Equational Reasoning about Formal Languages in Coalgebraic Style

Andreas Abel

Department of Computer Science and Engineering, Gothenburg University

Abstract

Formal languages and automata are a foundational topic of computer science, with many practical applications such as compiler construction, textual search, model checking, and decidability of certain logics. Automata are an instance of transitions systems which have the structure of a coalgebra, and coalgebraic and coinductive reasoning tools such as simulations, bisimulations, and up-to techniques have been successfully employed to study formal languages.

In this paper, we show how routine reasoning about formal languages can be carried out with just the coinductive notion of equality of languages aka bisimilarity. We formalize a coinductive type of languages and the coinductive type family of strong bisimilarity of languages in the proof assistant Agda using sized types. The sized typing enables us to establish algebraic properties of language operations through coinductive proofs of bisimilarity by equational reasoning. In particular, after verifying that formal languages form a Kleene algebra, the laws of the Kleene algebra are sufficient to prove correctness of the usual constructions on automata.

Keywords: Automaton, coalgebra, coinduction, copattern,
formal language, sized types, type theory.
2010 MSC: 03B15, 68N18, 68Q45, 68Q70

It is recommended to print this article in color.

1. Introduction

Formal languages and automata are a foundational topic of computer science, with many practical applications such as compiler construction, textual

Preprint submitted to JLAMP *December 23, 2016*

search, model checking, and decidability of certain logics. While the theory of formal languages and automata goes back to the 1960s, its connection to the coalgebras of category theory is more recent. This coalgebraic formulation suggests that *coinduction* should be central for reasoning about formal languages and automata. In particular, to show that two automata recognize the same language, one can find a bisimulation between these automata and then use coinduction in form of the theorem of Knaster and Tarski (1955) for the greatest fixed point. However, coming up with a suitable bisimulation is so far an act of creativity and has not been automated, although remedies like parameterized coinduction (Hur et al., 2013) have been suggested for a smoother experience of formal coinductive reasoning. Further, a set of standard strategies to find a bisimulation have become known as *coinduction-up-to techniques* (Pous and Sangiorgi, 2012).

In this article, we demonstrate that a generalization of primitive coinduction to *well-founded coinduction* as both a definition principle and a proof principle lets us carry through a substantial part of basic automata theory with just *equational reasoning* over coinductive equality (bisimilarity). Technically, we rely on a formulation of coinduction with sized types (Hughes et al., 1996; Amadio and Coupet-Grimal, 1998; Barthe et al., 2004; Abel, 2008; Sacchini, 2013; Abel and Pientka, 2013) in the context of Martin-Löf Type Theory (1975). This enables us to define the Kleene algebra operations of formal languages elegantly via their Brzozowski derivatives (1964). Further, we can define coinductive language equality in a way that gives us the ability to prove theorems by coinduction and equation chains. In particular, it allows us to apply the coinductive hypothesis under the proof term for transitivity. The latter is not possible in weaker formulations of coinduction such as the *Calculus of (Co)inductive Constructions* underlying the Coq (2016) proof assistant, which makes use of an untyped guardedness check (Coquand, 1994). As an alternative to sized types, well-founded coinduction could probably be based on ordered families of equivalences (Gianantonio and Miculan, 2003) or ultrametric spaces. A type theory utilizing these approaches is emerging (Bizjak et al., 2016), but has not seen a mature implementation yet.

Agda (2016) is currently the only type-theoretic proof assistant with support for sized types. Albeit still experimental, Agda's sized types let us formalize decidable languages and automata elegantly through definition by copattern matching (Abel et al., 2013). All the definitions, theorems, and proofs of this paper have been extracted from Agda code via an Agda to

LaTeX translation and are, thus, guaranteed to be correct (assuming the consistency of Agda itself).

This article does not present any new or surprising results about formal languages or automata but is solely concerned about the techniques for formal definition and proof. It should, thus, be seen as a tutorial. It has been inspired by Rutten's tutorial on coalgebraic techniques for formal languages (1998) and Traytel's recent formalizations in Isabelle (2016). Traytel reports that Isabelle is being extended by *friendly* coinductive definitions that bring part of the reasoning power of sized types, possibly sufficient to mimic the reasoning style employed in this article.

Overview. In Section 2 we briefly recapitulate the bits and pieces of Type Theory and Agda relevant for this article. In Section 3 we define decidable languages as infinite tries and some operations on them in the style of Brzozowski derivatives. Subsequently, in Section 4 we prove the Kleene algebras for these operations. The final technical section, 5, is devoted to the corresponding constructions on automata and their correctness.

2. Preliminaries: Type Theory and Agda

The definitions and theorems of this article are formulated in Dependent Type Theory à la Martin-Löf (1975), in particular, in the Agda (2016) language. Agda is an implementation of Type Theory with several extensions; we will most notably make use of *sized types* (Hughes et al., 1996; Abel and Pientka, 2013) to express coinductive types and definitions by coinduction.

On the one hand, Agda is a dependently-typed purely functional programming language, and on the other hand, thanks to the Curry-Howard correspondence, a proof assistant for constructive logic. In the following, we will briefly introduce the syntax of Agda by example. The experienced Agda user can safely skip the remainder of this section.

