 \sqsubseteq S_2 \wedge p(S_1) \implies p(S_2)$. Let $S_{\hookrightarrow} = S_1, \ldots, S_n$ be a non-approximating reduction sequence which calls f a minimal number*

of times while satisfying $p(e_n)$. *Let* $S_1^L = (e_1^L,\ Y_1^L)$ *and* $S_2^L = (e_2^L,\ Y_1^L)$ *be states such that* $S_1^L \equiv S_2^L$. *Pick* k *such that* $e_k = f e_k^1 \ldots e_k^t$ *(*e_k *is in function application form),* $f \notin e_1^L$, *and for some* V *we have* $\exists e_k' \preccurlyeq e_k.(e_k',\ Y_k) \sqsubseteq_V (e_1^L,\ Y_1^L)$. *Let* $S_k' = e_k\ [(e_2^L\ [V(s)\ /\ s])\ /\ e_k']$. *Then there exists some reduction sequence* $S_{\hookrightarrow}' = S_k' \ldots S_m'$ *such that* $p(S_m')$ *and* $\forall 1 \le i \le k, k \le j \le m.S_j' \not\sqsubseteq S_i$.

Proof The existence of $S_{\hookrightarrow}' = S_k' \ldots S_m'$ such that $p(S_m')$ is satisfied is straightforward, since all we have done is substitute one subexpression for an equivalent subexpression.

Suppose that in reduction sequence $S_{\hookrightarrow}$, the function f is called x times before state k, and y times after state k. Thus, it is called $x + y + 1$ times in total (the 1 extra time being at state k itself). Since $(e_2^L\ [V(s)\ /\ s])$ does not contain f, there must be reduction of S_{k+1}' to S_m' which calls f exactly y times. Now suppose there exist i and k such that $1 \le i \le k, k \le j \le m$ and $S_j' \sqsubseteq S_i$. S_j' must be reducible to S_m' calling f at most y times. Then, by Lemma 9, S_i must also be able to be reduced to satisfy p calling f at most y times, which contradicts our assumption that $S_{\hookrightarrow}$ calls f a minimal number of times. Thus, it must be that for all $1 \le i \le k, k \le j \le m$, we have $S_j' \not\sqsubseteq S_i$.

■

Lemma 11. *Consider a finite paired reduction sequence* $S_{\hookrightarrow} = (e_1^1,\ e_1^2,\ Y_1) \ldots (e_k^1,\ e_k^2,\ Y_k)$. *There exists a paired reduction sequence* $S_{\hookrightarrow}' = (e_1^{1'},\ e_1^{2'},\ Y_1') \ldots (e_k^{1'},\ e_k^{2'},\ Y_k')$ *with the same initial and final expressions, but such that all reductions of the first expression are completed before any reductions of the second expression. That is,* $e_1^{1'} = e_1^1$, $e_1^{2'} = e_1^2$, $e_k^{1'} = e_k^1$, $e_k^{2'} = e_k^2$, *and there exists some* b *such that* $\forall 1 \le i \le b.e_b^{2'} = e_{b+1}^{2'}$ *and* $\forall b < i \le k.e_b^{1'} = e_{b+1}^{1'}$.

Proof Follows from reasoning similar to that required for Lemma 6.

The only rules which may cause any sort of interaction between the evaluation of e^1 and e^2 are FRDC, BTDC, and LKDC, which set and lookup variables in the same symbolic store. Thus, any two neighboring reductions where the reduction of e^2 happens before the reduction of e^1 may be swapped. The only catch is that that if FRDC or BTDC is being applied to a variable s in e^2, and LKDC is being applied to that same variable in e^1, then the rules being applied to each state must also be swapped. That is, the application of FRDC or BTDC on e^2 will become an application of LKDC, and the application of LKDC on e^1 will become an application of FRDC or BTDC. ■

