Logical Methods in Computer Science
Vol. 13(3:28)2017, pp. 1–22
https://lmcs.episciences.org/

Submitted Nov. 30, 2016
Published Sep. 19, 2017

FORMAL LANGUAGES, FORMALLY AND COINDUCTIVELY

DMITRIY TRAYTEL

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
e-mail address: traytel@inf.ethz.ch

Abstract. Traditionally, formal languages are defined as sets of words. More recently, the alternative coalgebraic or coinductive representation as infinite tries, i.e., prefix trees branching over the alphabet, has been used to obtain compact and elegant proofs of classic results in language theory. In this article, we study this representation in the Isabelle proof assistant. We define regular operations on infinite tries and prove the axioms of Kleene algebra for those operations. Thereby, we exercise corecursion and coinduction and confirm the coinductive view being profitable in formalizations, as it improves over the set-of-words view with respect to proof automation.

1. Introduction

If we ask a computer scientist what a formal language is, the answer will most certainly be: a set of words. Here, we advocate another valid answer: an infinite trie. This is the coalgebraic approach to languages [30], viewed through the lens of a lazy functional programmer.

This article shows how to formalize the coalgebraic or coinductive approach to formal languages in the Isabelle/HOL proof assistant in the form of a gentle introduction to corecursion and coinduction. Our interest in the coalgebraic approach to formal languages arose in the context of a larger formalization effort of coalgebraic decision procedures for regular languages [36, 37]. Indeed, we present here a reusable library modeling languages, which lies at the core of those formalized decision procedures. A lesson we have learned from this exercise and hope to convey here is that often it is worthwhile to look at well-understood objects from a different (in this case coinductive) perspective.

The literature is abound with tutorials on coinduction. So why bother writing yet another one? First, because we finally can do it in Isabelle/HOL, which became a coinduction-friendly proof assistant recently [6]. Earlier studies of coinduction in Isabelle had to engage in tedious manual constructions just to define a coinductive datatype [25]. Second, coinductive techniques are still not as widespread as they could be (and we believe should be, since they constitute a convenient proof tool for questions about semantics). Third, many tutorials [11, 14, 18, 19, 21, 33], with or without a theorem prover, exercise streams to the extent that one starts to believe having seen every single stream example one can imagine. In contrast, Rutten [30] demonstrates that it is entirely feasible to start a tutorial with a structure slightly more complicated than streams, but familiar to every computer scientist. Moreover, Rot, Bonsangue, and Rutten [28, 29] present an accessible introduction to coinduction up to

DOI:10.23638/LMCS-13(3:28)2017

congruence using the coinductive view of formal languages. Our work additionally focuses on corecursion up-to and puts Rutten's exposition in the context of a proof assistant.

When programming with infinite structures in the total setting of a proof assistant, productivity must be ensured. Intuitively, a corecursive function is productive if it always eventually yields observable output, e.g., in form of a constructor. Functions that output exactly one constructor before proceeding corecursively or stopping with a fixed (non-corecursive) value are called primitively corecursive—a fragment dual to well-understood primitively recursive functions on inductive datatypes. Primitively corecursive functions are productive. While sophisticated methods involving domain, measure, and category theory for handling more complex corecursive specifications have been proposed [7, 23] and implemented in Isabelle [5], we explore here how far primitive corecursion can get us. Restricting ourselves to this fragment is beneficial in several ways. First, our constructions become mostly Isabelle independent, since primitive corecursion is supported by all coinduction-friendly proof assistants. Second, when working in the restricted setting, we quickly hit and learn to understand the limits. In fact, we will face some non-primitively corecursive specifications on infinite tries, which we reduce to a composition of primitively corecursive specifications. Those reductions are insightful and hint at a general pattern for handling certain non-primitively corecursive specifications.

Infinite data structures are often characterized in terms of observations. For infinite tries, which we define as a coinductive datatype or short codatatype (Section 2), we can observe the root, which in our case is labeled by a Boolean value. This label determines if the empty word is *accepted* by the trie. Moreover, we can observe the immediate subtrees of a trie, of which we have one for each alphabet letter. This observation corresponds to making transitions in an automaton or rather computing the left quotient $L_a = \{w \mid aw \in L\}$ of the language L by the letter a. Indeed, we will see that Brzozowski's ingenious derivative operation [10], which mimics this computation recursively on the syntax of regular expressions, arises very naturally when defining regular operations corecursively on tries (Section 3). To validate our definitions, we formally prove by coinduction that they satisfy the axioms of Kleene algebra. Thereby, we use two similar but different coinduction principles for equalities and inequalities (Section 4). After having presented our formalization, we step back and connect concrete intuitive notions (such as tries) with abstract coalgebraic terminology (Section 5). Furthermore, we discuss our formalization and its relation to other work on corecursion and coinduction with or without proof assistants (Section 6).

This article extends the homonymous FSCD 2016 paper [38] with the definition of the shuffle product on tries (Subsection 3.3), a trie construction for context-free grammars (Subsection 3.4), the coinductive treatment of inequalities (Subsection 4.2) instead of reducing them to equalities, and the proof that the trie construction for context-free grammars is sound (Subsection 4.3). The material presented here is based on the publicly available Isabelle/HOL formalization [35] and is partly described in the author's Ph.D. thesis [37].

Preliminaries. Isabelle/HOL is a proof assistant for higher-order logic, built around a small trusted inference kernel. The kernel accepts only non-recursive type and constant definitions. High-level specification mechanisms, which allow the user to enter (co)recursive specifications, reduce this input to something equivalent but non-recursive. The original (co)recursive specification is later derived as a theorem. For a comprehensive introduction to Isabelle/HOL we refer to a recent textbook [24, Part I].

In Isabelle/HOL types τ are built from type variables α, β, etc., via type constructors κ written postfix (e.g., $\alpha\ \kappa$). Some special types are the sum type $\alpha + \beta$, the product type $\alpha \times \beta$, and the function type $\alpha \to \beta$, for which the type constructors are written infix. Infix operators bind less tightly than the postfix or prefix ones. Other important types are the type of Booleans *bool* inhabited by exactly two different values $\top$ (truth) and $\bot$ (falsity) and the types α *list* and α *set* of lists and sets of elements of type α. For Boolean connectives and sets common mathematical notation is used. A special constant is equality $= :: \alpha \to \alpha \to \mathit{bool}$, which is polymorphic (it exists for any type, including the function type, on which it is extensional, i.e., $(\forall x.\ f\ x = g\ x) \longrightarrow f = g$). The functions Inl and Inr are the standard embeddings of $+$. Lists are constructed from $[] :: \alpha\, \mathit{list}$ and $\# :: \alpha \to \alpha\, \mathit{list} \to \alpha\, \mathit{list}$; the latter written infix and often omitted, i.e., we write aw for $a \# w$. Likewise, list concatenation $+\!\!+$ is written infix and may be omitted. The notation $|w|$ stands for the length of the list w, i.e., $|[]| = 0$ and $|aw| = 1 + |w|$.

2. Languages as Infinite Tries

We define the type of formal languages as a codatatype of infinite tries, that is, (prefix) trees of infinite depth branching over the alphabet. We represent the alphabet by the type parameter α. Each node in a trie carries a Boolean label, which indicates whether the (finite) path to this node constitutes a word inside or outside of the language. The function type models branching: for each letter $x :: \alpha$ there is a subtree, which we call *x-subtree*.

$$\texttt{codatatype}\ \alpha\, \mathit{lang} = \mathsf{L}\ (o : \mathit{bool})\ (\delta : \alpha \to \alpha\, \mathit{lang})$$

The `codatatype` command defines the type $\alpha\, \mathit{lang}$ together with a *constructor* $\mathsf{L} :: \mathit{bool} \to (\alpha \to \alpha\, \mathit{lang}) \to \alpha\, \mathit{lang}$ and two *selectors* $o :: \alpha\, \mathit{lang} \to \mathit{bool}$ and $\delta :: \alpha\, \mathit{lang} \to \alpha \to \alpha\, \mathit{lang}$. For a binary alphabet $\alpha = \{a, b\}$, the trie *even* shown in Figure 1 is an inhabitant of $\alpha\, \mathit{lang}$. The label of its root is given by $o\ \mathit{even} = \top$ and its subtrees by another trie $\delta\ \mathit{even}\ a = \delta\ \mathit{even}\ b = \mathit{odd}$. Similarly, we have $o\ \mathit{odd} = \bot$ and $\delta\ \mathit{odd}\ a = \delta\ \mathit{odd}\ b = \mathit{even}$. Note that we could have equally written $\mathit{even} = \mathsf{L}\ \top\ (\lambda\_.\ \mathit{odd})$ and $\mathit{odd} = \mathsf{L}\ \bot\ (\lambda\_.\ \mathit{even})$ to obtain the same mutual characterization of *even* and *odd*.

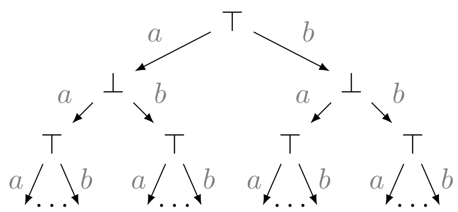

Figure 1: Infinite trie *even*

We gave our type the name $\alpha\, \mathit{lang}$, to remind us to think of its inhabitants as formal languages. In the following, we use the terms language and trie synonymously.

Beyond defining the type and the constants, the `codatatype` command also exports a wealth of properties about them such as $o\ (\mathsf{L}\ b\ d) = b$, the injectivity of L, or more interestingly the coinduction rule. Informally, coinduction allows us to prove equality of tries which cannot be distinguished by finitely many selector applications.

Clearly, we would like to identify the trie *even* with the regular language of all words of even length $\{w \in \{a, b\}^* \mid |w| \ \mathsf{mod}\ 2 = 0\}$, also represented by the regular expression $((a+b)\cdot(a+b))^*$. Therefore, we define the notion of word membership $\Subset$ on tries by primitive (or structural) recursion on the word using Isabelle's `primrec` command.

$$
\begin{array}{l}
\texttt{primrec}\ \Subset :: \alpha\ \mathit{list} \to \alpha\ \mathit{lang} \to \mathit{bool}\ \texttt{where} \\
\quad [] \Subset L = o\ L \\
\quad aw \Subset L = w \Subset \delta\ L\ a
\end{array}
$$

Using $\Subset$, each trie can be assigned a language in the traditional set of lists view.

$$
\begin{array}{l}
\texttt{definition}\ \mathsf{out} :: \ \alpha\ \mathit{lang} \to \alpha\ \mathit{list}\ \mathit{set}\ \texttt{where} \\
\quad \mathsf{out}\ L = \{w \mid w \Subset L\}
\end{array}
$$

With this definition, we obtain $\mathsf{out}\ \mathit{even} = \{w \in \{a, b\}^* \mid |w| \ \mathsf{mod}\ 2 = 0\}$.