2.1. Agda as a dependently-typed functional language

The core features of Agda are inductive families (Dybjer, 1994) and functions defined by pattern matching. A very simple inductive family is the enumeration type Bool with two constructors true and false.

```
data Bool : Set where
  true  :  Bool
  false : Bool
```

Bool itself has type Set, which is universe or a type of types, but not all types, to avoid vicious cycles. For instance Set itself does not have type Set, but inhabits the next universe Set$_1$. Boolean negation can be defined by simple pattern matching.

```
not : Bool → Bool
not true  = false
not false = true
```

Agda also supports Unicode and infix and mixfix operators. For example, we can define boolean disjunction like this:

```
_∨_ : (a b : Bool) → Bool
true  ∨ b = true
false ∨ b = b
```

The notation $(a\ b : \mathsf{Bool}) \to \mathsf{Bool}$ is short for $(a : \mathsf{Bool})(b : \mathsf{Bool}) \to \mathsf{Bool}$ or $(a : \mathsf{Bool}) \to (b : \mathsf{Bool}) \to \mathsf{Bool}$, which is the syntax for dependent function types. In this case, there is no actual dependency, since the bound variables a and b are not subsequently used in the type.

Data types can be parameterized over other types A. For instance Maybe A embeds an arbitrary type A via just and extends it by a new value nothing.

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

Data types can be recursive, such as List i A parameterized by a size i and a type A. The size i : Size acts as an upper bound on the length of the list. The special size ∞ : Size means unbounded length.

```
data List (i : Size) (A : Set) : Set where
  []   : List i A
  _::_ : {j : Size< i} (x : A) (xs : List j A) → List i A
```

A list is either empty (constructor []); then any size i is an upper bound on its length. Or, the list is nonempty (e. g., $x :: xs$); then, if j is an upper bound on the length of its tail xs and $j < i$, then i is an upper bound on the length of $x :: xs$. Strictly speaking, the constructor $\_::\_$ takes three arguments: $j : \mathsf{Size}\!< i$ and $x : A$ and $xs : \mathsf{List}\ A\, j$, but the first argument j is in {braces} and thus declared as hidden. The user does not have to write it, and its value will be inferred by Agda if possible. Note that j occurs in the type of xs, thus, this is a proper dependency.

It is possible to supply a hidden argument to a function by enclosing it in braces. In case of the infix operator $\_::\_$, we have to fall back to prefix style $\_::\_\ \{j\}\, x\, xs$ though.

The types Size and $\mathsf{Size}\!<\ i$ are Agda primitives that are used for termination checking. For instance, consider the mapping function on lists. Note that the notation $\forall\{i\, A\, B\} \to \ldots$ is short for $\{i : \_\}\{A : \_\}\{B : \_\} \to \ldots$ and becomes $\{i : \mathsf{Size}\}\{A : \mathsf{Set}\}\{B : \mathsf{Set}\} \to \ldots$ after type reconstruction.

```
map : ∀{i A B} → (A → B) → List i A → List i B
map f []          =  []
map f (x :: xs)   =  f x :: map f xs
```

Function $\mathsf{map}\, f$ takes a list of size i as input and returns a list with the same upper bound. We say map is *size preserving*. As map is defined recursively, termination is not obvious. Agda infers from pattern $x :: xs : \mathsf{List}\ i\, A$, which is short for pattern $\_::\_\ \{j\}\, x\, xs : \mathsf{List}\ i\, A$ with a pattern variable j introduced by Agda, that $xs : \mathsf{List}\ j\, A$ and $j : \mathsf{Size}\!<\! i$. Consequently, the recursive call $\mathsf{map}\, f\, xs$—which internally expands to $\mathsf{map}\,\{j\}\, f\, xs$— is justified, by the descent in size $j < i$. An analogous argument assures termination of foldr, the iteration principle for lists, which replaces in a list the $[]$ constructor by $n : B$ and any $\_::\_$ constructor by $c : A \to B \to B$.

```
foldr : ∀{i} {A B : Set} → (A → B → B) → B → List i A → B
foldr c n []          =  n
foldr c n (x :: xs)   =  c x (foldr c n xs)
```

As an application of foldr and map, we define $\mathsf{any}\, p\, xs$ which is true if $p\, x$ is true for any element x of xs.

```
any : ∀{i A} → (A → Bool) → List i A → Bool
any p xs = foldr _∨_ false (map p xs)
```

Finally, data types with a single constructor can alternatively be defined as record types. For instance, the Cartesian product $A \times B$ can be implemented as record type with the projections fst : $A \times B \to A$ and snd : $A \times B \to B$ and constructor _,_ : $A \to B \to A \times B$.

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field fst  :  A
        snd  :  B
```

Agda allows the projections also on the left hand sides of definitions by pattern matching. In the following, we define for a pair $p : A \times B$ is reversal swap p : $B \times A$ by giving its value for all valid projections. This definition form is called *definition by copattern matching*, since we are not matching on a function argument, but on the possible observations on the function result (Abel et al., 2013).

```
swap : ∀{A B} → A × B → B × A
fst  (swap p)  =  snd p
snd  (swap p)  =  fst p
```

This is, of course, just one possible implementation of swap, which we chose to exemplify copattern matching. A simple clause swap $(a\,,\,b) = (b\,,\,a)$ would have done the job, but copattern matching will be the definition principle of choice for coinductive structures in Section 3.

2.2. Agda as a proof assistant

Martin-Löf Type Theory allows us to reason about program via the propositions-as-types paradigm. A proposition is seen as the type of its proofs, for instance, the absurd proposition ⊥ *(Falsehood)* has no proof, and the trivial proposition ⊤ *(Truth)* has a proof with no further content.

In Agda, ⊥ is modeled as a data type with no constructors. Given a proof p : ