Proof Relevant Corecursive Resolution

Peng Fu[1⋆], Ekaterina Komendantskaya[1], Tom Schrijvers[2], Andrew Pond[1⋆⋆]

[1] Computer Science, University of Dundee
[2] Department of Computer Science, KU Leuven

Abstract. Resolution lies at the foundation of both logic programming and type class context reduction in functional languages. Terminating derivations by resolution have well-defined inductive meaning, whereas some non-terminating derivations can be understood coinductively. Cycle detection is a popular method to capture a small subset of such derivations. We show that in fact cycle detection is a restricted form of coinductive proof, in which the atomic formula forming the cycle plays the rôle of coinductive hypothesis.
This paper introduces a heuristic method for obtaining richer coinductive hypotheses in the form of Horn formulas. Our approach subsumes cycle detection and gives coinductive meaning to a larger class of derivations. For this purpose we extend resolution with Horn formula resolvents and corecursive evidence generation. We illustrate our method on non-terminating type class resolution problems.
Keywords: Horn Clause Logic, Resolution, Corecursion, Haskell Type Class Inference, Coinductive Proofs.

1 Introduction

Horn clause logic is a fragment of first-order logic known for its simple syntax, well-defined models, and efficient algorithms for automated proof search. It is used in a variety of applications, from program verification [3] to type inference in object-oriented programming languages [1]. Similar syntax and proof methods underlie type class inference in functional programming languages [24,17]. For example, the following declaration specifies equality class instances for pairs and integers in Haskell:

instance $Eq\ Int$ **where** ...
instance $(Eq\ x, Eq\ y) \Rightarrow Eq\ (x, y)$ **where** ...

It corresponds to a Horn clause program Φ_{Pair} with two clauses κ_{Int} and κ_{Pair}:

$\kappa_{Int} : Eq\ Int$
$\kappa_{Pair} : (Eq\ x, Eq\ y) \Rightarrow Eq\ (x, y)$

Horn clause logic uses SLD-resolution as an inference engine. If a derivation for a given formula A and a Horn clause program Φ terminates successfully with

⋆ This author is supported by EPSRC grant EP/K031864/1.
⋆⋆ This author is supported by Carnegie Trust Scotland.

substitution θ, then θA is logically entailed by Φ, or $\Phi \vdash \theta A$. The search for a suitable θ reflects the problem-solving nature of SLD-resolution. When the unification algorithm underlying SLD-resolution is restricted to matching, resolution can be viewed as theorem proving: the successful terminating derivations for A using Φ will guarantee $\Phi \vdash A$. For example, $Eq\ (Int, Int) \rightsquigarrow Eq\ Int, Eq\ Int \rightsquigarrow Eq\ Int \rightsquigarrow \emptyset$. Therefore, we have: $\Phi_{Pair} \vdash Eq\ (Int, Int)$. For the purposes of this paper, we always assume resolution by term-matching.

To emphasize the proof-theoretic meaning of resolution, we will record proof evidence alongside the derivation steps. For instance, $Eq\ (Int, Int)$ is proven by applying the clauses κ_{Pair} and κ_{Int}. We denote this by $\Phi_{Pair} \vdash Eq\ (Int, Int) \Downarrow \kappa_{Pair}\ \kappa_{Int}\ \kappa_{Int}$.

Horn clause logic can have inductive and coinductive interpretation, via the least and greatest fixed points of the *consequence operator* F_{Φ}. Given a Horn clause program Φ, and a set S containing (ground) formulas formed from the signature of Φ, $F_{\Phi}(S) = \{\sigma A \mid \sigma B_1, \ldots, \sigma B_n \in S \text{ and } B_1, \ldots B_n \Rightarrow A \in \Phi\}$ [18]. Through the Knaster-Tarski construction, the least fixed point of this operator gives the set of all finite ground formulas *inductively entailed* by Φ. Extending S to include infinite terms, the greatest fixed point of F_{Φ} defines the set of all finite and infinite ground formulas *coinductively entailed* by Φ.

Inductively, SLD-resolution is sound: if $\Phi \vdash A$, then A is inductively entailed by Φ. It is more difficult to characterise coinductive entailment computationally; several approaches exist [22,18,16]. So far the most popular solution is to use cycle detection [22]: given a Horn clause program Φ, if a cycle is found in a derivation for a formula A, then A is coinductively entailed by Φ.

Consider, as an example, the following Horn clause program Φ_{AB}:

$\kappa_A : B\ x \Rightarrow A\ x$
$\kappa_B : A\ x \Rightarrow B\ x$

It gives rise to an infinite derivation $A\ x \rightsquigarrow B\ x \rightsquigarrow A\ x \rightsquigarrow \ldots$. By noticing the cycle, we can conclude that (an instance) of $A\ x$ is coinductively entailed by Φ_{AB}. We can *construct* a proof evidence that reflects the circular nature of this derivation: $\alpha = \kappa_A\ (\kappa_B\ \alpha)$. This being a recursive equation expecting the greatest fixed point solution, we can represent it with the greatest fix point ν operator, $\nu\alpha.\kappa_A\ (\kappa_B\ \alpha)$. Now we have $\Phi_{AB} \vdash A\ x \Downarrow \nu\alpha.\kappa_A\ (\kappa_B\ \alpha)$. From now on, we call the evidence containing ν-term a *corecursive evidence.*

According to Gibbons and Hutton [7] and inspired by Moss and Danner [20], a corecursive program is defined to be a function whose range is a type defined recursively as the greatest solution of some equation (i.e. whose range is a coinductive type). We can informally understand the Horn clause Φ_{AB} as the following Haskell data type declarations:

data $B\ x = K_B\ (A\ x)$
data $A\ x = K_A\ (B\ x)$

So the corecursive evidence $\nu\alpha.\kappa_A\ (\kappa_B\ \alpha)$ for $A\ x$ corresponds to the corecursive program $(d\ :: A\ x) =\ K_A\ (K_B\ d)$. In our case, the corecursive evidence d is that function, and its range type $A\ x$ can be seen as a coinductive type.

Corecursion also arises in type class inference. Consider the following mutually recursive definitions of lists of even and odd length in Haskell:

$$\begin{aligned}
&\mathbf{data}\ \mathit{OddList}\ a\ \ = \mathit{OCons}\ a\ (\mathit{EvenList}\ a)\\
&\mathbf{data}\ \mathit{EvenList}\ a = \mathit{Nil} \mid \mathit{ECons}\ a\ (\mathit{OddList}\ a)
\end{aligned}$$

They give rise to Eq type class instance declarations that can be expressed using the following Horn clause program $\Phi_{EvenOdd}$:

$$\begin{aligned}
&\kappa_{Odd} : (Eq\ a, Eq\ (\mathit{EvenList}\ a)) \Rightarrow Eq\ (\mathit{OddList}\ a)\\
&\kappa_{Even} : (Eq\ a, Eq\ (\mathit{OddList}\ a)) \Rightarrow Eq\ (\mathit{EvenList}\ a)
\end{aligned}$$

When resolving the type class constraint $Eq\ (\mathit{OddList}\ \mathit{Int})$, Haskell's standard type class resolution diverges. The state-of-the-art is to use cycle detection [17] to terminate otherwise infinite derivations. Resolution for $Eq\ (\mathit{OddList}\ \mathit{Int})$ exhibits a cycle on the atomic formula $Eq\ (\mathit{OddList}\ \mathit{Int})$, thus the derivation can be terminated, with corecursive evidence $\nu\alpha.\kappa_{Odd}\ \kappa_{Int}\ (\kappa_{Even}\ \kappa_{Int}\ \alpha)$.

The method of cycle detection is rather limited: there are many Horn clause programs that have coinductive meaning, but do not give rise to detectable cycles. For example, consider the program Φ_Q:

$$\begin{aligned}
&\kappa_S : (Q\ (S\ (G\ x)), Q\ x) \Rightarrow Q\ (S\ x)\\
&\kappa_G : Q\ x \Rightarrow Q\ (G\ x)\\
&\kappa_Z : Q\ Z
\end{aligned}$$

It gives rise to the following derivation without cycling:
$\underline{Q\ (S\ Z)} \leadsto Q\ Z, \underline{Q\ (S\ (G\ Z))} \leadsto Q\ Z, Q\ (G\ Z), \underline{Q\ (S\ (G\ (G\ Z)))} \leadsto \ldots$. When such derivations arise, we cannot terminate the derivation by cycle detection.