3. Operations on Tries

So far, we have only specified some concrete infinite tries informally. Formally, we will use primitive corecursion, which is dual to primitive recursion. Primitively recursive functions consume one constructor before proceeding recursively. Primitively corecursive functions produce one *guarding* constructor whose arguments are allowed to be either non-recursive terms or a corecursive call (applied to arbitrary non-recursive arguments). We call a function truly primitively corecursive if not all of the constructor's argument are non-recursive. The `primcorec` command reduces a primitively corecursive specification to a non-recursive definition, which is accepted by Isabelle's inference kernel [6]. Internally, the reduction employs a dedicated combinator for primitive corecursion on tries generated by the `codatatype` command. The `primcorec` command slightly relaxes the above restriction of primitive corecursion by allowing syntactic conveniences, such as lambda abstractions, `case`-, and `if`-expressions, to appear between the guarding constructor and the corecursive call. The relaxation is resembling the syntactic guardedness check used in type theory [12, Section 2.3], however still allowing only exactly one constructor to guard a corecursive call.

3.1. **Primitively Corecursive Operations.** We start with some simple examples: the languages of the base cases of regular expressions. Intuitively, the trie $\varnothing$ representing the empty language is labeled with $\bot$ everywhere and the trie ε representing the empty word language is labeled with $\top$ at its root and with $\bot$ everywhere else. The trie $\mathsf{A}\ a$ representing the singleton language of the one letter word a is labeled with $\bot$ everywhere except for the root of its a-subtree. This intuition is easy to capture formally.

$$
\begin{array}{ll}
\texttt{primcorec}\ \varnothing :: \alpha\ \mathit{lang}\ \texttt{where} & \texttt{primcorec}\ \varepsilon :: \alpha\ \mathit{lang}\ \texttt{where} \\
\quad \varnothing = \mathsf{L}\ \bot\ (\lambda x.\ \varnothing) & \quad \varepsilon = \mathsf{L}\ \top\ (\lambda x.\ \varnothing)
\end{array}
$$

$$
\begin{array}{l}
\texttt{primcorec}\ \mathsf{A} :: \alpha \to \alpha\ \mathit{lang}\ \texttt{where} \\
\quad o\ (\mathsf{A}\ a) = \bot \\
\quad \delta\ (\mathsf{A}\ a) = \lambda x.\ \texttt{if}\ a = x\ \texttt{then}\ \varepsilon\ \texttt{else}\ \varnothing
\end{array}
$$

Among these three definitions only $\varnothing$ is truly primitively corecursive.

The specifications of $\varnothing$ and ε differ syntactically from the one of A. The constants $\varnothing$ and ε are defined using the so called *constructor view.* The constructor view allows the user to enter equations of the form constant or function equals constructor, where the arguments of the constructor are restricted as described above. Such definitions should be familiar to any (lazy) functional programmer.

In contrast, the specification of A is expressed in the *destructor view*. Here, we specify the constant or function being defined by observations or experiments via *selector equations*. The allowed experiments on a trie are given by its selectors o and δ. We can observe the label at the root and the subtrees. Specifying the observation for each selector—again restricted either to be a non-recursive term or to contain the corecursive call only at the outermost position (ignoring lambda abstractions, `case`-, and `if`-expressions)—yields a unique characterization of the function being defined.

It is straightforward to rewrite specifications in either of the views into the other one. The `primcorec` command performs this rewriting internally and outputs the theorems corresponding to the user's input specification in both views. The constructor view theorems serve as executable code equations. Isabelle's code generator [17] can use these equations to generate code which make sense in programming languages with lazy evaluation. In contrast, the destructor view offers safe simplification rules even when applied eagerly during rewriting as done by Isabelle's simplifier. Note that constructor view specifications such as $\varnothing = \mathsf{L}\ \bot\ (\lambda x.\ \varnothing)$ will cause the simplifier to loop when applied eagerly.

Now that the basic building blocks $\varnothing$, ε, and A are in place, we turn our attention to how to combine them to obtain more complex languages. We start with the simpler combinators for union, intersection, and complement, before moving to the more interesting concatenation and iteration. The union $+$ of two tries should denote set union of languages (i.e., $\mathsf{out}\ (L + K) = \mathsf{out}\ L \cup \mathsf{out}\ K$ should hold). It is defined corecursively by traversing the two tries in parallel and computing for each pair of labels their disjunction. Intersection $\cap$ is analogous. Complement $\overline{}$ simply inverts every label.

$$
\begin{array}{l}
\texttt{primcorec}\ + :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where} \\
\quad o\ (L + K) = o\ L \vee o\ K \\
\quad \delta\ (L + K) = \lambda x.\ \delta\ L\ x + \delta\ K\ x
\end{array}
$$

$$
\begin{array}{l}
\texttt{primcorec}\ \cap :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where} \\
\quad o\ (L \cap K) = o\ L \wedge o\ K \\
\quad \delta\ (L \cap K) = \lambda x.\ \delta\ L\ x \cap \delta\ K\ x
\end{array}
$$

$$
\begin{array}{l}
\texttt{primcorec}\ \overline{} :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where} \\
\quad o\ \overline{L} = \neg\ o\ L \\
\quad \delta\ \overline{L} = \lambda x.\ \overline{\delta\ L\ x}
\end{array}
$$

Let us look at the specifying selector equations which we have seen so far from a different perspective. Imagine L and K being not tries but instead syntactic regular expressions, A, $+$, $\cap$, and $\overline{}$ constructors of a datatype for regular expressions, and o and δ two operations that we define recursively on this syntax. From that perspective, the operations are familiar: rediscovered Brzozowski derivatives of regular expressions [10] and the empty word acceptance (often also called *nullability*) test on regular expressions in the destructor view equations for the selectors δ and o. There is an important difference, though: while Brzozowski derivatives work with syntactic objects, our tries are semantic objects on which equality denotes language equivalence. For example, we will later prove $\varnothing + L = L$ for tries, whereas $\varnothing + L \neq L$ holds for regular expressions. The coinductive view reveals that derivatives and the acceptance test are the two fundamental ingredients that completely characterize regular languages and arise naturally when only considering the semantics.

Figure 2: Tries for L (left) and the concatenation $L \cdot K$ (right)

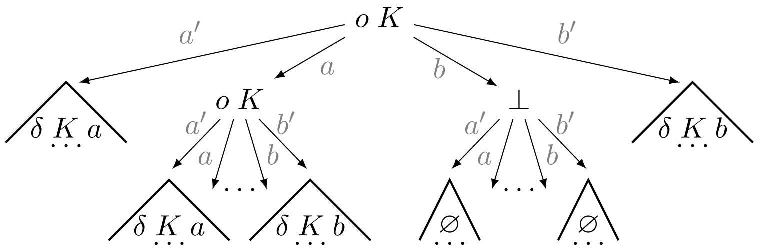

Figure 3: Trie for deferred concatenation $L \mathbin{\hat{\cdot}} K$

3.2. **Reducing Corecursion Up-to to Primitive Corecursion.** Concatenation $\cdot$ is the next regular operation that we want to define on tries. Thinking of Brzozowski derivatives and the acceptance test, we would usually specify it by the following two equations.

$$\begin{aligned} o\ (L \cdot K) &= o\ L \wedge o\ K \\ \delta\ (L \cdot K) &= \lambda x.\ (\delta\ L\ x \cdot K) + (\texttt{if}\ o\ L\ \texttt{then}\ \delta\ K\ x\ \texttt{else}\ \varnothing) \end{aligned} \tag{3.1}$$

A difficulty arises here, since this specification is not primitively corecursive—the right hand side of the second equation contains a corecursive call but not at the topmost position (but rather under $+$ here). We call this kind of corecursion *up to* $+$.

Without tool support for corecursion up-to, concatenation must be defined differently—as a composition of other primitively corecursive operations. Intuitively, we would like to separate the above δ-equation into two along the $+$ and sum them up afterwards. Technically, the situation is more involved. Since the δ-equation is corecursive, we cannot just create two tries by primitive corecursion.

Figure 2 depicts the trie that should result from concatenating an arbitrary trie K to the concrete given trie L. Procedurally, the concatenation must replace every subtree T of L that has $\top$ at the root (those are positions where words from L end) by the trie $U + K$ where U is the trie obtained from T by changing its root from $\top$ to $o\ K$. For uniformity with the above δ-equation, we imagine subtrees F of L with label $\perp$ at the root as also being replaced by $F + \varnothing$, which, as we will prove later, has the same effect as leaving F alone.

Figure 3 presents one way to bypass the restrictions imposed by primitive corecursion. We are not allowed to use $+$ *after* proceeding corecursively, but we may arrange the arguments of $+$ in a broader trie over a doubled alphabet formally modeled by pairing letters of the alphabet with a Boolean flag. In Figure 3 we write a for $(a, \top)$ and a' for $(a, \perp)$. Because it defers the summation, we call this primitively corecursive procedure *deferred concatenation* $\hat{\cdot}$.

$$\begin{aligned} &\texttt{primcorec}\ \hat{\cdot} :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang} \to (\alpha \times \mathit{bool})\ \mathit{lang}\ \texttt{where} \\ &\quad o\ (L \mathbin{\hat{\cdot}} K) = o\ L \wedge o\ K \\ &\quad \delta\ (L \mathbin{\hat{\cdot}} K) = \lambda(x, b).\ \texttt{if}\ b\ \texttt{then}\ \delta\ L\ x \mathbin{\hat{\cdot}} K\ \texttt{else}\ \texttt{if}\ o\ L\ \texttt{then}\ \delta\ K\ x \mathbin{\hat{\cdot}} \varepsilon\ \texttt{else}\ \varnothing \end{aligned}$$

Note that unlike in the Figure 3, where we informally plug the trie $\delta\ K\ x$ as some x'-subtrees, the formal definition must be more careful with the types. More precisely, $\delta\ K\ x$ is of type $\alpha\ \mathit{lang}$, while something of type $(\alpha \times \mathit{bool})\ \mathit{lang}$ is expected. This type mismatch is resolved by further concatenating ε to $\delta\ K\ x$ (again in a deferred fashion) without corrupting the intended semantics.