Lemma 12. *Consider an infinite paired reduction sequence* $S_{\hookrightarrow} = (e_1^1,\ e_1^2,\ Y_1) \ldots (e_k^1,\ e_k^2,\ Y_k)$, *such that the evaluation of* e_1^1 *(resp.* e_1^2*) eventually reaches SWHNF. That is, there exists some* b *such that* $\forall b < i \le k.e_b^1 = e_{b+1}^1$. *Then there exists an infinite paired reduction sequence* $S_{\hookrightarrow}'$: $S_{\hookrightarrow}' = (e_1^{1'},\ e_1^{2'},\ Y_1') \ldots (e_k^{1'},\ e_k^{2'},\ Y_k')$ *with the same initial expression, but such that all reductions of the first expression are completed before any reductions of the second expression. That is,* $e_1^{1'} = e_1^1$, $e_1^{2'} = e_1^2$, *and there exists some* b' *such that* $\forall 1 \le i \le b'.e_{b'}^{2'} = e_{b'+1}^{2'}$ *and* $\forall b' < i \le k.e_{b'}^{1'} = e_{b'+1}^{1'}$.

Proof Follows from the same basic argument as Lemma 11. ■

Lemma 9 can be extended to apply to paired reduction sequences, even with the choice of predicate differing between the first and second state:

Lemma 13. *Let p and q be predicates on states such that $S_1 \sqsubseteq S_2 \wedge p(S_1) \implies p(S_2)$ and $S_1 \sqsubseteq S_2 \wedge q(S_1) \implies q(S_2)$. If there exists a (possibly infinite) paired reduction sequence $S_{\hookrightarrow} = S_1, \ldots, S_n$, where $S_n = (e_n^1, e_n^2, Y_n)$ and $p((e_n^1,\ Y_n))$ and $q((e_n^2,\ Y_n))$, then there exists some (possibly infinite) paired reduction sequence $S'_{\hookrightarrow} = S'_1, S'_2, \ldots, S'_{n'}$ with $S'_1 = S_1$, where $p((e'^1_n,\ Y'_n))$ and $q((e'^2_n,\ Y'_n))$, and such that, for all $1 < i < j \leq n'$, it is the case that*

$$e_{i-1}^1 \hookrightarrow e_i^1 \implies (e_j^{1\prime},\ Y'_j) \not\sqsubseteq (e_i^{1\prime},\ Y'_i)$$

and

$$e_{i-1}^2 \hookrightarrow e_i^2 \implies (e_j^{2\prime},\ Y'_j) \not\sqsubseteq (e_i^{2\prime},\ Y'_i).$$

Proof By Lemma 11, $S_{\hookrightarrow}$ can be reordered into a paired reduction sequence that first performs all reductions on e_1 until it reaches state n, and then only performs reductions on e_2 afterward. By Lemma 9, we can then reduce both reductions individually to ensure this lemma holds. Note, importantly, that the construction in Lemma 9 never requires changing the constructor (or assignment to bottom) for an expression's concretization. Consequently, the only impact that these two individual changes to reductions might have on each other is that, if an application of FRDC is removed from e_1, a corresponding LKDC applied to e_2 may need to be changed to a FRDC. ■

Corollary 4. *If there exists a paired reduction sequence $S_{\hookrightarrow} = S_1, \ldots, S_n$ where $S_n = (e_n^1, e_n^2, Y_n)$ and $S_A \sqsubseteq (e_n^1,\ Y_n)$ (resp. $S_A \sqsubseteq (e_n^2,\ Y_n)$), then there exists some (possibly infinite) paired reduction sequence $S'_{\hookrightarrow} = S'_1 = S_1, S'_2, \ldots, S'_{n'}$, where $S_A \sqsubseteq (e_{n'}^1,\ Y_{n'})$ (resp. $S_A \sqsubseteq (e_{n'}^2,\ Y_{n'})$), such that for all $1 \leq i < j \leq n'$ it is the case that*

$$e_{i-1}^1 \hookrightarrow e_i^1 \implies (e_j^{1\prime},\ Y'_j) \not\sqsubseteq (e_i^{1\prime},\ Y'_i)$$

and

$$e_{i-1}^2 \hookrightarrow e_i^2 \implies (e_j^{2\prime},\ Y'_j) \not\sqsubseteq (e_i^{2\prime},\ Y'_i).$$

Lemma 14. *Lemma 10 can be applied to the reduction of both the first and second state in a (possibly infinite) reduction sequence.*

Proof The proof follows from the proof of Lemmas 10 and 12 and resembles the proof of Lemma 13. ■