Let us look at a similar situation for type classes. Consider a datatype-generic representation of perfect trees: a nested datatype [2], with fixpoint Mu of the higher-order functor HPTree [12].

$$\begin{aligned}
&\mathbf{data}\ \mathit{Mu}\ h\ a = \mathit{In}\ \{\mathit{out} :: h\ (\mathit{Mu}\ h)\ a\}\\
&\mathbf{data}\ \mathit{HPTree}\ f\ a = \mathit{HPLeaf}\ a \mid \mathit{HPNode}\ (f\ (a, a))
\end{aligned}$$

These two datatypes give rise to the following Eq type class instances.

$$\begin{aligned}
&\mathbf{instance}\ Eq\ (h\ (\mathit{Mu}\ h)\ a) \Rightarrow Eq\ (\mathit{Mu}\ h\ a)\ \mathbf{where}\\
&\quad \mathit{In}\ x \equiv \mathit{In}\ y = x \equiv y\\
&\mathbf{instance}\ (Eq\ a, Eq\ (f\ (a, a))) \Rightarrow Eq\ (\mathit{HPTree}\ f\ a)\ \mathbf{where}\\
&\quad \mathit{HPLeaf}\ x \ \ \equiv \mathit{HPLeaf}\ y \ \ = x \equiv y\\
&\quad \mathit{HPNode}\ xs \equiv \mathit{HPNode}\ ys = xs \equiv ys\\
&\quad \_ \qquad\qquad\ \ \equiv \_ \qquad\qquad\ \ = \mathit{False}
\end{aligned}$$

The corresponding Horn clause program Φ_{HPTree} consists of Φ_{Pair} and the following two clauses :

$$\begin{aligned}
&\kappa_{Mu} : Eq\ (h\ (\mathit{Mu}\ h)\ a) \Rightarrow Eq\ (\mathit{Mu}\ h\ a)\\
&\kappa_{HPTree} : (Eq\ a, Eq\ (f\ (a, a))) \Rightarrow Eq\ (\mathit{HPTree}\ f\ a)
\end{aligned}$$

The type class resolution for $Eq\ (Mu\ HPTree\ Int)$ cannot be terminated by cycle detection. Instead we get a context reduction overflow error in the Glasgow Haskell Compiler, even if we just compare two finite data structures of the type $Mu\ HPTree\ Int$.

To find a solution to the above problems, let us view infinite resolution from the perspective of coinductive proof in the Calculus of Coinductive Constructions [4,8]. There, in order to prove a proposition F from the assumptions $F_1, .., F_n$, the proof may involve not only natural deduction and lemmas, but also F, provided the use of F is *guarded*. We could say that the existing cycle detection methods treat the atomic formula forming a cycle as a *coinductive hypothesis*. We can equivalently describe the above-explained derivation for Φ_{AB} in the following terms: when a cycle with a formula $A\ x$ is found in the derivation, Φ_{AB} gets extended with a coinductive hypothesis $\alpha :\ A\ x$. So to prove $A\ x$ coinductively, we would need to apply the clause κ_A first, and then clause κ_B, finally apply the coinductive hypothesis. The resulting proof witness is $\nu\alpha.\ \kappa_A\ (\kappa_B\ \alpha)$.

The next logical step we can make is to use the above formalism to extend the syntax of the coinductive hypotheses. While cycle detection only uses atomic formulas as coinductive hypotheses, we can try to generalise the syntax of coinductive hypotheses to full Horn formulas.

For example, for program Φ_Q, we could prove a lemma $e : Q\ x \Rightarrow Q\ (S\ x)$ coinductively, which would allow us to form finite derivation for $Q\ (S\ Z)$, which is described by $(e\ \kappa_Z)$. The proof of $e : Q\ x \Rightarrow Q\ (S\ x)$ is of a coinductive nature: if we first assume $\alpha : Q\ x \Rightarrow Q\ (S\ x)$ and $\alpha_1 : Q\ C$, then all we need to show is $Q\ (S\ C)$.[3] To show $Q\ (S\ C)$, we apply κ_S, which gives us $Q\ C$, $Q\ (S\ (G\ C))$. We first discharge $Q\ C$ with α_1 and then apply the coinductive hypothesis α which yields $Q\ (G\ C)$, and can be proved with κ_G and α_1. So we have obtained a coinductive proof for e, which is $\nu\alpha.\lambda\alpha_1.\kappa_S\ (\alpha\ (\kappa_G\ \alpha_1))\ \alpha_1$. We can apply similar reasoning to show that $\Phi_{HPTree} \vdash Eq\ (Mu\ HPTree\ Int) \Downarrow (\nu\alpha.\lambda\alpha_1.\kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (\alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1))))\ \kappa_{Int}$ using the coinductively proved lemma $Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ [4].

To formalise the above intuitions, we need to solve several technical problems.

1. How to generate suitable lemmas? We propose to observe a more general notion of a loop invariant than a cycle in the non-terminating resolution. In Section 3 we devise a heuristic method to identify potential loops in the resolution tree and extract *candidate lemmas* in Horn clause form.
In general, it is very challenging to develop a practical method for generating candidate lemmas based on loop analysis, since the admissibility of a loop in reduction is a semi-decidable problem [25].

2. How to enrich resolution to allow coinductive proofs for Horn formulas? and how to formalise the corecursive proof evidence construction? Coinductive proofs involve not only applying the axioms, but also modus ponens and generalization. Therefore, the resolution mechanism will have to be extended in order to support such automation.

[3] Note that here C is an eigenvariable.
[4] The proof term can be type-checked with polymorphic recursion.

In Section 4, we introduce proof relevant *corecursive resolution* – a calculus that extends the standard resolution rule with two further rules: one allows us to resolve Horn formula queries, and the other to construct corecursive proof evidence for non-terminating resolution.

3. How to give an operational semantics to the evidence produced by corecursive resolution of Section 4? In particular, we need to show the correspondence between corecursive evidence and resolution seen as infinite reduction. In Section 5, we prove that for every non-terminating resolution resulting from a *simple loop*, a coinductively provable candidate lemma can be obtained and its evidence is *observationally equivalent* to the non-terminating resolution process.
In type class inference, the proof evidence has computational meaning, i.e. the evidence will be run as a program. So the corecursive evidence should be able to recover the original infinite resolution trace.

In Sections 6 and 7 we survey the related work, explain the limitations of our method and conclude the paper. We have implemented our method of candidate lemma generation based on loop analysis and corecursive resolution, and incorporated it in the type class inference process of a simple functional language. Additional examples and implementation information are provided in the extended version.

2 Preliminaries: Resolution with Evidence

This section provides a standard formalisation of resolution with evidence together with two derived forms: a *small-step* variant of resolution and a reification of resolution in a resolution tree.

We consider the following syntax.

Definition 1 (Basic syntax).

Term	t	$::=$	$x \mid K \mid t\ t'$
Atomic Formula	A, B, C, D	$::=$	$P\ t_1\ ...\ t_n$
Horn Formula	H	$::=$	$B_1, ..., B_n \Rightarrow A$
Proof/Evidence	e	$::=$	$\kappa \mid e\ e'$
Axiom Environment	Φ	$::=$	$\cdot \mid \Phi, (\kappa : H)$

We consider first-order applicative terms, where K stands for some constant symbol. Atomic formulas are predicates on terms, and Horn formulas are defined as usual. We assume that all variables x in Horn formulas are implicitly universally quantified. There are no *existential variables* in the Horn formulas, i.e., $\bigcup_i \mathrm{FV}(B_i) \subseteq \mathrm{FV}(A)$ for $B_1, \ldots, B_n \Rightarrow A$. The axiom environment Φ is a set of Horn formulas labelled with distinct evidence constants κ. Evidence terms e are made of evidence constants κ and their applications. Finally, we often use $\underline{A}$ to abbreviate $A_1, ..., A_n$ when the number n is unimportant.

The above syntax can be used to model the Haskell type class setting as follows. Terms denote Haskell types like *Int* or (x, y), and atomic formulas denote Haskell type class constraints on types like *Eq* $(\mathit{Int}, \mathit{Int})$. Horn formulas correspond to the type-level information of type class instances.

Our evidence e models type class dictionaries, following Wadler and Blott's dictionary-passing elaboration of type classes [24]. In particular the constants κ refer to dictionaries that capture the term-level information of type class instances, i.e., the implementations of the type class methods. Evidence application $(e\ e')$ accounts for dictionaries that are parametrised by other dictionaries. Horn formulas in turn represent type class instance declarations. The axiom environment Φ corresponds to Haskell's global environment of type class instances. Note that the treatment of type class instance declaration and their corresponding evidence construction here are based on our own understanding of many related works ([15,14,23]), which are also discussed in Section 6.

In order to define resolution together with evidence generation, we use resolution judgement $\Phi \vdash A \Downarrow e$ to state that the atomic formula A is entailed by the axioms Φ, and that the proof term e witnesses this entailment. It is defined by means of the following inference rule.

Definition 2 (Resolution). $\boxed{\Phi \vdash A \Downarrow e}$

$$\frac{\Phi \vdash \sigma B_1 \Downarrow e_1 \quad \cdots \quad \Phi \vdash \sigma B_n \Downarrow e_n}{\Phi \vdash \sigma A \Downarrow \kappa\ e_1 \cdots e_n}\ \textit{if}\ (\kappa : B_1, ..., B_n \Rightarrow A) \in \Phi$$

Using this definition we can show $\Phi_{Pair} \vdash Eq\ (Int, Int) \Downarrow \kappa_{Pair}\ \kappa_{Int}\ \kappa_{Int}$.

In case resolution is diverging, it is often more convenient to consider a *small-step* resolution judgement (in analogy to the small step operational semantics) that performs one resolution step at a time and allows us to observe the intermediate states.

The basic idea is to rewrite the initial query A step by step into its evidence e. This involves *mixed terms* on the way that consist partly of evidence, and partly of formulas that are not yet resolved.

Definition 3 (Mixed Terms).

$$\begin{array}{llll} \textit{Mixed term} & q & ::= & A \mid \kappa \mid q\ q' \\ \textit{Mixed term context} & \mathcal{C} & ::= & \bullet \mid \mathcal{C}\ q \mid q\ \mathcal{C} \end{array}$$

At the same time we have defined mixed term contexts $\mathcal{C}$ as mixed terms with a hole $\bullet$, where $\mathcal{C}[q]$ substitutes the hole with q in the usual way.