Once the trie for the deferred concatenation has been built, the desired trie for concatenation can be obtained by a second primitively corecursive traversal that sums the x- and x'-subtrees *before* proceeding corecursively.

$$\begin{array}{l}\texttt{primcorec}\ \hat{\oplus} :: (\alpha \times \mathit{bool})\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where} \\ \quad o\ (\hat{\oplus}\ L) = o\ L \\ \quad \delta\ (\hat{\oplus}\ L) = \lambda x.\ \hat{\oplus}\ (\delta\ L\ (x,\ \top) + \delta\ L\ (x,\ \bot))\end{array}$$

Finally, we can define the concatenation as the composition of $\hat{\cdot}$ and $\hat{\oplus}$. The earlier natural selector equations (3.1) for $\cdot$ are provable for this definition.

$$\begin{array}{l}\texttt{definition}\ \cdot :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where} \\ \quad L \cdot K = \hat{\oplus}\ (L\ \hat{\cdot}\ K)\end{array}$$

The situation with iteration is similar. The selector equations following the Brzozowski derivative of L^* yield a non-primitively corecursive specification: it is corecursive up to $\cdot$.

$$\begin{aligned} o\ (L^*) &= \top \\ \delta\ (L^*) &= \lambda x.\ \delta\ L\ x \cdot L^* \end{aligned} \tag{3.2}$$

As before, the restriction is circumvented by altering the operation slightly. We define the binary operation *deferred iteration* $L\ \hat{*}\ K$, whose language should represent $L \cdot K^*$ (although we have not defined * yet). When constructing the subtrees of $L\ \hat{*}\ K$ we keep pulling copies of the second argument into the first argument before proceeding corecursively (the second argument itself stays unchanged).

$$\begin{array}{l}\texttt{primcorec}\ \hat{*} :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where} \\ \quad o\ (L\ \hat{*}\ K) = o\ L \\ \quad \delta\ (L\ \hat{*}\ K) = \lambda x.\ (\delta\ (L \cdot (\varepsilon + K))\ x)\ \hat{*}\ K\end{array}$$

Supplying ε as the first argument to $\hat{*}$ defines iteration for which the original selector equations (3.2) hold.

$$\begin{array}{l}\texttt{definition}\ \_^* :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where} \\ \quad L^* = \varepsilon\ \hat{*}\ L\end{array}$$

We have defined all the standard regular operations on tries. Later we will prove that those definitions satisfy the axioms of Kleene algebra, meaning that they behave as expected. Already now we can compose the operations to define new tries, for example the introductory $\mathit{even} = ((\mathsf{A}\ a + \mathsf{A}\ b) \cdot (\mathsf{A}\ a + \mathsf{A}\ b))^*$.

3.3. **Adding Further Operations.** In the coinductive representation, adding new operations corresponds to defining a new corecursive function on tries. Compared with adding a new constructor to the inductive datatype of regular expressions and extending all previously defined recursive functions on regular expressions to account for this new case, this a is rather low-cost library extension. Wadler called this tension between extending syntactic and semantic objects the Expression Problem [40].[1]

As an example library extension, we define the regular shuffle product operation on languages, which adheres to the following selector equations.

$$\begin{aligned} &o\ (L \parallel K) = o\ L \wedge o\ K \\ &\delta\ (L \parallel K) = \lambda x.\ (\delta\ L\ x \parallel K) + (L \parallel \delta\ K\ x) \end{aligned} \tag{3.3}$$

Intuitively, for words $w \Subset L$ and $v \Subset K$, the shuffle product $L \parallel K$ contains all possible interleavings of w and v. As it is the case for concatenation, the selector equations for the shuffle product $\parallel$ are corecursive up to $+$. Thus, as for $\cdot$, we first define a *deferred shuffle product* operation, which keeps the $+$ occurring outside of the corecursive calls in the second equation "unevaluated" by using the doubled alphabet $\alpha \times bool$ instead of α.

$$\begin{aligned} &\texttt{primcorec}\ \hat{\parallel} :: \alpha\ lang \to \alpha\ lang \to (\alpha \times bool)\ lang\ \texttt{where} \\ &\quad o\ (L \mathbin{\hat{\parallel}} K) = o\ L \wedge o\ K \\ &\quad \delta\ (L \mathbin{\hat{\parallel}} K) = \lambda(x, b).\ \texttt{if}\ b\ \texttt{then}\ \delta\ L\ x \mathbin{\hat{\parallel}} K\ \texttt{else}\ L \mathbin{\hat{\parallel}} \delta\ K\ x \end{aligned}$$

A second primitively corecursive traversal sums the $(a, \top)$- and $(a, \bot)$-subtrees using the same function $\hat{\oplus}$ as in the definition of concatenation. Then, the shuffle product can be defined as the composition of $\hat{\parallel}$ and $\hat{\oplus}$.

$$\begin{aligned} &\texttt{definition}\ \parallel :: \alpha\ lang \to \alpha\ lang \to \alpha\ lang\ \texttt{where} \\ &\quad L \parallel K = \hat{\oplus}\ (L \mathbin{\hat{\parallel}} K) \end{aligned}$$

3.4. **Context-Free Grammars as Tries.** Tries are not restricted to regular operations. We define an operation that transforms a context-free grammar (in a particular normal form) into the trie denoting the same context-free language. The operation borrows ideas from Winter's et al. [41] coalgebraic exposition of context-free languages.

For the rest of the subsection, we fix a context-free grammar given by the type of its terminal symbols $\mathcal{T}$, the type of its non-terminal symbols $\mathcal{N}$, a distinguished starting non-terminal $\mathsf{S} :: \mathcal{N}$, and its productions $\mathsf{P} :: \mathcal{N} \to (\mathcal{T} + \mathcal{N})\ list\ set$—an assignment of a set of words consisting of terminal and non-terminal symbols for each non-terminal. For readability, we write T for $\mathsf{Inl} :: \mathcal{T} \to \mathcal{T} + \mathcal{N}$ and NT for $\mathsf{Inr} :: \mathcal{N} \to \mathcal{T} + \mathcal{N}$ in this section. The inductive semantics of such a grammar is standard: a word over the alphabet $\mathcal{T}$ is accepted, if there is a finite derivation of that word using a sequence of productions. (For a formal definition, see Subsection 4.3.) For example, the grammar given by the productions

$$\mathsf{P}\ N = \texttt{if}\ N = \mathsf{S}\ \texttt{then}\ \{[], [\mathsf{T}\ a], [\mathsf{T}\ b], [\mathsf{T}\ a, \mathsf{NT}\ \mathsf{S}, \mathsf{T}\ a], [\mathsf{T}\ b, \mathsf{NT}\ \mathsf{S}, \mathsf{T}\ b]\}\ \texttt{else}\ \{\}$$

[1] Note that codatatypes alone are not the solution to the Expression Problem—they just populate the other side of the spectrum with respect to datatypes. In fact, adding new selectors to tries would be as painful as adding new constructors to the datatype of regular expressions. Rendel et al. [27] outline how automatic conversions between the inductive and the coinductive view can help solving the Expression Problem.

where $a,\, b :: \mathcal{T}$ with $a \neq b$, denotes the non-regular language of palindromes over $\{a,\, b\}$. In conventional syntax, we would write the above productions as $\mathsf{S} \to \varepsilon \mid a \mid b \mid a\mathsf{S}a \mid b\mathsf{S}b$.

To give a coinductive semantics in form of a trie to a grammar, we must solve the word problem for context-free grammars and use that algorithm to assign corecursively the Boolean labels in the trie. Different algorithms solve the word problem for context-free grammars. Earley's algorithm [13] is the most flexible (but also complex) one: it works for arbitrary grammars without requiring a syntactic normal form. To simplify the presentation, we work with a syntactic normal form that allows us to use a much simplified version of Earley's algorithm. We require the productions to be in *weak Greibach normal form* [41]: every produced word should either be the empty list or start with a terminal. Formally:

$$\forall N.\ \forall \alpha \in \mathsf{P}\ N.\ \texttt{case}\ \alpha\ \texttt{of}\ \mathsf{NT}\ M \# \_ \Rightarrow \bot \mid \_ \Rightarrow \top$$

Weak Greibach normal form is a relaxation of the standard Greibach normal form [15], which additionally requires the starting terminal to be followed only by non-terminals. The example palindrome grammar is in weak Greibach normal form but not in Greibach normal form.