Definition 4 (Small-Step Resolution). $\boxed{\Phi \vdash q \rightarrow q'}$

$$\frac{}{\Phi \vdash \mathcal{C}[\sigma A] \rightarrow \mathcal{C}[\kappa\ \sigma \underline{B}]}\ \textit{if}\ (\kappa : \underline{B} \Rightarrow A) \in \Phi$$

For instance, we resolve $Eq\ (Int, Int)$ in three small steps: $\Phi_{Pair} \vdash Eq\ (Int, Int) \rightarrow \kappa_{Pair}\ (Eq\ Int)\ (Eq\ Int)$, $\Phi_{Pair} \vdash \kappa_{Pair}\ (Eq\ Int)\ (Eq\ Int) \rightarrow \kappa_{Pair}\ \kappa_{Int}\ (Eq\ Int)$ and $\Phi_{Pair} \vdash \kappa_{Pair}\ \kappa_{Int}\ (Eq\ Int) \rightarrow \kappa_{Pair}\ \kappa_{Int}\ \kappa_{Int}$. We write $\Phi \vdash q \rightarrow^* q'$ to denote the transitive closure of small-step resolution.

The following theorem formalizes the intuition that resolution and small-step resolution coincide.

Theorem 1. *$\Phi \vdash A \Downarrow e$ iff $\Phi \vdash A \rightarrow^* e$.*

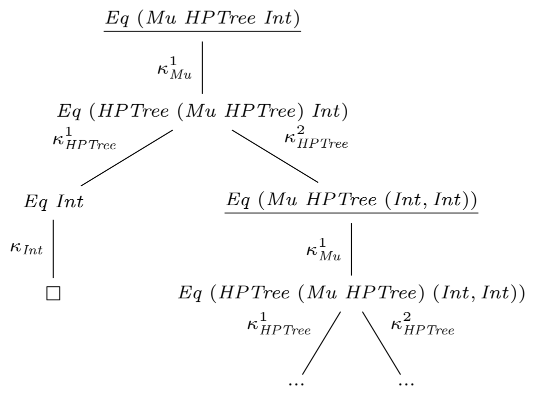

$\underline{Eq\ (Mu\ HPTree\ Int)}$
κ^1_{Mu}
$Eq\ (HPTree\ (Mu\ HPTree)\ Int)$
κ^1_{HPTree}
κ^2_{HPTree}
$Eq\ Int$
$\underline{Eq\ (Mu\ HPTree\ (Int, Int))}$
κ_{Int}
$\square$
κ^1_{Mu}
$Eq\ (HPTree\ (Mu\ HPTree)\ (Int, Int))$
κ^1_{HPTree}
κ^2_{HPTree}
...
...

Fig. 1: The infinite resolution tree for $\Phi_{HPTree} \vdash Eq\ (Mu\ HPTree\ Int) \Uparrow$

The proof tree for a judgement $\Phi \vdash A \Downarrow e$ is called a *resolution tree*. It conveniently records the history of resolution and, for instance, it is easy to observe the ancestors of a node. This last feature is useful for our heuristic loop invariant analysis in Section 3.

Our formalisation of trees in general is as follows: We use w, v to denote positions $\langle k_1, k_2, ..., k_n \rangle$ in a tree, where $k_i \geqslant 1$ for $1 \leqslant i \leqslant n$. Let ϵ denote the empty position or *root*. We also define $\langle k_1, k_2, ..., k_n \rangle \cdot i = \langle k_1, k_2, ..., k_n, i \rangle$ and $\langle k_1, k_2, ..., k_n \rangle + \langle l_1, ..., l_m \rangle = \langle k_1, k_2, ..., k_n, l_1, ..., l_m \rangle$. We write $w > v$ if there exists a non-empty v' such that $w = v + v'$. For a tree T, $T(w)$ refers to the node at position w, and $T(w, i)$ refers to the edge between $T(w)$ and $T(w \cdot i)$. We use $\square$ as a special proposition to denote success.

Resolution trees are defined as follows, note that they are a special case of *rewriting trees* [13,16]:

Definition 5 (Resolution Tree). *The resolution tree for atomic formula A is a tree T satisfying:*

- $T(\epsilon) = A$.
- $T(w \cdot i) = \sigma B_i$ *and* $T(w, i) = \kappa^i$ *with* $i \in \{1, ..., n\}$ *if* $T(w) = \sigma D$ *and* $(\kappa : B_1, ..., B_n \Rightarrow D) \in \Phi$. *When* $n = 0$, *we write* $T(w \cdot i) = \square$ *and* $T(w, i) = \kappa$ *for any* $i > 0$.

In general, the resolution tree can be infinite, this means that resolution is non-terminating, which we denote as $\Phi \vdash A \Uparrow$. Figure 1 shows a finite fragment of the infinite resolution tree for $\Phi_{HPTree} \vdash Eq\ (Mu\ HPTree\ Int) \Uparrow$.

We note that Definitions 2 and 4 describe a special case of SLD-resolution in which unification taking place in derivations is restricted to term-matching. This restriction is motivated by two considerations. The first one comes directly from the application area of our results: type class resolution uses exactly this restricted version of SLD-resolution. The second reason is of more general nature. As discussed in detail in [6,16], SLD-derivations restricted to term-matching reflect the *theorem proving* content of a proof by SLD-resolution. That is, if A can be derived from Φ by SLD-resolution with term-matching only, then A is inductvely entailed by Φ. If, on the other hand, A is derived from Φ by SLD-resolution with unification and computes a substitution σ, then σA is inductively entailed by Φ. In this sense, SLD-resolution with unification additionally has a *problem-solving* aspect. In developing proof-theoretic approach to resolution here, we thus focus on resolution by term-matching.

The resolution rule of Definition 2 resembles the definition of the consequence operator [18] used to define declarative semantics of Horn clause Logic. In fact, the forward and backward closure of the rule of Definition 2 can be directly used to construct the usual least and greatest Herbrand models for Horn clause logic, as shown in [16]. There, it was also shown that SLD-resolution by term-matching is sound but incomplete relative to the least Herbrand models.

3 Candidate Lemma Generation

This section explains how we generate candidate lemma from a potentially infinite resolution tree. Based on Paterson's condition we obtain a finite pruned approximation (Definition 8) of this resolution tree. Anti-unification on this approximation yields an abstract atomic formula and the corresponding abstract approximated resolution tree. It is from this abstract tree that we read off the candidate lemma (Definition 11).

We use $\Sigma(A)$ and $\mathrm{FVar}(A)$ to denote the multi-sets of respectively function symbols and variables in A.

Definition 6 (Paterson's Condition). *For* $(\kappa : \underline{B} \Rightarrow A) \in \Phi$*, we say* κ *satisfies Paterson's condition if* $(\Sigma(B_i) \cup \mathrm{FVar}(B_i)) \subset (\Sigma(A) \cup \mathrm{FVar}(A))$ *for each* B_i*.*

Paterson's condition is used in Glasgow Haskell Compiler to enforce termination of context reduction [23]. In this paper, we use it as a practical criterion to detect problematic instance declarations. Any declarations that do not satisfy the condition could potentially introduce diverging behavior in the resolution tree.

If $\kappa : A_1, ..., A_n \Rightarrow B$, then we have $\kappa^i : A_i \Rightarrow B$ for projection on index i.

Definition 7 (Critical Triple). *Let* $v = (w \cdot i) + v'$ *for some* v'*. A critical triple in* T *is a triple* $\langle \kappa^i, T(w), T(v) \rangle$ *such that* $T(v, i) = T(w, i) = \kappa^i$*, and* κ^i *does not satisfy Paterson's condition.*

We will omit κ^i from the triple and write $\langle T(w), T(v) \rangle$ when it is not important. Intuitively, it means the nodes $T(w)$ and $T(v)$ are using the same problematic projection κ^i, which could give rise to infinite resolution.

The absence of a critical triple in a resolution tree means that it has to be finite [23], while the presence of a critical triple only means that the tree is *possibly* infinite. In general the infiniteness of a resolution tree is undecidable and the critical triples provide a convenient over-approximation.

Definition 8 (Closed Subtree). *A closed subtree T is a subtree of a resolution tree such that for all leaves $T(v) \neq \Box$, the root $T(\epsilon)$ and $T(v)$ form a critical triple.*

The critical triple in Figure 1 is formed by the underlined nodes. The closed subtree in that figure is the subtree without the infinite branch below node $Eq\ (Mu\ HPTree\ (Int, Int))$. A closed subtree can intuitively be understood as a finite approximation of an infinite resolution tree. We use it as the basis for generating candidate lemma by means of anti-unification [21].

Definition 9 (Anti-Unifier). *We define the least general anti-unifier of atomic formulas A and B (denoted by $A \sqcup B$) and the least general anti-unifier of the terms t and t' (denoted by $t \sqcup t'$) as:*

- $P\ t_1\ ..., t_n\ \sqcup P\ t'_1\ ..., t'_n = P\ (t_1 \sqcup t'_1)\ ...\ (t_n \sqcup t'_n)$
- $K\ t_1\ ...\ t_n \sqcup K\ t'_1\ ...\ t'_n = K\ (t_1 \sqcup t'_1)\ ...\ (t_n \sqcup t'_n)$
- *Otherwise,* $A \sqcup B = \phi(A, B), t \sqcup t' = \phi(t, t')$*, where ϕ is an injective function from a pair of terms (atomic formulas) to a set of fresh variables.*

Anti-unification allows us to extract the common pattern from different ground atomic formulas.