The intermediate *states* α, β, ... of a word derivation are words of type $(\mathcal{T} + \mathcal{N})\ \mathit{list}$, which are reachable from the initial non-terminal. We observe that such a state can only produce the empty word according to P, if it consists only of non-terminals, each of which can immediately produce the empty word, i.e., $[] \in \mathsf{P}\ N$. Note that due to weak Greibach normal form any non-$[]$ production will produce at least one terminal symbol. We compute recursively whether a state can produce the empty word according to P .

$$\begin{array}{l}
\texttt{primrec}\ o_{\mathsf{P}} :: (\mathcal{T} + \mathcal{N})\ \mathit{list} \to \mathit{bool}\ \texttt{where} \\
\quad o_{\mathsf{P}}\ [] = \top \\
\quad o_{\mathsf{P}}\ (x \# \alpha) = \texttt{case}\ x\ \texttt{of}\ \mathsf{T}\ _ \Rightarrow \bot \mid \mathsf{NT}\ N \Rightarrow [] \in \mathsf{P}\ N \wedge o_{\mathsf{P}}\ \alpha
\end{array}$$

A second useful function δ_{P} "reads" a terminal symbol a in a state from the left, yielding a set of successor states to choose from non-deterministically. The recursive definition of δ_{P} is based on a similar observation as the one of o_{P}. If the state starts with a terminal b, the only successor state is the tail of the state if $a = b$. There is no successor state if $a \neq b$. If the state starts with a non-terminal N, we consider all non-empty productions in $\mathsf{P}\ N$ starting with a and replace N with their tails. Additionally, if N may produce the empty word, we drop it and continue recursively with the next terminal or non-terminal symbol of the state, which possibly results in additional successor states. Formally:

$$\begin{array}{l}
\texttt{primrec}\ \delta_{\mathsf{P}} :: (\mathcal{T} + \mathcal{N})\ \mathit{list} \to \mathcal{T} \to (\mathcal{T} + \mathcal{N})\ \mathit{list}\ \mathit{set}\ \texttt{where} \\
\quad \delta_{\mathsf{P}}\ []\ _ = \{\} \\
\quad \delta_{\mathsf{P}}\ (x \# \alpha)\ a = \texttt{case}\ x\ \texttt{of} \\
\qquad\quad \mathsf{T}\ b \Rightarrow \texttt{if}\ a = b\ \texttt{then}\ \{\alpha\}\ \texttt{else}\ \{\} \\
\qquad \mid \mathsf{NT}\ N \Rightarrow \{\beta +\!\!+ \alpha \mid (\mathsf{T}\ a) \# \beta \in \mathsf{P}\ N\} \cup \texttt{if}\ [] \in \mathsf{P}\ N\ \texttt{then}\ \delta_{\mathsf{P}}\ \alpha\ a\ \texttt{else}\ \{\}
\end{array}$$

Finally, we obtain a trie from a set of states by primitive corecursion using the two above functions to specify the observations. Note that the set of states changes when proceeding corecursively. For this definition we use the constructor view.

$$\begin{array}{l}
\texttt{primcorec}\ \mathsf{close} :: (\mathcal{T} + \mathcal{N})\ \mathit{list}\ \mathit{set} \to \mathcal{T}\ \mathit{lang}\ \texttt{where} \\
\quad \mathsf{close}\ X = \mathsf{L}\ (\exists \alpha \in X.\ o_{\mathsf{P}}\ \alpha)\ (\lambda a.\ \mathsf{close}\ \big(\bigcup_{\alpha \in X} \delta_{\mathsf{P}}\ \alpha\ a\big))
\end{array}$$

The initial state, in which no terminal has been read yet, is the singleton list $[\mathsf{S}]$. We obtain the trie G corresponding to our fixed grammar.

$\texttt{definition}$ $\mathsf{G} :: \mathcal{T}\ lang$ $\texttt{where}$
$\mathsf{G} = \mathsf{close}\ \{[\mathsf{S}]\}$

3.5. **Arbitrary Formal Languages.** Before we turn to proving, let us exercise one more corecursive definition. Earlier, we have assigned each trie a set of lists via the function out. Primitive corecursion enables us to define the converse operation.

$\texttt{primcorec}$ $\mathsf{in} :: \alpha\ list\ set \rightarrow \alpha\ lang$ $\texttt{where}$
$o\ (\mathsf{in}\ L) = [] \in L$
$\delta\ (\mathsf{in}\ L) = \lambda a.\ \mathsf{in}\ \{w \mid aw \in L\}$

The function out and in are both bijections. We show this by proving that their compositions (either way) are both the identity function. One direction, $\mathsf{out}\ (\mathsf{in}\ L) = L$, follows by set extensionality and a subsequent induction on words. The reverse direction requires a proof by coinduction, which is the topic of the next section.

Using in we can turn every (even undecidable) set of lists into a trie. This is somewhat counterintuitive, since, given a concrete trie, its word problem seems easily decidable (via the function $\Subset$). But of course in order to compute the trie out of a set of lists L via in the word problem for L must be solved—we are reminded that higher-order logic is not a programming language where everything is executable, but a logic in which we write down specifications (and not programs). For regular operations and context-free grammars from the previous subsections the situation is different. For example, Isabelle's code generator can produce valid Haskell code that evaluates $abaa \Subset (\mathsf{A}\ a \cdot (\mathsf{A}\ a + \mathsf{A}\ b))^*$ to $\top$ and $abaa \Subset \mathsf{G}$ to $\bot$, where G is the trie for the palindrome grammar from Subsection 3.4. The latter is possible, despite the seemingly non-executable existential quantification and unions in the definition of close, due to Isabelle's code generator, which makes (co)finite sets executable through data refinement to lists or red-black trees [16].

4. Reasoning about Tries

We have seen the definitions of many operations, justifying their meaningfulness by appealing to the reader's intuition. This is often not enough to guarantee correctness, especially if we have a theorem prover at hand. So let us formally prove in Isabelle that the regular operations on tries form a Kleene algebra by proving Kozen's famous axioms [20] as equalities (Subsection 4.1) or inequalities (Subsection 4.2) on tries and prove the soundness our our trie construction for a context free grammar with respect to the standard inductive semantics of grammars (Subsection 4.3).

4.1. **Proving Equalities on Tries.** Codatatypes are equipped with the perfect tool for proving equalities: the coinduction principle. Intuitively, this principle states that the existence of a relation R that is *closed* under the codatatype's observations (given by selectors) implies that elements related by R are equal. Being closed means here that for all R-related codatatype elements their immediate (non-corecursive) observations are equal and the corecursive observations are again related by R. In other words, if we cannot distinguish elements of a codatatype by (finite) observations, they must be equal. Formally, for our codatatype $\alpha\ lang$ we obtain the following coinduction rule.

```
1  theorem ∅ + L = L
2  proof (rule coinduct_lang)
3      def R L1 L2 = (∃K. L1 = ∅ + K ∧ L2 = K)
4      show R (∅ + L) L by simp
5      fix L1 L2
6      assume R L1 L2
7      then obtain K where L1 = ∅ + K and L2 = K by simp
8      then show o L1 = o L2 ∧ ∀x. R (δ L1 x) (δ L2 x) by simp
9  qed
```

Figure 4: A detailed proof by coinduction

$$\frac{R\ L\ K \qquad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow (o\ L_1 = o\ L_2 \wedge \forall x.\ R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x))}{L = K}\ \mathit{coinduct}_{lang}$$

The second assumption of this rule formalizes the notion of being closed under observations: If two tries are related then their root's labels are the same and all their subtrees are related. A relation that satisfies this assumption is called a *bisimulation*. Thus, proving an equation by coinduction amounts to exhibiting a bisimulation witness that relates the left and the right hand sides.

Let us observe the coinduction rule, which we call $\mathit{coinduct}_{lang}$, in action. Figure 4 shows a detailed proof of the Kleene algebra axiom that the empty language is the left identity of union that is accepted by Isabelle. After applying the coinduction rule backwards (line 2), the proof has three parts. First, we supply a definition of a witness relation R (line 3). Second, we prove that R relates the left and the right hand side (line 4). Third, we prove that R is a bisimulation (lines 5–8). Proving $R\ (\varnothing + L)\ L$ for our particular definition of R is trivial after instantiating the existentially quantified K with L. Proving the bisimulation property is barely harder: for two tries L_1 and L_2 related by R we need to show $o\ L_1 = o\ L_2$ and $\forall x.\ R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x)$. Both properties follow by simple calculations rewriting L_1 and L_2 in terms of a trie K (line 7), whose existence is guaranteed by $R\ L_1\ L_2$, and simplifying with the selector equations for $+$ and $\varnothing$.

$$o\ L_1 = o\ (\varnothing + K) = (o\ \varnothing \vee o\ K) = (\bot \vee o\ K) = o\ K = o\ L_2$$

$$\begin{aligned}&R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x) = R\ (\delta\ (\varnothing + K)\ x)\ (\delta\ K\ x)\\ &= R\ (\delta\ \varnothing\ x + \delta\ K\ x)\ (\delta\ K\ x) = R\ (\varnothing + \delta\ K\ x)\ (\delta\ K\ x)\\ &= (\exists K'.\ \varnothing + \delta\ K\ x = \varnothing + K' \wedge \delta\ K\ x = K') = \top\end{aligned}$$

The last step is justified by instantiating K' with $\delta\ K\ x$.

So in the end, the only part that required ingenuity was the definition of the witness R. But was it truly ingenious? It turns out that in general, when proving a conditional equation $P\ \overline{x} \longrightarrow l\ \overline{x} = r\ \overline{x}$ by coinduction, where $\overline{x}$ denotes a list of variables occurring freely in the equation, the canonical choice for the bisimulation witness is $R\ a\ b = (P\ \overline{x} \wedge \exists \overline{x}.\ a = l\ \overline{x} \wedge b = r\ \overline{x})$. There is no guarantee that this definition yields a bisimulation. However, after performing a few proofs by coinduction, this particular pattern emerges as a natural first choice to try. Indeed, the choice is so natural, that it was worth to automate it in the *coinduction* proof method [6]. This method instantiates the coinduction rule $\mathit{coinduct}_{lang}$ with the canonical bisimulation witness constructed from the goal, where the existentially quantified variables must be given explicitly using the *arbitrary* modifier. Then it applies the instantiated rule in a resolution step, discharges the first assumption, and unpacks the existential quantifiers from R in the remaining subgoal (the `obtain` step in the above proof). Many proofs collapse to an automatic one-line proof using this convenience, including the one above. Some examples follow.

$$\texttt{theorem}\ \varnothing + L = L\ \ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L)\ \mathit{auto}$$

$$\texttt{theorem}\ L + L = L\ \ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L)\ \mathit{auto}$$

$$\texttt{theorem}\ L_1 + L_2 = L_2 + L_1\ \ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L_1\ L_2)\ \mathit{auto}$$

$$\texttt{theorem}\ (L_1 + L_2) + L_3 = L_1 + L_2 + L_3\ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L_1\ L_2\ L_3)\ \mathit{auto}$$

$$\texttt{theorem}\ \mathsf{in}\ (\mathsf{out}\ L) = L\ \ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L)\ \mathit{auto}$$

$$\texttt{theorem}\ \mathsf{in}\ (L \cup K) = \mathsf{in}\ L + \mathsf{in}\ K\ \ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L\ K)\ \mathit{auto}$$

As indicated earlier, sometimes the *coinduction* method does not succeed. It is instructive to consider one example where this is the case: $o\ L \longrightarrow \varepsilon + L = L$.

If we attempt to prove the above statement by coinduction instantiated with the canonical bisimulation witness $R\ L_1\ L_2 = (\exists K.\ L_1 = \varepsilon + K \wedge L_2 = K \wedge o\ K)$, after some simplification we are stuck with proving that R is a bisimulation.

$$\begin{aligned}&R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x) = R\ (\delta\ (\varepsilon + K)\ x)\ (\delta\ K\ x)\\&= R\ (\delta\ \varepsilon\ x + \delta\ K\ x)\ (\delta\ K\ x) = R\ (\varnothing + \delta\ K\ x)\ (\delta\ K\ x)\\&= R\ (\delta\ K\ x)\ (\delta\ K\ x) = (\exists K'.\ \delta\ K\ x = \varepsilon + K' \wedge \delta\ K\ x = K' \wedge o\ K')\end{aligned}$$

The remaining goal is not provable: we would need to instantiate K' with $\delta\ K\ x$, but then, we are left to prove $o\ (\delta\ K\ x)$, which we cannot deduce (we only know $o\ K$). If we, however, instantiate the coinduction rule with $R^{=}\ L_1\ L_2 = R\ L_1\ L_2 \vee L_1 = L_2$, we are able to finish the proof. This means that our canonical bisimulation witness R is not a bisimulation, but $R^{=}$ is. In such cases R is called a *bisimulation up to equality* [32].

Instead of modifying the *coinduction* method to instantiate the rule $\mathit{coinduct}_{\mathit{lang}}$ with $R^{=}$, it is more convenient to capture this improvement directly in the coinduction rule. This results in the following rule which we call *coinduction up to equality* or $\mathit{coinduct}^{=}_{\mathit{lang}}$.

$$\frac{R\ L\ K \qquad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow (o\ L_1 = o\ L_2 \wedge \forall x.\ R^{=}\ (\delta\ L_1\ x)\ (\delta\ L_2\ x))}{L = K}\ \mathit{coinduct}^{=}_{\mathit{lang}}$$

Note that $\mathit{coinduct}^{=}_{\mathit{lang}}$ is not just an instance of $\mathit{coinduct}_{\mathit{lang}}$, with R replaced by $R^{=}$. Instead, after performing this replacement, the first assumption is further simplified to $R\ L\ K$—we would not use coinduction in the first place, if we could prove $R^{=}\ L\ K$ by reflexivity. Similarly, the reflexivity part in the occurrence on the left of the implication in the second assumption is needless and therefore eliminated. The resulting coinduction up to equality principles are independent of the particular codatatypes and thus uniformly produced by the `codatatype` command. Using this coinduction up to equality rule, we have regained the ability to write one-line proofs.

$$\texttt{theorem}\ o\ L \longrightarrow \varepsilon + L = L\ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L\ \mathit{rule}\colon \mathit{coinduct}^{=}_{\mathit{lang}})\ \mathit{auto}$$

One might think that the principle $\mathit{coinduct}^{=}_{\mathit{lang}}$ is always preferable to $\mathit{coinduct}_{\mathit{lang}}$. This is true from the expressiveness point of view: whatever can be proved with $\mathit{coinduct}_{\mathit{lang}}$, can also be proved with $\mathit{coinduct}^{=}_{\mathit{lang}}$. However, for proof automation $\mathit{coinduct}^{=}_{\mathit{lang}}$ is often less beneficial: to prove membership in $R^=$ we need to prove a disjunction which may result in a larger search space, given that neither of the disjuncts is trivially false. In summary, using the weakest rule that suffices to finish the proof helps proof automation.

This brings us to the next hurdle. Suppose that we already have proved the natural selector equations (3.1) for concatenation $\cdot$. (This requires finding some auxiliary properties of $\hat{:}$ and $\hat{\oplus}$ but is an overall straightforward usage of coinduction up to equality.) Next, we want to reason about $\cdot$. For example, we prove its distributivity over $+$: $(L + K) \cdot M = (L \cdot M) + (K \cdot M)$. As before, we are stuck proving that the canonical bisimulation witness $R\ L_1\ L_2 = (\exists L'\ K'\ M'.\ L_1 = (L' + K') \cdot M' \wedge L_2 = (L' \cdot M') + (K' \cdot M'))$ is a bisimulation (and this time even for up to equality).

$$
\begin{aligned}
&R^=\,(\delta\ L_1\ x)\ (\delta\ L_2\ x) = R^=\,(\delta\ ((L' + K') \cdot M')\ x)\ (\delta\ ((L' \cdot M') + (K' \cdot M'))\ x)\\
&= \begin{cases}
R^=\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M') \\
\quad ((\delta\ L'\ x \cdot M') + (\delta\ K'\ x \cdot M')) & \texttt{if } \neg o\ L' \wedge \neg o\ K' \\
R^=\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M' + \delta\ M'\ x) \\
\quad ((\delta\ L'\ x \cdot M' + \delta\ M'\ x) + (\delta\ K'\ x \cdot M')) & \texttt{if } o\ L' \wedge \neg o\ K' \\
R^=\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M' + \delta\ M'\ x) \\
\quad ((\delta\ L'\ x \cdot M') + (\delta\ K'\ x \cdot M' + \delta\ M'\ x)) & \texttt{if } \neg o\ L' \wedge o\ K' \\
R^=\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M' + \delta\ M'\ x) \\
\quad ((\delta\ L'\ x \cdot M' + \delta\ M'\ x) + (\delta\ K'\ x \cdot M' + \delta\ M'\ x)) & \texttt{if } o\ L' \wedge o\ K'
\end{cases}\\
&= \begin{cases}
\top & \texttt{if } \neg o\ L' \wedge \neg o\ K' \\
R^=\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M'\ \boxed{+}\ \delta\ M'\ x) \\
\quad ((\delta\ L'\ x \cdot M' + \delta\ K'\ x \cdot M')\ \boxed{+}\ \delta\ M'\ x) & \text{otherwise}
\end{cases}
\end{aligned}
$$

The remaining subgoal cannot be discharged by the definition of R. In principle it could be discharged by equality (the two tries are equal), but this is almost exactly the property we originally started proving, so we have not simplified the problem by coinduction but rather are going in circles here. However, if our relation could somehow split its arguments across the outermost $+$ highlighted in gray, we could prove the left pair being related by R and the right pair by $=$. The solution is easy: we allow the relation to do just that. Accordingly, we define the congruence closure R^+ of a relation R under $+$ inductively by the following rules.

$$
\frac{L = K}{R^+\ L\ K} \qquad \frac{R\ L\ K}{R^+\ L\ K} \qquad \frac{R^+\ L\ K}{R^+\ K\ L} \qquad \frac{R^+\ L_1\ L_2 \quad R^+\ L_2\ L_3}{R^+\ L_1\ L_3} \qquad \frac{R^+\ L_1\ K_1 \quad R^+\ L_2\ K_2}{R^+\ (L_1 + L_2)\ (K_1 + K_2)}
$$

The closure R^+ is then used to strengthen the coinduction rule, just as the earlier reflexive closure $R^=$ strengthening. Note that the closure $R^=$ can also be viewed as inductively generated by the first two of the above rules. In summary, we obtain *coinduction up to congruence of* $+$, denoted by $\mathit{coinduct}^+_{\mathit{lang}}$.

$$
\frac{R\ L\ K \quad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow (o\ L_1 = o\ L_2 \wedge \forall x.\ R^+\,(\delta\ L_1\ x)\ (\delta\ L_2\ x))}{L = K}\ \mathit{coinduct}^+_{\mathit{lang}}
$$

This rule is easily derived from plain coinduction by instantiating R in $\mathit{coinduct}_{\mathit{lang}}$ with R^+ and proceeding by induction on the definition of the congruence closure.

As intended $coinduct^{+}_{lang}$ makes the proof of distributivity into another automatic one-liner. This is because our new proof principle, coinduction up to congruence of $+$, matches exactly the definitional principle of corecursion up to $+$ used in the selector equations (3.1) of $\cdot$.

$\texttt{theorem}\ (L + K) \cdot M = (L \cdot M) + (K \cdot M)$
$\quad \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L\ K\ M\ \mathit{rule}\colon \mathit{coinduct}^{+}_{lang})\ \mathit{auto}$

Coinduction up to congruence of $+$ allows us also to prove properties of the shuffle product, e.g., commutativity $L \parallel K = K \parallel L$ and associativity $(K \parallel L) \parallel M = K \parallel (L \parallel M)$.

For properties involving iteration *, whose selector equations (3.2) are corecursive up to $\cdot$, we will need a further strengthening of the coinduction rule (using the congruence closure under $\cdot$). Overall, the most robust solution to keep track of the different rules is to maintain a coinduction rule up to all previously defined operations on tries: we define $R^{\bullet}$ to be the congruence closure of R under $+$, $\cdot$, and * and then use the following rule for proving.

$$\frac{R\ L\ K \quad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow (o\ L_1 = o\ L_2 \wedge \forall x.\ R^{\bullet}\ (\delta\ L_1\ x)\ (\delta\ L_2\ x))}{L = K}\ \mathit{coinduct}^{\bullet}_{lang}$$

We will not spell out all equational axioms of Kleene algebra [20] and their formal proofs [35] here. Most proofs are automatic; some require a few manual hints in which order to apply the congruence rules.

4.2. **Proving Inequalities on Tries.** A few axioms of Kleene algebra also contain inequalities, such as $\varepsilon + L \cdot L^* \leq L^*$, and even conditional inequalities, such as $L + M \cdot K \leq M \longrightarrow L \cdot K^* \leq M$. On languages, $L \leq K$ is defined as $L + K = K$, such that in principle proofs by coinduction also are applicable here. However, we can achieve even better proof automation, if we formulate and use the following dedicated coinduction principle for $\leq$.

$$\frac{R\ L\ K \quad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow ((o\ L_1 \longrightarrow o\ L_2) \wedge \forall x.\ R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x))}{L \leq K}\ \mathit{coinduct}_{\leq}$$

This theorem has the same shape as the usual coinduction principle $\mathit{coinduct}_{lang}$, however the relation R is only required to be a simulation instead of a bisimulation. In other words R still needs to be closed under corecursive observations, however the immediate observation of the first argument must only imply the one of the second argument (as opposed to being equal to it). We call this coinduction principle $\mathit{coinduct}_{\leq}$ and prove it by unfolding the definition of $\leq$ and then using the equational coinduction principle $\mathit{coinduct}_{lang}$.