Definition 10 (Abstract Representation). *Let $\langle T(\epsilon), T(v_1)\rangle, ..., \langle T(\epsilon), T(v_n)\rangle$ be all the critical triples in a closed subtree T. Let $C = T(\epsilon) \sqcup T(v_1) \sqcup ... \sqcup T(v_n)$, then the abstract representation T' of the closed subtree T is a tree such that:*

- $T'(\epsilon) = C$
- $T'(w \cdot i) = \sigma B_i$ *and* $T'(w, i) = \kappa^i$ *with* $i \in \{1, ..., n\}$ *if* $T'(w) = \sigma D$ *and* $(\kappa : B_1, ..., B_n \Rightarrow D) \in \Phi$*. When $n = 0$, we write $T'(w \cdot i) = \Box$ and $T'(w, i) = \kappa$ for any $i > 0$.*
- $T'(w)$ *is undefined if* $w > v_i$ *for some* $1 \leqslant i \leqslant n$*.*

The abstract representation unfolds the anti-unifier of all the critical triples. Thus the abstract representation can always be embedded into the original closed subtree. It is an abstract form of the closed subtree, and we can extract the candidate lemma from the abstract representation.

Definition 11 (Candidate Lemma). *Let T be an abstract representation of a closed subtree, then the candidate lemma induced by this abstract representation is $T(v_1), ..., T(v_n) \Rightarrow T(\epsilon)$, where the $T(v_i)$ are all the leaves for which $T(v_i) \Rightarrow T(\epsilon)$ satisfies Paterson's condition.*

Figure 2 shows the abstract representation of the closed subtree of Figure 1. We read off the candidate lemma as $Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$.

The candidate lemma plays a double role. Firstly, it allows us to construct a finite resolution tree. For example, we know that $Eq\ (Mu\ HPTree\ Int)$ gives rise to infinite tree with the axiom environment Φ_{HPTree}. However, a finite tree can be constructed with $Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$, since it reduces

$$\begin{tikzcd}
& \underline{Eq\ (Mu\ HPTree\ x)} \arrow[d, dash, "\kappa^1_{Mu}"'] & \\
& Eq\ (HPTree\ (Mu\ HPTree)\ x) \arrow[dl, dash, "\kappa^1_{HPTree}"'] \arrow[dr, dash, "\kappa^2_{HPTree}"] & \\
Eq\ x & & \underline{Eq\ (Mu\ HPTree\ (x,x))}
\end{tikzcd}$$

Fig. 2: The abstract representation of the closed subtree of Figure 1

$Eq\ (Mu\ HPTree\ Int)$ to $Eq\ Int$, which succeeds trivially with κ_{Int}. Next we show how to prove the candidate lemma coinductively, and such proofs will encapsulate the infinite aspect of the resolution tree. Since an infinite resolution tree gives rise to infinite evidence, the finite proof of the lemma has to be coinductive. We discuss such evidence construction in detail in Section 4 and Section 5.

4 Corecursive Resolution

In this section, we extend the definition of resolution from Section 2 by introducing two additional rules: one to handle coinductive proofs, and another – to allow Horn formula goals, rather than atomic goals, in the derivations. We call the resulting calculus *corecursive resolution.*

Definition 12 (Extended Syntax).

$$\begin{array}{llcl}
\textit{Proof/Evidence} & e & ::= & \kappa \mid e\ e' \mid \alpha \mid \lambda\alpha.e \mid \nu\alpha.e \\
\textit{Axiom Environment} & \Phi & ::= & \cdot \mid \Phi, (e : H)
\end{array}$$

To support coinductive proofs, we extend the syntax of evidence with functions $\lambda\alpha.e$, variables α and fixed point $\nu\alpha.e$ (which models the recursive equation $\alpha = e$ expecting the greatest solution). Also we allow the Horn clauses H in the axiom environment Φ to be supported by any form of evidence e (and not necessarily by constants κ).

Definition 13 (Corecursive Resolution). *The following judgement for corecursive resolution extends the resolution in Definition 2.*

$$\frac{\Phi \vdash \sigma B_1 \Downarrow e_1 \quad \cdots \quad \Phi \vdash \sigma B_n \Downarrow e_n}{\Phi \vdash \sigma A \Downarrow e\ e_1 \cdots e_n}\ \textit{if}\ (e : B_1, ..., B_m \Rightarrow A) \in \Phi$$

$$\frac{\Phi, (\alpha : \underline{A} \Rightarrow B) \vdash \underline{A} \Rightarrow B \Downarrow e \quad \mathrm{HNF}(e)}{\Phi \vdash \underline{A} \Rightarrow B \Downarrow \nu\alpha.e}\ (\textsc{Mu}) \qquad \frac{\Phi, (\underline{\alpha} : \underline{A}) \vdash B \Downarrow e}{\Phi \vdash \underline{A} \Rightarrow B \Downarrow \lambda\underline{\alpha}.e}\ (\textsc{Lam})$$

Note that $\mathrm{HNF}(e)$ means e has to be in *head normal form* $\lambda\underline{\alpha}.\kappa\ \underline{e}$. This requirement is essential to ensure the corecursive evidence satisfies the *guardedness* condition.[5] The Lam rule implicitly assumes the treatment of *eigenvariables*, i.e. we instantiate all the free variables in $\underline{A} \Rightarrow B$ with fresh term constants.

We implicitly assume that axiom environments are well-formed.

Definition 14 (Well-formedness of environment).

$$\frac{}{\cdot \vdash \mathsf{wf}} \qquad \frac{\Phi \vdash \mathsf{wf}}{\Phi, \alpha : H \vdash \mathsf{wf}} \qquad \frac{\Phi \vdash \mathsf{wf}}{\Phi, \kappa : H \vdash \mathsf{wf}} \qquad \frac{\Phi \vdash H \Downarrow e}{\Phi, e : H \vdash \mathsf{wf}}$$

As an example, let us consider resolving the candidate lemma $Eq\ x \Rightarrow Eq\ (My\ HPTree\ x)$ against the axiom environment Φ_{HPTree}. This yields the following derivation, where $\Phi_1 = \Phi_{HPTree}, (\alpha : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x))$ and $\Phi_2 = \Phi_1, (\alpha_1 : Eq\ C)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi_2 \vdash Eq\ C \Downarrow \alpha_1} \quad \dfrac{\dfrac{\dfrac{}{\Phi_2 \vdash Eq\ C \Downarrow \alpha_1} \quad \dfrac{}{\Phi_2 \vdash Eq\ C \Downarrow \alpha_1}}{\Phi_2 \vdash Eq\ (C, C) \Downarrow (\kappa_{Pair}\ \alpha_1\ \alpha_1)}}{\Phi_2 \vdash Eq\ (HPTree\ (C, C)) \Downarrow \alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1)}}{\Phi_2 \vdash Eq\ (HPTree\ (Mu\ HPTree\ C)) \Downarrow \kappa_{HPTree}\ \alpha_1\ (\alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1))}}{(\Phi_2 = \Phi_1, \alpha_1 : Eq\ C) \vdash Eq\ (Mu\ HPTree\ C) \Downarrow \kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (\alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1)))}}{\Phi_1 \vdash Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x) \Downarrow \lambda\alpha_1.\kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (\alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1)))}}{\Phi_{HPTree} \vdash Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x) \Downarrow \nu\alpha.\lambda\alpha_1.\kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (\alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1)))}$$

Once we prove $Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x)$ from Φ_{HPTree} by corecursive resolution, we can add it to the axiom environment and use it to prove the ground query $Eq\ (Mu\ HPTree\ Int)$. Let $\Phi' = \Phi_{HPTree}, (\nu\alpha.\lambda\alpha_1.\kappa_1\ (\kappa_2\ \alpha_1\ (\alpha\ (\kappa_3\ \alpha_1\ \alpha_1))) : Eq\ x \Rightarrow Eq\ (Mu\ HPTree\ x))$. We have the following derivation.

$$\frac{\Phi' \vdash Eq\ Int \Downarrow \kappa_{Int}}{\Phi' \vdash Eq\ (Mu\ HPTree\ Int) \Downarrow (\nu\alpha.\lambda\alpha_1.\kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (\alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1))))\ \kappa_{Int}}$$

5 Operational Semantics of Corecursive Evidence

The purpose of this section is to give operational semantics to corecursive resolution, and in particular, we are interested in giving operational interpretation to the corecursive evidence constructed as a result of applying corecursive resolution. In type class applications, for example, the evidence constructed for a query will be run as a program. It is therefore important to establish the exact relationship between the non-terminating resolution as a process and the proof-term that we obtain via corecursive resolution. We prove that corecursive evidence indeed faithfully captures the otherwise infinite resolution process of Section 2.

[5] See the extended version for a detailed discussion.

In general, we know that if $\Phi \vdash A \rightarrow^* \mathcal{C}[\sigma A]$, then we can observe the following looping infinite reduction trace:

$$\Phi \vdash A \rightarrow^* \mathcal{C}[\sigma A] \rightarrow^* \mathcal{C}[\sigma\mathcal{C}[\sigma^2 A]] \rightarrow^* \mathcal{C}[\sigma\mathcal{C}[\sigma^2\mathcal{C}[\sigma^3 A]]] \rightarrow ...$$

Each iteration of the loop gives rise to repeatedly applying substitution σ to the reduction context $\mathcal{C}$.