While $\mathit{coinduct}_{\leq}$ allows us to prove simple properties like $r \leq r + s$, it is not strong enough to automatically prove the inequational Kleene algebra axioms, which involve concatenation and iteration. As in the case of equations, up-to reasoning is the familiar way out of this dilemma. However, since R is only a simulation, and thus in general not an equivalence relation, we can not consider its congruence closure. Instead, we follow Rot et al. [29] and define inductively the so-called precongruence closure $R^{\bullet}_{\leq}$.

$$\frac{L \leq K}{R^{\bullet}_{\leq}\ L\ K} \qquad \frac{R\ L\ K}{R^{\bullet}_{\leq}\ L\ K} \qquad \frac{R^{\bullet}_{\leq}\ L_1\ L_2 \quad R^{\bullet}_{\leq}\ L_2\ L_3}{R^{\bullet}_{\leq}\ L_1\ L_3} \qquad \frac{R^{\bullet}_{\leq}\ L_1\ K_1 \quad R^{\bullet}_{\leq}\ L_2\ K_2}{R^{\bullet}_{\leq}\ (L_1 + L_2)\ (K_1 + K_2)}$$

$$\frac{R^{\bullet}_{\leq}\ L_1\ K_1 \quad R^{\bullet}_{\leq}\ L_2\ K_2}{R^{\bullet}_{\leq}\ (L_1 \cdot L_2)\ (K_1 \cdot K_2)} \qquad \frac{R^{\bullet}_{\leq}\ L\ K}{R^{\bullet}_{\leq}\ (L^*)\ (K^*)} \qquad \frac{R^{\bullet}_{\leq}\ L_1\ K_1 \quad R^{\bullet}_{\leq}\ L_2\ K_2}{R^{\bullet}_{\leq}\ (L_1 \cap L_2)\ (K_1 \cap K_2)}$$

With this definition we are able to prove the following strengthened coinduction principle up to precongruence closure, called $\mathit{coinduct}^{\bullet}_{\leq}$.

$$\frac{R\ L\ K \quad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow ((o\ L_1 \longrightarrow o\ L_2) \wedge \forall x.\ R^{\bullet}_{\leq}\ (\delta\ L_1\ x)\ (\delta\ L_2\ x))}{L \leq K}\ \mathit{coinduct}^{\bullet}_{\leq}$$

The proof of $\mathit{coinduct}^{\bullet}_{\leq}$ is structurally very similar to the one of $\mathit{coinduct}^{\bullet}_{\mathit{lang}}$: after using the plain $\mathit{coinduct}_{\leq}$ rule, we are left with proving that the precongruence closure $R^{\bullet}_{\leq}$ is a simulation. This follows by induction on the definition of the precongruence closure. Crucially, the complement operation $\overline{}$ is not included in this definition. For simulation up-to precongruence closure to be a simulation, all operations must be monotone with respect to their immediate observations o, which is not the case for $\overline{}$ [29].

Finally, we are capable to write automatic proofs for inequalities.

$\texttt{theorem}\ \varepsilon + L \cdot L^* \leq L^*\ \ \texttt{by}\ (\mathit{coinduction\ rule}\colon \mathit{coinduct}^{\bullet}_{\leq})\ \mathit{auto}$

$\texttt{theorem}\ o\ K \longrightarrow L \leq L \cdot K\ \ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon L\ \mathit{rule}\colon \mathit{coinduct}^{\bullet}_{\leq})\ \mathit{auto}$

We remark that working with inequalities also has its cost. The reason for this is that Isabelle excels at equational reasoning. Isabelle also provides automation for reasoning with orders, but it is noticeably less powerful than the one for $=$. On the other hand, equations of the form $L + K = K$, which one gets after unfolding the definition of $\leq$, are not ideal for rewriting. Proofs that reduce inequalities to equalities ofter require manual hints to expand K into $L + K$ at the right places. In the end, when using up-to simulations a careful setup of rewriting rules and classical reasoning support for $\leq$ results in a higher degree of automation. This is especially perceivable for more complicated inequational properties like $L + M \cdot K \leq M \longrightarrow L \cdot K^* \leq M$.

4.3. **Reasoning about Context-Free Languages.** We connect our trie construction from Subsection 3.4 for a context-free grammar in weak Greibach normal form to the traditional inductive semantics of context-free grammars. We use the notational conventions and definition of Subsection 3.4, including fixing the starting non-terminal $\mathsf{S} :: \mathcal{N}$ and the productions $\mathsf{P} :: \mathcal{N} \to (\mathcal{T} + \mathcal{N})\ \mathit{list\ set}$. First, we formalize the traditional inductive semantics using an inductive binary predicate $\Subset_{\mathsf{P}} :: \mathcal{T}\ \mathit{list} \to (\mathcal{T} + \mathcal{N})\ \mathit{list} \to \mathit{bool}$ (written infix). Intuitively, $w \Subset_{\mathsf{P}} \alpha$ holds if and only if w is derivable from α in finitely many production steps via P, where each time we replace the leftmost non-terminal first.

$$[] \Subset_{\mathsf{P}} [] \qquad \frac{w \Subset_{\mathsf{P}} \alpha}{(a \# w) \Subset_{\mathsf{P}} (\mathsf{T}\ a \# \alpha)} \qquad \frac{\exists \beta \in \mathsf{P}\ N.\ w \Subset_{\mathsf{P}} \beta +\!\!+ \alpha}{w \Subset_{\mathsf{P}} (\mathsf{NT}\ N \# \alpha)}$$

Note that $\Subset_{\mathsf{P}}$ gives a way to assign the language $\{w \mid w \Subset_{\mathsf{P}} [N]\}$ to each non-terminal N, and in particular the language $\{w \mid w \Subset_{\mathsf{P}} [\mathsf{S}]\}$ for the whole grammar given by P and S. We now prove that our trie G for the fixed grammar represents the same language, i.e., $\mathsf{G} = \mathsf{in}\ \{w \mid w \Subset_{\mathsf{P}} [\mathsf{S}]\}$. Our proof uses an auxiliary intermediate inductive predicate $\Subset^{o\delta}_{\mathsf{P}} :: \mathcal{T}\ \mathit{list} \to (\mathcal{T} + \mathcal{N})\ \mathit{list\ set} \to \mathit{bool}$ (written infix) that reflects the change of the set of states during corecursion in close function (which is used to construct G).

$$\frac{\exists \alpha \in X.\ o_{\mathsf{P}}\ \alpha}{[] \Subset^{o\delta}_{\mathsf{P}} X} \qquad \frac{w \Subset^{o\delta}_{\mathsf{P}} (\bigcup_{\alpha \in X} .\ \delta_{\mathsf{P}}\ \alpha\ a)}{(a \# w) \Subset^{o\delta}_{\mathsf{P}} X}$$

In some sense, $\Subset^{o\delta}_{\mathsf{P}}$ is the inductive view on the close function, as established next.

$$\begin{tikzcd} \alpha \arrow[r, "s"] \arrow[d, "f"'] & \alpha\ F \arrow[d, "\mathsf{map}_F\ f"] \\ \beta \arrow[r, "t"] & \beta\ F \end{tikzcd}$$

Figure 5: Commutation property of a coalgebra morphism

$$\texttt{theorem}\ \mathsf{close}\ X = \mathsf{in}\ \{w \mid w \Subset_{\mathsf{P}}^{o\delta} X\}\ \texttt{by}\ (\mathit{coinduction\ arbitrary}\colon X)\ \mathit{auto}$$

The proof uses the simplest coinduction rule $\mathit{coinduct}_{lang}$ and relies on injectivity of in.

Next, we establish that $w \Subset_{\mathsf{P}} \alpha$ holds if and only if $w \Subset_{\mathsf{P}}^{o\delta} \{\alpha\}$ holds. We prove the two directions separately. Thereby we generalize the "if"-direction.

$$\texttt{theorem}\ w \Subset_{\mathsf{P}}^{o\delta} X \longrightarrow \exists \alpha \in X.\ w \Subset_{\mathsf{P}} \alpha$$
$$\texttt{theorem}\ w \Subset_{\mathsf{P}} \alpha \longrightarrow w \Subset_{\mathsf{P}}^{o\delta} \{\alpha\}$$

We do not show the proofs for the above statements about since both are standard inductions on the inductive definitions of $\Subset_{\mathsf{P}}^{o\delta}$ and $\Subset_{\mathsf{P}}$. Out of all theorems shown in this subsection, only the last one requires the grammar to be in weak Greibach normal form. Putting the three theorems together, we obtain the desired characterization: $\mathsf{G} = \mathsf{in}\ \{w \mid w \Subset_{\mathsf{P}} [\mathsf{S}]\}$.

We remark that a definition of the trie G via a full Earley parser, would not only remove the need of weak Greibach normal form, but also enable coinductive reasoning about arbitrary grammars. For example, one could also formalize and prove correct the translation of grammars to certain normal forms without resorting to the traditional inductive semantics (and thus induction). We leave this study as future work, as it may burst the scope of an introduction to coinduction.

5. Coalgebraic Foundations

We briefly connect the formalized but still intuitive notions, such as tries, from earlier sections with the key coalgebraic concepts and terminology that is usually used to present the coalgebraic view on formal languages. Thereby, we explain how particularly useful abstract objects gave rise to concrete tools in Isabelle/HOL. More theoretical and detailed introductions to coalgebra can be found elsewhere [19, 31].

Given a functor F an (F-)*coalgebra* is a *carrier* object A together with a map $A \to F\ A$—the *structural map* of a coalgebra. In the context of higher-order logic—that is in the category of types which consists of types as objects and of functions between types as arrows—a functor is a type constructor F together with a map function $\mathsf{map}_F :: (\alpha \to \beta) \to \alpha\ F \to \beta\ F$ that preserves identity and composition: $\mathsf{map}_F\ \mathsf{id} = \mathsf{id}$ and $\mathsf{map}_F\ (f \circ g) = \mathsf{map}_F\ f \circ \mathsf{map}_F\ g$. An F-coalgebra in HOL is therefore simply a function $s :: \alpha \to \alpha\ F$. A function $f :: \alpha \to \beta$ is a coalgebra *morphism* between two coalgebras $s :: \alpha \to \alpha\ F$ and $t :: \beta \to \beta\ F$ if it satisfies the commutation property $t \circ f = \mathsf{map}_F\ f \circ s$, also depicted by the commutative diagram in Figure 5.