In principle, this mixed term context $\mathcal{C}$ may contain an atomic formula B that itself is normalizing, but σB spawns another loop. Clearly this is a complicating factor. For instance, a loop can spawn off additional loops in each iteration. Alternatively, a loop can have multiple iteration points such as $\Phi \vdash A \rightarrow^* \mathcal{C}[\sigma_1 A, \sigma_2 A, ..., \sigma_n A]$.[6] These complicating factors are beyond the scope of this section. We focus only on *simple loops*. These are loops with a single iteration point that does not spawn additional loops.

We use $|\mathcal{C}|$ to denote the set of atomic formulas in the context $\mathcal{C}$. If all atomic formulas $D \in |\mathcal{C}|$ are irreducible with respect to Φ, then we call $\mathcal{C}$ a *normal context.*

Definition 15 (Simple Loop). *Let $\Phi \vdash B \rightarrow^* \mathcal{C}[\sigma B]$, where $\mathcal{C}$ is normal. If for all $D \in |\mathcal{C}|$, we have that $\Phi \vdash \sigma D \rightarrow^* \mathcal{C}'[D]$ with $|\mathcal{C}'| = \emptyset$, then we call $\Phi \vdash B \rightarrow^* \mathcal{C}[\sigma B]$ a simple loop.*

In the above definition, the normality of $\mathcal{C}$ ensures that the loop has a single iteration point. Likewise the condition $\Phi \vdash \sigma D \rightarrow^* \mathcal{C}'[D]$, which implies that $\Phi \vdash \sigma^n D \rightarrow^* \mathcal{C}'^n[D]$, guarantees that each iteration of the loop spawns no further loops.

Definition 16 (Observational Point). *Let $\Phi \vdash B \rightarrow^* \mathcal{C}'[\sigma B]$ be a simple loop and $\Phi \vdash B \rightarrow^* q$. We call q an observational point if it is of the form $\mathcal{C}[\delta B]$. We use $\mathcal{O}(B)_\Phi$ to denote the set of observational points in the simple loop.*

For example, we have the following infinite resolution trace generated by the simple loop (with the subterms of observational points underlined).

$$\begin{gathered}
\Phi_{HPTree} \vdash \underline{Eq\ (Mu\ HPTree\ x)} \rightarrow \kappa_{Mu}\ (Eq\ (HPTree\ (Mu\ HPTree)\ x)) \rightarrow \\
\kappa_{Mu}\ (\kappa_{HPTree}\ (Eq\ x)\ \underline{(Eq\ (Mu\ HPTree\ (x,x)))}) \rightarrow \\
\kappa_{Mu}\ (\kappa_{HPTree}\ (Eq\ x)\ (\kappa_{Mu}\ (Eq\ (HPTree\ (Mu\ HPTree)\ (x,x))))) \rightarrow \\
\kappa_{Mu}(\kappa_{HPTree}(Eq\ x)(\kappa_{Mu}(\kappa_{HPTree}(Eq(x,x))(Eq(Mu\ HPTree\ ((x,x),(x,x))))))) \rightarrow \\
\kappa_{Mu}(\kappa_{HPTree}(Eq\ x)(\kappa_{Mu}(\kappa_{HPTree}(\kappa_{Pair}(Eq\ x)(Eq\ x)))\underline{(Eq\ (Mu\ HPTree\ ((x,x),(x,x))))}))) \rightarrow \ldots
\end{gathered}$$

In this case, we have $\sigma = [(x,x)/x]$ and $\Phi \vdash \sigma(Eq\ x) \rightarrow \kappa_{Pair}\ (Eq\ x)\ (Eq\ x)$.

The corecursive evidence encapsulates an infinite derivation in a finite fix-point expression. We can recover the infinite resolution by reducing the corecursive expression. To define small-step *evidence reduction*, we first extend mixed terms to cope with richer corecursive evidence.

Definition 17. *Mixed term* $q\ ::=\ A \mid \kappa \mid q\ q' \mid \alpha \mid \lambda\alpha.q \mid \nu\alpha.q$

Now we define the small-step evidence reduction relation $q \rightsquigarrow q'$.

[6] Note that we abuse notation here to denote contexts with multiple holes. Also we abbreviate identical instantiation of $\mathcal{C}[D, \ldots, D]$ those multiple holes to $\mathcal{C}[D]$.

Definition 18 (Small Step Evidence Reduction). $\boxed{q \rightsquigarrow q'}$

$$\frac{}{\mathcal{C}[\nu\alpha.q] \rightsquigarrow_\nu \mathcal{C}[[\nu\alpha.q/\alpha]q]} \qquad \frac{}{\mathcal{C}[(\lambda\alpha.q)\ q'] \rightsquigarrow_\beta \mathcal{C}[[q'/\alpha]q]}$$

Note that for simplicity we still use the mixed term context $\mathcal{C}$ as defined in Section 2, but we only allow the reduction of an outermost redex, i.e., a redex that is not a subterm of some other redex. In other words, reduction unfolds the evidence term strictly downwards from the root, this follows closely the way evidence is constructed during resolution.

We call the states where we perform a ν-transition *corecursive points.* Note that ν-transitions unfold a corecursive definition. These correspond closely to the observational points in resolution.

Definition 19 (Corecursive Point). *Let $q' \rightsquigarrow^* q$. We call q a corecursive point if it is of the form $\mathcal{C}[(\nu\alpha.e)\ q_1 \ldots\ q_n]$. We use $\mathcal{S}(q')$ to denote the set of corecursive points in $q' \rightsquigarrow^* q$.*

Let $e \equiv \nu\alpha.\lambda\alpha_1.\kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (\alpha\ (\kappa_{Pair}\ \alpha_1\ \alpha_1)))$. We have the following evidence reduction trace (with the subterms of corecursive points underlined):

$$\begin{gathered}
\underline{e\ (Eq\ x)} \rightsquigarrow_\nu (\lambda\alpha_1.\kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (e\ (\kappa_{Pair}\ \alpha_1\ \alpha_1))))\ (Eq\ x) \rightsquigarrow_\beta \\
\kappa_{Mu}\ (\kappa_{HPTree}\ (Eq\ x)\ \underline{(e\ (\kappa_{Pair}\ (Eq\ x)\ (Eq\ x)))}) \rightsquigarrow_\nu \\
\kappa_{Mu}\ (\kappa_{HPTree}\ (Eq\ x)\ ((\lambda\alpha_1.\kappa_{Mu}\ (\kappa_{HPTree}\ \alpha_1\ (e\ (\kappa_{Pair}\ \alpha_1\ \alpha_1))))\ (\kappa_{Pair}\ (Eq\ x)\ (Eq\ x)))) \rightsquigarrow_\beta \\
\kappa_{Mu}(\kappa_{HPTree}(Eq\ x)(\kappa_{Mu}(\kappa_{HPTree}(\kappa_{Pair}(Eq\ x)(Eq\ x))\underline{(e(\kappa_{Pair}(\kappa_{Pair}(Eq\ x)(Eq\ x))(\kappa_{Pair}(Eq\ x)(Eq\ x))))}))) \\
\rightsquigarrow_\nu \ldots
\end{gathered}$$

Observe that the mixed term contexts of the observational points and the corecursive points in the above traces coincide. This allows us to show observational equivalence of resolution and evidence reduction without explicitly introducing actual infinite evidence.

The following theorem shows that if resolution gives rise to a simple loop, then we can obtain a corecursive evidence e (Theorem 2 (1)) such that the infinite resolution trace is observational equivalent to e's evidence reduction trace (Theorem 2 (2)).

Theorem 2 (Observational Equivalence). *Let $\Phi \vdash B \rightarrow^* \mathcal{C}[\sigma B]$ be a simple loop and $|\mathcal{C}| = \{D_1, \ldots, D_n\}$. Then:*
1. We have $\Phi \vdash D_1, \ldots, D_n \Rightarrow B \Downarrow \nu\alpha.\lambda\alpha_1.\ldots.\lambda\alpha_n.e$ for some e.
2. $\mathcal{C}[\delta B] \in \mathcal{O}(B)_\Phi$ iff $\mathcal{C}[(\nu\alpha.\lambda\underline{\alpha}.e)\ \underline{q}] \in \mathcal{S}((\nu\alpha.\lambda\underline{\alpha}.e)\ \underline{D})$.

The proof can be found in the extended version.

6 Related Work

Calculus of Coinductive Constructions. Interactive theorem prover Coq pioneered implementation of the *guarded coinduction principle* ([4,8]). The Coq termination checker may prevent some nested uses of coinduction, e.g. a proof term such as $(\nu\alpha.\lambda x.\kappa_0\ (\kappa_1\ x\ (\alpha\ (\alpha\ x))))\ \kappa_2$ is not accepted by Coq, while from the outermost reduction point of view, this proof term is productive.

Loop detection in term rewriting. Distinctions between cycle, loop and non-looping has long been established in term rewriting research ([5,25]). For us, detecting loop is the first step of invariant analysis, but we also want to extract corecursive evidence such that it captures the infinite reduction trace.

Non-terminating type-class resolution. Hughes (Section 4 [11]) observed the cyclic nature of the instance declarations **instance** $\mathit{Sat}\ (\mathit{EqD}\ a) \Rightarrow \mathit{Eq}\ a$ and **instance** $\mathit{Eq}\ a \Rightarrow \mathit{Sat}\ (\mathit{EqD}\ a)$. He proposed to treat the looping context reduction as failure, in which case the compiler would need to search for an alternative reduction.

The cycle detection method [17] was proposed to generate corecursive evidence for a restricted class of non-terminating resolution. It is supported by the current Glasgow Haskell Compiler.

Hinze and Peyton Jones [10] came across an example of an instance of the form **instance** $(\mathit{Binary}\ a, \mathit{Binary}\ (f\ (\mathit{GRose}\ f\ a))) \Rightarrow \mathit{Binary}\ (\mathit{GRose}\ f\ a)$, but discovered that it causes resolution to diverge. They suggested the following as a replacement: **instance** $(\mathit{Binary}\ a, \forall b\ .\ \mathit{Binary}\ b \Rightarrow \mathit{Binary}\ f\ b) \Rightarrow \mathit{Binary}\ (\mathit{GRose}\ f\ a)$. Unfortunately, Haskell does not support instances with *polymorphic higher-order instance contexts*. Nevertheless, allowing such implication constraints would greatly increase the expressitivity of corecursive resolution. In the terminology of our paper, it amounts to extending Horn formulas to intuitionistic formulas. Working with intuitionistic formulas would require a certain amount of searching, as the non-overlapping condition for Horn formulas is not enough to ensure uniqueness of the evidence. For example, consider the following axioms:

$$\begin{gathered}\kappa_1 : (A \Rightarrow B\ x) \Rightarrow D\ (S\ x)\\ \kappa_2 : A, D\ x \Rightarrow B\ (S\ x)\\ \kappa_3 : \Rightarrow D\ Z\end{gathered}$$

We have two distinct proof terms for $D\ (S\ (S\ (S\ (S\ Z))))$:

$$\begin{gathered}\kappa_1\ (\lambda\alpha_1.\kappa_2\ \alpha_1\ (\kappa_1\ (\lambda\alpha_2.\kappa_2\ \underline{\alpha_1}\ \kappa_3)))\\ \kappa_1\ (\lambda\alpha_1.\kappa_2\ \alpha_1\ (\kappa_1\ (\lambda\alpha_2.\kappa_2\ \underline{\alpha_2}\ \kappa_3)))\end{gathered}$$

This is undesirable from the perspective of generating evidence for type class.

Instance declarations and (Horn Clause) logic programs. The process of simplifying type class constraints is formally described as the notion of *context reduction* by Peyton Jones et. al. [15]. Section 3.2 of the same paper also describes the form of type class instance declarations. Type class evidence in its connection with type system is studied in Mark Jones's thesis [14, Chapter 4.2]. Instance declarations can also be interpreted as single head *simplification* rules in Constraints Handling Rules (CHR) [23], which implies that instance declarations can be modeled as Horn formulas naturally. To our knowledge, the tradition of studying logic programming proof-theoretically dates back to Girard's suggestion that the cut rule can model resolution for Horn formulas [9, Chapter

13.4]. Alternatively, Miller et. al. [19] model Horn formulas using cut-free sequent calculus. Context reduction, instance declaration and their connection to proof relevant resolution are also discussed under the name of *LP-TM* (logic programming with term-matching) in Fu and Komendantskaya [6, Section 4.1].

7 Conclusion and Future Work

We have introduced a novel approach to non-terminating resolution. Firstly, we have shown that the popular cycle detection methods employed for logic programming or type class resolution can be understood via more general coinductive proof principles ([4,8]). Secondly, we have shown that resolution can be enriched with rules that capture the intuition of richer coinductive hypothesis formation. This extension allows to provide corecursive evidence to some derivations that could not be handled by previous methods. Moreover, corecursive resolution is formulated in a proof-relevant way, i.e. proof-evidence construction is an essential part of corecursive resolution. This makes it easier to integrate it directly into type class inference.

We have implemented the techniques of Sections 3 and 4, and have incorporated them as part of the evidence construction process for a simple language that admits previously non-terminating examples.[7]

Future Work In general, the interactions between different loops can be complicated. Consider Φ_{Pair} with the following declarations (denoted by Φ_M):

$$\kappa_M : Eq\ (h_1\ (M\ h_1\ h_2)\ (M\ h_2\ h_1)\ a) \Rightarrow Eq\ (M\ h_1\ h_2\ a)$$
$$\kappa_H : (Eq\ a, Eq\ ((f_1\ a), (f_2\ a))) \Rightarrow Eq\ (H\ f_1\ f_2\ a)$$
$$\kappa_G : Eq\ ((g\ a), (f\ (g\ a))) \Rightarrow Eq\ (G\ f\ g\ a)$$

A partial resolution tree generated by the query $Eq\ (M\ H\ G\ Int)$ is described in Figure 3. In this case the cycle (underlined with the index 1) is mutually nested with a loop (underlined with index 2). Our method in Section 3 is not able to generate any candidate lemmas. Yet there are two candidate lemmas for this case (with the proof of e_2 refer to e_1):

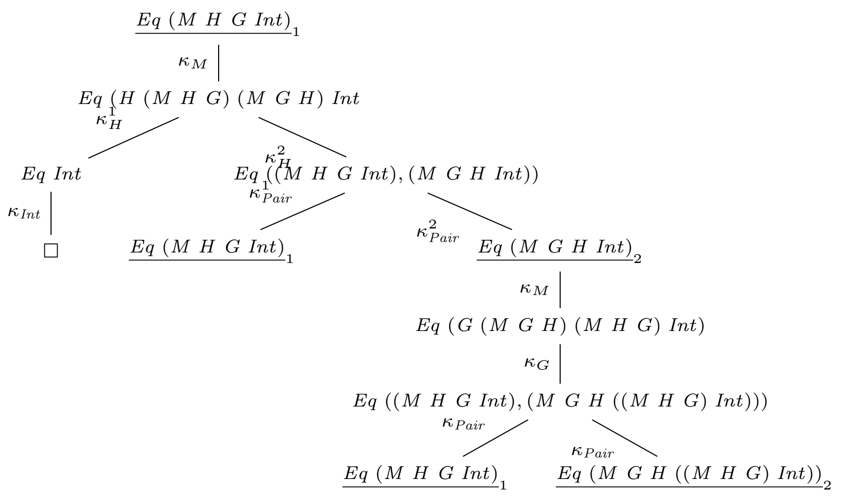
$$e_1 : (Eq\ x, Eq\ (M\ G\ H\ x)) \Rightarrow Eq\ (M\ H\ G\ x)$$
$$e_2 : Eq\ x \Rightarrow Eq\ (M\ G\ H\ x)$$

We would like to improve our heuristics to allow generating multiple candidate lemmas, where their corecursive evidences mutually refer to each other.

There are situations where resolution is non-terminating but does not form any loop such as $\Phi \vdash A \to^* \mathcal{C}[\sigma A]$. Consider the following program Φ_D:

$$\kappa_1 : D\ n\ (S\ m) \Rightarrow D\ (S\ n)\ m$$
$$\kappa_2 : D\ (S\ m)\ Z \Rightarrow D\ Z\ m$$

[7] See the extended version for more examples and information about the implementation. Extended version is available from authors' homepages.



$\underline{Eq\ (M\ H\ G\ Int)}_1$
κ_M
$Eq\ (H\ (M\ H\ G)\ (M\ G\ H)\ Int$
κ_H^1
κ_H^2
$Eq\ Int$
$Eq\ ((M\ H\ G\ Int), (M\ G\ H\ Int))$
κ_{Int}
$\square$
κ_{Pair}^1
κ_{Pair}^2
$\underline{Eq\ (M\ H\ G\ Int)}_1$
$\underline{Eq\ (M\ G\ H\ Int)}_2$
κ_M
$Eq\ (G\ (M\ G\ H)\ (M\ H\ G)\ Int)$
κ_G
$Eq\ ((M\ H\ G\ Int), (M\ G\ H\ ((M\ H\ G)\ Int)))$
κ_{Pair}
κ_{Pair}
$\underline{Eq\ (M\ H\ G\ Int)}_1$
$\underline{Eq\ (M\ G\ H\ ((M\ H\ G)\ Int))}_2$

Fig. 3: A Partial Resolution tree for $\Phi_M \vdash Eq\ (M\ H\ G\ Int) \Uparrow$

For query $D\ Z\ Z$, the resolution diverges without forming any loop. We would have to introduce extra equality axioms in order to obtain finite corecursive evidence.[8] We would like to investigate the ramifications of incorporating equality axioms in the corecursive resolution in the future.

We plan to extend the observational equivalence result of Section 5 to cope with more general notions of loop and extend our approach to allow intuitionistic formulas as candidate lemmas. Another avenue for future work is a formal proof that the calculus of Definition 13 is sound relative to the the greatest Herbrand models [18], and therefore reflects the broader notion of the coinductive entailment for Horn clause logic as discussed in the introduction.

Acknowledgements

We thank Patricia Johann and the FLOPS'16 reviewers for their helpful comments, and František Farka for many discussions. Part of this work was funded by the Flemish Fund for Scientific Research.

References

1. D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theory of Information and Applications*, 45(1):3–33, 2011.
2. R. Bird and L. Meertens. Nested datatypes. In *Mathematics of program construction*, pages 52–67. Springer, 1998.