An (F-)coalgebra to which there exists a unique morphism from any other coalgebra is called a *final* (F-)*coalgebra*. Not all functors F admit a final coalgebra [31, Section 10]. Two different final coalgebras are necessarily isomorphic. Final coalgebras correspond to codatatypes in Isabelle/HOL. Isabelle's facility for defining codatatypes maintains a large class of functors—bounded natural functors [39]—for which a final coalgebra does exists. Moreover, for any bounded natural functor F, Isabelle can construct its final coalgebra with the codatatype

$$\begin{tikzcd} \tau \arrow[r, "s"] \arrow[d, "\mathsf{coiter}\ s"'] & \tau\ D \arrow[d, "\mathsf{map}_D\ (\mathsf{coiter}\ s)"] \\ \alpha\ \mathit{lang} \arrow[r, "\langle o, \delta\rangle"] & \alpha\ \mathit{lang}\ D \end{tikzcd}$$

Figure 6: Unique morphism coiter s to the final coalgebra $(\alpha\ \mathit{lang}, \langle o, \delta\rangle)$

$$\begin{tikzcd} \tau \arrow[r, "s"] \arrow[d, "\mathsf{corec}\ s"'] & (\alpha\ \mathit{lang} + \tau)\ D \arrow[d, "\mathsf{map}_D\ [\mathsf{id}, \mathsf{corec}\ s]"] \\ \alpha\ \mathit{lang} \arrow[r, "\langle o, \delta\rangle"] & \alpha\ \mathit{lang}\ D \end{tikzcd}$$

Figure 7: Characteristic theorem of the corecursor

CF as the carrier and define a bijective constructor $\mathsf{C}_F :: CF\ F \to CF$ and its inverse, the destructor $\mathsf{D}_F :: CF \to CF\ F$. The latter takes the role of the structural map of the coalgebra.

`codatatype` $CF = \mathsf{C}_F\ (\mathsf{D}_F : CF\ F)$

Finally, we are ready to connect these abstract notions to our tries. The codatatype of tries $\alpha\ \mathit{lang}$ is the final coalgebra of the functor $\beta\ D = \mathit{bool} \times (\alpha \to \beta)$ with the associated map function $\mathsf{map}_D\ g = \mathsf{id} \times (\lambda f.\ g \circ f)$, where $(f \times g)\ (x,\ y) = (f\ x,\ g\ y)$. The structural map of this final coalgebra is the function $\mathsf{D}_D = \langle o,\ \delta\rangle$, where $\langle f,\ g\rangle\ x = (f\ x,\ g\ x)$.

The finality of $\alpha\ \mathit{lang}$ gives rise to the definitional principles of primitive coiteration and corecursion. In Isabelle the coiteration principle is embodied by the primitive *coiterator* $\mathsf{coiter} :: (\tau \to \tau\ D) \to \tau \to \alpha\ \mathit{lang}$, that assigns to the given D-coalgebra the unique morphism from itself to the final coalgebra. In other words, the primitive coiterator allows us to define functions of type $\tau \to \alpha\ \mathit{lang}$ by providing a D-coalgebra on τ, i.e., a function of type $\tau \to \mathit{bool} \times (\alpha \to \tau)$ that essentially describes a deterministic (not necessarily finite) automaton without an initial state. To clarify this automaton analogy, it is customary to present the F-coalgebra s as two functions $s = \langle \mathsf{o},\ \mathsf{d}\rangle$ with τ being the states of the automaton, $\mathsf{o} : \tau \to \mathit{bool}$ denoting accepting states, and $\mathsf{d} : \alpha \to \tau \to \tau$ being the transition function. From a given s, we uniquely obtain the function $\mathsf{coiter}\ s$ that assigns to a separately given initial state $t : \tau$ the language $\mathsf{coiter}\ s\ t : \alpha\ \mathit{lang}$ and makes the diagram in Figure 6 commute. Note that Figure 6 is an instance of Figure 5.

Corecursion differs from coiteration by additionally allowing the user to stop the coiteration process by providing a fixed non-corecursive value. In Isabelle this is mirrored by another combinator: the *corecursor* $\mathsf{corec} :: (\tau \to (\alpha\ \mathit{lang} + \tau)\ D) \to \tau \to \alpha\ \mathit{lang}$ where the sum type $+$ offers the possibility either to continue corecursively as before (represented by the type τ) or to stop with a fixed value of type $\alpha\ \mathit{lang}$. The corecursor satisfies the characteristic property shown in Figure 7, where the square brackets denote a case distinction on $+$, i.e. $[f,\ g]\ x = \mathtt{case}\ x\ \mathtt{of}\ \mathsf{Inl}\ l \Rightarrow f\ l \mid \mathsf{Inr}\ r \Rightarrow g\ r$. Corecursion is not more expressive than coiteration (since corec can be defined in terms of coiter), but it is more convenient to use. For instance, the non-corecursive specifications of ε and A, and the `else` branch of $\hat{\cdot}$ exploit this additional flexibility.

The `primcorec` command [6] reduces a user specification to a non-recursive definition using the corecursor. For example, the union operation $+$ is internally defined as $\lambda L\ K.\ \mathsf{corec}\ (\lambda(L, K).\ (o\ L \wedge o\ K,\ \lambda a.\ \mathsf{Inr}\ (\delta\ L\ a,\ \delta\ K\ a)))\ (L, K)$. The D-coalgebra argument to corec resembles the right hand sides of the selector equations for $+$ (with the corecursive calls replaced by Inr). In fact, for this simple definition mere coiteration would suffice. An example that uses the convenience that corecursion provides, is the deferred concatenation $\hat{\cdot}$. Its internal definition reads: $\lambda L\ K.\ \mathsf{corec}\ (\lambda(L, K).\ (o\,L \wedge o\,K,\ \lambda(x, b).\ \texttt{if}\ b\ \texttt{then}\ \mathsf{Inr}\ (\delta\ L\ x,\ K)\ \texttt{else if}\ o\,L\ \texttt{then}\ \mathsf{Inr}\ (\delta\ K\ x,\ \varepsilon)\ \texttt{else}\ \mathsf{Inl}\ \varnothing)$. As end users, most of the time we are happy being able to write the high-level corecursive specifications, without having to explicitly supply coalgebras.

It is worth noting that the final coalgebra $\alpha\ \mathit{lang}$ itself corresponds to the automaton, whose states are languages, acceptance is given by $\mathsf{o}\ L = o\ L$, and the transition function by $\mathsf{d}\ a\ L = \delta\ L\ a$. For these definitions, we obtain $\mathsf{corec}\ \langle \mathsf{o}, \mathsf{d}\rangle\ (L : \alpha\ \mathit{lang}) = L$. For regular languages this automaton corresponds to the minimal automaton (since equality on tries corresponds to language equivalence), which is finite by the Myhill–Nerode theorem. This correspondence is not very practical though, since we typically label states of automata with something finite, in particular not with languages (represented by infinite tries).

A second consequence of the finality of $\alpha\ \mathit{lang}$ is the coinduction principle that we have seen earlier. It follows from the fact that final coalgebras are quotients by bisimilarity, where bisimilarity is defined as the existence of a bisimulation relation.

6. Discussion and Related Work

Our development is a formalized counterpart of Rutten's introduction to the coalgebraic view on languages [30]. In this section we discuss further related work.

Coalgebraic View on Formal Languages. The coalgebraic approach to languages has recently received some attention. Landmark results in language theory were rediscovered and generalized. Silva's recent survey [34] highlights some of those results including the proofs of correctness of Brzozowski's subtle deterministic finite automaton minimization algorithm [8]. The coalgebraic approach yields some algorithmic advantages, too. Bonchi and Pous present a coinductive algorithm for checking equivalence of non-deterministic automata that outperforms all previously known algorithms by one order of magnitude [9]. Another recent development is our formally verified coalgebraic algorithm for deciding weak monadic second-order logic of one successor (WS1S) [36]. This formalization employs the Isabelle library presented here.

Formal Languages in Proof Assistants. The traditional set-of-words view on formal languages is formalized in most proof assistants. In contrast, we are not aware of any other formalization of the coalgebraic view on formal languages in a proof assistant.

Here, we want to compare our formalization with the Isabelle incarnation of the set-of-words view developed by Krauss and Nipkow for the correctness proof of their regular expression equivalence checker [22]. Both libraries are comparably concise. In 500 lines Krauss and Nipkow prove almost all axioms of Kleene algebra and the characteristic equations for the left quotients (the δ-specifications in our case). They reuse Isabelle's libraries for sets and lists, which come with carefully tuned automation setup. Still, their proofs tend to require additional induction proofs of auxiliary lemmas, especially when reasoning about iteration. Our formalization is 700 lines long. We prove all axioms of Kleene algebra and connect our representation to the set-of-words view via the bijections out and in. Except for those bijections our formalization does not rely on any other library. Moreover, when we changed our 5000 lines long formalization of a coalgebraic decision procedure for WS1S [36] to use the infinite tries instead of the set-of-words view, our proofs about WS1S became approximately 300 lines shorter. Apparently, a coalgebraic library is a good fit for a coalgebraic procedure.

Paulson presents a concise formalization of automata theory based on hereditarily finite sets [26]. For the semantics he reuses Krauss and Nipkow's set-of-words formalization.

Non-Primitive Corecursion in Proof Assistants. Automation for corecursion in proof assistants is much less developed than its recursive counterpart. The Coq proof assistant supports corecursion up to constructors [11]. Looking at our examples, however, this means that Coq will not be able to prove productivity of the natural concatenation and iteration specifications automatically, since both go beyond up-to constructors. Instead, our reduction to primitive corecursion can be employed to bypass Coq's productivity checker.

Recently, we have added the support for corecursion up to so called friendly operations to Isabelle/HOL [5, 7]. (Before this addition, Isabelle supported only primitive corecursion [6].) An operation is friendly if, under lazy evaluation, it does not peek too deeply into its arguments, before producing at least one constructor. For example, the friendly operation $L + K = \mathsf{L}\ (o\ L \lor o\ K)\ (\lambda x.\ \delta\ L\ x + \delta\ K\ x)$ destructs only one layer of constructors, in order to produce the topmost L. In contrast, the primitively corecursive equation $\mathsf{deep}\ L = \mathsf{L}\ (o\ L)\ (\lambda x.\ \mathsf{deep}\ (\delta\ (\delta\ L\ x)\ x))$ destructs two layers of constructors (via δ) before producing one and is therefore not friendly. Indeed, we will not be able to reduce the equation $\mathsf{bad} = \mathsf{L}\ \top\ (\lambda\_.\ \mathsf{deep}\ \mathsf{bad})$ (which is corecursive up to deep) to a primitively corecursive specification. And there is a reason for it: bad is not uniquely specified by the above equation, or in other words not productive.

Since $+$ is friendly, and $\cdot$ and $\|$ are corecursive up to $+$, this new infrastructure allows us to use the constructor view version of the natural selector equations (3.1) and (3.3) for $\cdot$ and $\|$ instead of the more complicated primitively corecursive definitions from Section 3.

$$\begin{array}{l}\texttt{corec (friend)}\ + :: \alpha\ lang \to \alpha\ lang \to \alpha\ lang\ \texttt{where}\\ \quad L + K = \mathsf{L}\ (o\ L \lor o\ K)\ (\lambda x.\ \delta\ L\ x + \delta\ K\ x)\\ \texttt{corec (friend)}\ \cdot :: \alpha\ lang \to \alpha\ lang \to \alpha\ lang\ \texttt{where}\\ \quad L \cdot K = \mathsf{L}\ (o\ L \land o\ K)\ (\lambda x.\ (\delta\ L\ x \cdot K) + (\texttt{if}\ o\ L\ \texttt{then}\ \delta\ K\ x\ \texttt{else}\ \varnothing))\\ \texttt{corec (friend)}\ \| :: \alpha\ lang \to \alpha\ lang \to \alpha\ lang\ \texttt{where}\\ \quad L \parallel K = \mathsf{L}\ (o\ L \land o\ K)\ (\lambda x.\ (\delta\ L\ x \parallel K) + (L \parallel \delta\ K\ x))\end{array}$$

The `corec` command defines the specified constants and the `friend` option registers them as friendly operations by automatically discharging the emerging proof obligations ensuring that the operations consume at most one constructor to produce one constructor. Since $\cdot$ is friendly, too, we can define the corecursive up to $\cdot$ iteration * using its natural equations (3.2).

$$\begin{array}{l}\texttt{corec (friend)}\ \_^* :: \alpha\ lang \to \alpha\ lang\ \texttt{where}\\ \quad L^* = \mathsf{L}\ \top\ (\lambda x.\ \delta\ L\ x \cdot L^*)\end{array}$$

Internally, `corec` reduces the corecursive specification to a primitively corecursive one following an abstract, category theory inspired construction. Yet, what this abstract construction yields in practice is relatively close to our manual construction for concatenation. (In contrast, the iteration case takes some shortcuts, which the abstract view does not offer.)

On the reasoning side, `corec` provides some automation, too. It automatically derives the corresponding coinduction up-to rules for the registered (sets of) friendly operations. Overall, the usage of `corec` compresses our development from 700 to 550 lines of Isabelle text.

Agda's combination of copatterns (i.e., destructor view) and sized types [3, 4] is the most expressive implemented support for corecursion in proof assistants to date. However, using sized types often means that one has to encode proofs of productivity manually in the type of the defined function. Thus, it is possible to define concatenation and iteration using their natural equations (3.1) and (3.2) in Agda. Recently, Abel [1,2] has formalized those definitions of regular operations up to proving the recursion equation $L^* = \varepsilon + L \cdot L^*$ for iteration in 219 lines of Agda text, which correspond to 125 lines in our version. His definitions are equally concise as the ones using `corec`, but his proofs require more manual steps.

7. Conclusion

We have presented a particular formal structure for computation and deduction: infinite tries modeling formal languages. Although this representation is semantic and infinite, it is suitable for computation—in particular we obtain a matching algorithm for free on tries constructed by regular operations. Deduction does not come short either: coinduction is the convenient reasoning tool for infinite tries. Coinductive proofs are concise, especially for (in)equational theorems such as the axioms of Kleene algebra.

Codatatypes might be just the right tool for thinking algorithmically about semantics. We hope to have contributed to their dissemination by outlining some of their advantages.

Acknowledgments. I thank Andrei Popescu for introducing me to the coinductive view on formal languages. I am grateful to Tobias Nipkow and David Basin for encouraging this work. Jasmin Blanchette, Andrei Popescu, Ralf Sasse, and the anonymous FSCD and LMCS reviewers gave many precious hints on early drafts of this article helping to improve both the presentation and the content. Jurriaan Rot suggested important related work and recommended proving inequalities of tries by coinduction on $\leq$ rather than on $=$.

References

[1] A. Abel. Compositional coinduction with sized types. In I. Hasuo, ed., *CMCS 2016*, vol. 9608 of *LNCS*, pp. 5–10. Springer, 2016.
[2] A. Abel. Equational reasoning about formal languages in coalgebraic style. Available online at: `http://www2.tcs.ifi.lmu.de/~abel/jlamp17.pdf`, 2016.
[3] A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In G. Morrisett and T. Uustalu, eds., *ICFP '13*, pp. 185–196. ACM, 2013.
[4] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In R. Giacobazzi and R. Cousot, eds., *POPL'13*, pp. 27–38. ACM, 2013.
[5] J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with benefits: Implementing corecursion in foundational proof assistants. In H. Yang, ed., *ESOP 2017*, vol. 10201 of *LNCS*, pp. 111–140. Springer, 2017.
[6] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, eds., *ITP 2014*, vol. 8558 of *LNCS*, pp. 93–110. Springer, 2014.
[7] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion. In J. Reppy, ed., *ICFP 2015*, pp. 192–204. ACM, 2015.
[8] F. Bonchi, M. M. Bonsangue, J. J. M. M. Rutten, and A. Silva. Brzozowski's algorithm (co)algebraically. In R. L. Constable and A. Silva, eds., *Logic and Program Semantics - Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*, vol. 7230 of *LNCS*, pp. 12–23. Springer, 2012.
[9] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In R. Giacobazzi and R. Cousot, eds., *POPL 2013*, pp. 457–468. ACM, 2013.
[10] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
[11] A. Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[12] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, eds., *TYPES'93*, vol. 806 of *LNCS*, pp. 62–78. Springer, 1994.
[13] J. Earley. An efficient context-free parsing algorithm. *C. ACM*, 13(2):94–102, 1970.
[14] E. Giménez and P. Castéran. A tutorial on [co-]inductive types in Coq. `http://www.labri.fr/perso/casteran/RecTutorial.pdf`, 1998.
[15] S. A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.
[16] F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, eds., *ITP 2013*, vol. 7998 of *LNCS*, pp. 100–115. Springer, 2013.
[17] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, eds., *FLOPS 2010*, vol. 6009 of *LNCS*, pp. 103–117. Springer, 2010.
[18] R. Hinze. Concrete stream calculus: An extended study. *J. Funct. Program.*, 20(5-6):463–535, 2010.
[19] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
[20] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994.
[21] D. Kozen and A. Silva. Practical coinduction. Tech. report, Cornell University, 2012. `http://hdl.handle.net/1813/30510`.
[22] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, 49:95–106, 2012. published online March 2011.
[23] A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. In G. Klein and R. Gamboa, eds., *ITP 2014*, vol. 8558 of *LNCS*, pp. 341–357. Springer, 2014.
[24] T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014. `http://www.in.tum.de/~nipkow/Concrete-Semantics`.
[25] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Logic Comp.*, 7(2):175–204, 1997.
[26] L. C. Paulson. A formalisation of finite automata using hereditarily finite sets. In A. P. Felty and A. Middeldorp, eds., *CADE-25*, vol. 9195 of *LNCS*, pp. 231–245. Springer, 2015.
[27] T. Rendel, J. Trieflinger, and K. Ostermann. Automatic refunctionalization to a language with copattern matching: with applications to the expression problem. In K. Fisher and J. H. Reppy, eds., *ICFP 2015*, pp. 269–279. ACM, 2015.
[28] J. Rot, M. M. Bonsangue, and J. J. M. M. Rutten. Coinductive proof techniques for language equivalence. In A. H. Dediu, C. Martín-Vide, and B. Truthe, eds., *LATA 2013*, vol. 7810 of *LNCS*, pp. 480–492. Springer, 2013.
[29] J. Rot, M. M. Bonsangue, and J. J. M. M. Rutten. Proving language inclusion and equivalence by coinduction. *Inf. Comput.*, 246:62–76, 2016.
[30] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorgi and R. de Simone, eds., *CONCUR 1998*, vol. 1466 of *LNCS*, pp. 194–218. Springer, 1998.
[31] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.
[32] D. Sangiorgi. On the bisimulation proof method. *J. Math. Structures in Comp. Sci.*, 8:447–479, 1998.
[33] A. Setzer. How to reason coinductively informally. To appear in R. Kahle, T. Strahm, and T. Studer (Eds.): Festschrift on occasion of Gerhard Jäger's 60th birthday. Available from http://www.cs.swan.ac.uk/∼csetzer/articles/jaeger60Birthdaymain.pdf, 2015.
[34] A. Silva. A short introduction to the coalgebraic method. *ACM SIGLOG News*, 2(2):16–27, 2015.
[35] D. Traytel. A codatatype of formal languages. In *Archive of Formal Proofs*. 2013. `http://devel.isa-afp.org/entries/Coinductive_Languages.shtml`.
[36] D. Traytel. A coalgebraic decision procedure for WS1S. In S. Kreutzer, ed., *CSL 2015*, vol. 41 of *LIPIcs*, pp. 487–503. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2015.
[37] D. Traytel. *Formalizing Symbolic Decision Procedures for Regular Languages*. PhD thesis, TU München, 2015.
[38] D. Traytel. Formal languages, formally and coinductively. In D. Kesner and B. Pientka, eds., *FSCD 2016*, vol. 52 of *LIPIcs*, pp. 31:1–31:17. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2016.
[39] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS 2012*, pp. 596–605. IEEE Computer Society, 2012.

[40] P. Wadler. The Expression Problem. Available online at: `http://tinyurl.com/wadler-ep`, 1998.
[41] J. Winter, M. M. Bonsangue, and J. J. M. M. Rutten. Context-free languages, coalgebraically. In A. Corradini, B. Klin, and C. Cîrstea, eds., *CALCO 2011*, vol. 6859 of *LNCS*, pp. 359–376. Springer, 2011.