[8] See the extended version for a solution in Haskell using type family and more discussion.

3. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.
4. T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs*, pages 62–78. Springer, 1994.
5. N. Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 1987.
6. P. Fu and E. Komendantskaya. A type-theoretic approach to resolution. In *25th International Symposium, LOPSTR 2015. Revised Selected Papers*, 2015.
7. J. Gibbons and G. Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4):353–366, 2005.
8. C. E. Gimenez. Un calcul de constructions infinies et son application a la vérification de systèmes communicants. 1996.
9. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge University Press Cambridge, 1989.
10. R. Hinze and S. Peyton-Jones. Derivable type classes. *Electronic notes in theoretical computer science*, 41(1):5–35, 2001.
11. J. Hughes. Restricted data types in Haskell. In *Haskell Workshop*, volume 99, 1999.
12. P. Johann and N. Ghani. Haskell programming with nested types: A principled approach. 2009.
13. P. Johann, E. Komendantskaya, and V. Komendantskiy. Structural Resolution for Logic Programming. In *Tech. Communications of ICLP'15*, July 2015.
14. M. P. Jones. *Qualified types: theory and practice*, volume 9. Cambridge University Press, 2003.
15. S. P. Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In *In Haskell Workshop*, 1997.
16. E. Komendantskaya and P.Johann. Structural resolution: a framework for coinductive proof search and proof construction in Horn clause logic. 2015. Submitted, http://arxiv.org/abs/1511.07865.
17. R. Lämmel and S. Peyton-Jones. Scrap your boilerplate with class: Extensible generic functions. In *Proc. 10th ACM SIGPLAN Int. Conference on Functional Programming*, ICFP '05, pages 204–215, New York, NY, USA, 2005. ACM.
18. J. W. Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 1987.
19. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied logic*, 51(1):125–157, 1991.
20. L. S. Moss and N. Danner. On the foundations of corecursion. *Logic Journal of IGPL*, 5(2):231–257, 1997.
21. G. D. Plotkin. A note on inductive generalization. *Machine intelligence*, 1970.
22. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *Automata, Languages and Programming*, pages 472–483. Springer, 2007.
23. M. Sulzmann, G. J. Duck, S. L. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, 2007.
24. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.
25. H. Zantema and A. Geser. *Non-looping rewriting.* Universiteit Utrecht, Faculty of Mathematics & Computer Science, 1996.

A Proof of Theorem 2

Theorem 3. *Let $\Phi \vdash B \to^* \mathcal{C}[D_1, ..., D_n, \sigma B]$ with $|\mathcal{C}| = \emptyset$ and D_i are normal for all i. Suppose $\Phi \vdash \sigma D_i \to^* \mathcal{C}_i[D_i]$, where $|\mathcal{C}_i| = \emptyset$ for any D_i. We have the following:*

1. $\Phi \vdash D_1, ..., D_n \Rightarrow B \Downarrow \nu\alpha.\lambda\alpha_1....\lambda\alpha_n.e$.
2. *Let* $e' \equiv \nu\alpha.\lambda\alpha_1....\lambda\alpha_n.e$. *We have:*
   $\mathcal{C}'[\sigma^m B] \in \mathcal{O}(B)_\Phi$ *iff* $\mathcal{C}'[e'\ \mathcal{C}_1^m[D_1]...\ \mathcal{C}_n^m[D_n]] \in \mathcal{S}(e'\ \underline{D})$.

Proof. 1. We have the following finite derivation.

$$\dfrac{\dfrac{\dfrac{\overline{...}}{\Phi, \alpha : \underline{D} \Rightarrow B, \underline{\alpha} : \underline{D} \vdash B \Downarrow e}}{\Phi, \alpha : \underline{D} \Rightarrow B \vdash \underline{D} \Rightarrow B \Downarrow \lambda\alpha_1....\lambda\alpha_n.e}}{\Phi \vdash \underline{D} \Rightarrow B \Downarrow \nu\alpha.\lambda\alpha_1....\lambda\alpha_n.e}$$

By Theorem 1, we just need to reduce B to a proof term using the rules $\Psi_1 = \Phi, \alpha : \underline{D} \Rightarrow B, \underline{\alpha} : \underline{D}$. We have the following reduction:

$$\Psi_1 \vdash B \to^* \mathcal{C}[D_1, ..., D_n, (\sigma B)] \to^* \mathcal{C}[\alpha_1...\alpha_n, (\sigma B)] \to$$
$$\mathcal{C}[\alpha_1, ..., \alpha_n, (\alpha\ (\sigma\ D_1)...(\sigma\ D_n))] \to^* \mathcal{C}[\alpha_1, ...\alpha_n, (\alpha\ \mathcal{C}_1[D_1]...\mathcal{C}_n[D_n])] \to$$
$$\mathcal{C}[\alpha_1...\alpha_n(\alpha\ \mathcal{C}_1[\alpha_1]...\ \mathcal{C}_n[\alpha_n])]$$

Thus we have the corecursive evidence $\nu\alpha.\lambda\alpha_1...\alpha_n.e$ for $D_1, ..., D_n \Rightarrow B$, and $e \equiv \mathcal{C}[\alpha_1, ...\alpha_n, (\alpha\ \mathcal{C}_1[\alpha_1]\ ...\ \mathcal{C}_n[\alpha_n])]$.

2. Using the same notation in (1), let $e' \equiv \nu\alpha.\lambda\alpha_1...\alpha_n.e$, we can observe following equivalence reduction traces:

$$\Phi \vdash B \to^* \mathcal{C}[D_1, ..., D_n, (\sigma B)] \to^*$$
$$\mathcal{C}[D_1, ..., D_n, \mathcal{C}[\mathcal{C}_1[D_1], ..., \mathcal{C}_n[D_n], (\sigma^2 B)]] \to ...$$

$$(e'\ D_1...\ D_n) \rightsquigarrow^* \mathcal{C}[D_1, ...D_n, (e'\ \mathcal{C}_1[D_1]\ ...\ \mathcal{C}_n[D_n])] \rightsquigarrow^*$$
$$\mathcal{C}[D_1, ..., D_n, \mathcal{C}[\mathcal{C}_1[D_1], ..., \mathcal{C}_n[D_n], (e'\ (\mathcal{C}_1[\mathcal{C}_1[D_1]])\ ...\ (\mathcal{C}_n[\mathcal{C}_n[D_n]]))]] \rightsquigarrow^* ...$$

We proceed by induction on m. When $m = 0$, it is obvious. Let $m = k + 1$. By IH, we know that $\mathcal{C}'[\sigma^k B] \in \mathcal{O}(B)_\Phi$ iff $\mathcal{C}'[e'\ \mathcal{C}_1^k[D_1]...\ \mathcal{C}_n^k[D_n]] \in \mathcal{S}(e'\ \underline{D})$.
We have $\Phi \vdash \mathcal{C}'[\sigma^k B] \to^* \mathcal{C}'[\mathcal{C}[\sigma^k D_1, ..., \sigma^k D_n, (\sigma^{k+1} B)]] \to^*$
$\mathcal{C}'[\mathcal{C}[\mathcal{C}_n^k[D_1], ..., \mathcal{C}_n^k[D_n], (\sigma^{k+1} B)]]$.
On the other hand, $\mathcal{C}'[e'\ \mathcal{C}_1^k[D_1]...\ \mathcal{C}_n^k[D_n]] \rightsquigarrow^*$
$\mathcal{C}'[\mathcal{C}[\mathcal{C}_n^k[D_1], ..., \mathcal{C}_n^k[D_n], (e'\ \mathcal{C}_1^{k+1}[D_1]...\ \mathcal{C}_n^{k+1}[D_n])]]$. Thus we have the observational equivalence.

B Weak Head Normalization of Corecursive Evidence

Definition 20.

$$\begin{array}{llcl} \textit{Formula} & F, G & ::= & A \mid \forall x.F \mid F \Rightarrow F' \\ \textit{Evidence/Proofs} & e & ::= & \kappa \mid \alpha \mid \lambda\alpha.e \mid e\ e' \mid \nu\alpha.e \\ \textit{Contexts/Axioms} & \Phi & ::= & \cdot \mid e : F, \Phi \end{array}$$

We write $G_1, ..., G_n \Rightarrow A$ as a shorthand for $G_1 \Rightarrow ... \Rightarrow G_n \Rightarrow A$. Note that Horn formulas are special case of formulas.

Definition 21. *Weak Head reduction context* $\mathcal{C} ::= \bullet \mid \mathcal{C}\ e$

Definition 22 (Weak Head Reduction). $\boxed{e \rightsquigarrow e'}$

$$\frac{}{\mathcal{C}[\nu\alpha.e] \rightsquigarrow_\nu \mathcal{C}[[\nu\alpha.e/\alpha]e]} \qquad \frac{}{\mathcal{C}[(\lambda\alpha.e)\ e'] \rightsquigarrow_\beta \mathcal{C}[[e'/\alpha]e]}$$

Note that weak head reduction context do not allow reduction under the constant κ. It is more restricted than the context in Section 2.

Definition 23 (General Corecursive Resolution).

$$\frac{\Phi \vdash \sigma G_1 \Downarrow e_1 \quad \cdots \quad \Phi \vdash \sigma G_n \Downarrow e_n}{\Phi \vdash \sigma A \Downarrow \kappa\ e_1 \cdots e_n}\ \textit{if}\ (e : G_1, ..., G_m \Rightarrow A) \in \Phi$$

$$\frac{\Phi, \alpha : F \vdash F \Downarrow e \quad \mathrm{HNF}(e)}{\Phi \vdash F \Downarrow \nu\alpha.e} \qquad \frac{\Phi, \underline{\alpha} : \underline{G} \vdash B \Downarrow e}{\Phi \vdash \underline{G} \Rightarrow B \Downarrow \lambda\underline{\alpha}.e}$$

Definition 24 (Howard's Type System).

$$\frac{(a : F) \in \Phi}{\Phi \vdash a : F}\ (\textsc{Assump}) \qquad \frac{\Phi \vdash e_1 : F' \quad \Phi \vdash e_2 : F' \Rightarrow F}{\Phi \vdash e_2\ e_1 : F}\ (\textsc{App})$$

$$\frac{\Phi, \alpha : F' \vdash e : F}{\Phi \vdash \lambda\alpha.e : F' \Rightarrow F}\ (\textsc{Abs}) \qquad \frac{\Phi \vdash e : F \quad x \notin \mathrm{FV}(\Phi)}{\Phi \vdash e : \forall x.F}\ (\textsc{Gen})$$

$$\frac{\Phi \vdash e : \forall x.F}{\Phi \vdash e : [t/x]F}\ (\textsc{Inst}) \qquad \frac{\Phi, \alpha : F \vdash e : F \quad \mathrm{HNF}(e)}{\Phi \vdash \nu\alpha.e : F}\ (\textsc{Mu})$$

Theorem 4. *If $\Phi \vdash F \Downarrow e$, then $\Phi \vdash e : F$.*

Proof. By induction on the derivation of $\Phi \vdash F \Downarrow e$.

Theorem 5. *If $\Phi \vdash e : F$, then e is terminating with respect to weak head reduction.*

Proof. Sketch. The proof is very similar to standard normalization proof for simply typed lambda calculus. The only tricky rule is the MU rule:

$$\frac{\Phi, \alpha : \underline{G} \Rightarrow A \vdash e : \underline{G} \Rightarrow A \quad \mathrm{HNF}(e)}{\Phi \vdash \nu\alpha.e : \underline{G} \Rightarrow A}$$

We want to show $\nu\alpha.e$ is in the reducible set of type $\underline{G} \Rightarrow A$. Since $e = \lambda\underline{\alpha}.\kappa\ \underline{e}$, we just need to show for any reducible $\underline{u}$ of type $\underline{G}$, we have $(\nu\alpha.e)\ \underline{u} \rightsquigarrow_\nu (\lambda\underline{\alpha}.\kappa\ [\nu\alpha.e/\alpha]\underline{e})\ \underline{u}$ is terminating. This is the case due to the expression is guarded by κ.

C Examples

In this section, we show several examples with the prototype that we developed. They are also available from the `examples` directory in `https://github.com/Fermat/corecursive-type-class`.

C.1 Example 1

```
module bush where
axiom (Eq a, Eq (f (f a))) => Eq (HBush f a)
axiom Eq (f (Mu f) a) => Eq (Mu f a)
axiom Eq Unit
auto Eq (Mu HBush Unit)
```

The keyword `axiom` introduces an axiom and the keyword `auto` requests the system to prove the formula automatically using the heuristic corecursive hypothesis generation that we described in Section 3. If we save the above code in the `bush.asl` file, and, at the command line, type `asl bush.asl`, then we get the following output:

```
Parsing success!
Type Checking success!
Program Definitions
  Ax0 :: (Eq (f (Mu f) a)) => Eq (Mu f a)
  = Ax0
  Ax1 :: (Eq a, Eq (f (f a))) => Eq (HBush f a)
  = Ax1
  Ax2 :: Eq Unit
  = Ax2
  genLemm4 :: (Eq var_1) => Eq (Mu HBush var_1)
  = \ b0 . Ax0 (Ax1 b0 (genLemm4 (genLemm4 b0)))
  goalLem3 :: Eq (Mu HBush Unit)
  = genLemm4 Ax2
Axioms
  Ax2 :: Eq Unit
  Ax1 :: (Eq a, Eq (f (f a))) => Eq (HBush f a)
  Ax0 :: (Eq (f (Mu f) a)) => Eq (Mu f a)
Lemmas
  goalLem3 :: Eq (Mu HBush Unit)
  genLemm4 :: (Eq var_1) => Eq (Mu HBush var_1)
```

The corecursive hypothesis generated is `genLemm4 :: (Eq var_1) => Eq (Mu HBush var_1)`, its proof is `\ b0 . Ax0 (Ax1 b0 (genLemm4 (genLemm4 b0)))`. The proof for `Eq (Mu HBush Unit)` is `genLemm4 Ax2`.

Using `axiom` and `auto` allows us to quickly experiment with different small examples. Here is the corresponding type-class code.

```
module bush where
data Unit where
  Unit :: Unit

data Maybe a where
  Nothing :: Maybe a
  Just :: a -> Maybe a

data Pair a b where
  Pair :: a -> b -> Pair a b

data HBush f a where
  HBLeaf :: HBush f a
  HBNode ::  a -> (f (f a)) -> HBush f a

data Mu f a where
  In :: f (Mu f) a -> Mu f a

data Bool where
     True :: Bool
     False :: Bool

class Eq a where
   eq :: Eq a => a -> a -> Bool

and = \ x y . case x of
                True -> y
                False -> False

instance  => Eq Unit where
   eq = \ x y . case x of
                   Unit -> case y of
                              Unit -> True
```

```
instance Eq a, Eq (f (f a)) => Eq (HBush f a) where
  eq = \ x y . case x of
                 HBLeaf -> case y of
                            HBLeaf -> True
                            HBNode a c -> False
                 HBNode a c1 -> case y of
                              HBLeaf -> False
                              HBNode b c2  -> and (eq a b) (eq c1 c2)

instance Eq (f (Mu f) a) => Eq (Mu f a) where
   eq = \ x y . case x of
                  In s -> case y of
              In t -> eq s t

term1 = In HBLeaf
term2 = In (HBNode Unit term1)
test = eq term2 term1
test1 = eq term2 term2
reduce test
reduce test1
```

Inspecting the output of this code, we see that the result of the evaluation of `test` (resp. `test1`) is `False` (resp. `True`). It is quite verbose and probably irrelevant to see the type-class code, so in the following we will show examples using only the `axiom`, `auto` and `lemma` keywords.

C.2 Example 2

```
module lam where
axiom Eq (f (Mu f) a) => Eq (Mu f a)
axiom (Eq a, Eq (f a), Eq (f a), Eq (f (Maybe a))) => Eq (HLam f a)
axiom Eq Unit
axiom Eq a => Eq (Maybe a)
lemma (Eq x) => Eq (Mu HLam x)
lemma Eq (Mu HLam Unit)
```

Of course, our heuristic `auto Eq (Mu HLam Unit)` also works for this case. But we can specify the corecursive hypothesis as lemma and guided our mini-prover to prove the final goal `Eq (Mu HLam Unit)`.

C.3 Example 3

Are there any examples where `auto` fails, but where we can come to the rescue with a `lemma`? The answer is yes.

```
axiom (Eq a, Eq (Pair (f1 a) (f2 a))) => Eq (H1 f1 f2 a)
axiom Eq (Pair (g a) (f (g a))) => Eq (H2 f g a)
axiom Eq (h1 (Mu h1 h2) (Mu h2 h1) a) => Eq (Mu h1 h2 a)
axiom (Eq a, Eq b) => Eq (Pair a b)
axiom Eq Unit
auto Eq (Mu H1 H2 Unit)
```

If we run this code, our mini-prover diverges, since this example require multiple lemmas in order to prove the final goal `Eq (Mu H1 H2 Unit)`.

```
axiom (Eq a, Eq (Pair (f1 a) (f2 a))) => Eq (H1 f1 f2 a)
axiom Eq (Pair (g a) (f (g a))) => Eq (H2 f g a)
axiom Eq (h1 (Mu h1 h2) (Mu h2 h1) a) => Eq (Mu h1 h2 a)
axiom (Eq a, Eq b) => Eq (Pair a b)
axiom Eq Unit
lemma (Eq x, Eq (Mu H2 H1 x)) => Eq (Mu H1 H2 x)
lemma Eq x => Eq (Mu H2 H1 x)
lemma Eq (Mu H1 H2 Unit)
```

C.4 Example 4

Are there any examples where even `lemma` does not work? We believe that the following is such an example.

```
axiom D n (S m) => D (S n) m
axiom D (S m) Z => D Z m
```

Note that `auto D Z Z` will not work because the corecursive hypothesis generated by our method is not provable. The following is a solution in Haskell using type families. We want to point out that using type families is a way to introduce equality axioms for addition, and these equality axioms are not derivable from the original axioms. The ability to learn addition seems to require a higher notion of intelligence.

```
{-# LANGUAGE Rank2Types, TypeFamilies, UndecidableInstances #-}
data Z
data S n
data D a b
type family Add m n
type instance Add n Z  =  n
type instance Add m (S n) = Add (S m) n

k1 :: D n (S m) -> D (S n) m
k1 = undefined
k2 :: D (S m) Z -> D Z m
k2 = undefined

f :: (forall n. D n (S m) -> D (S (Add m n)) Z) -> D Z m
f g = k2 (g (f (g . k1)))
```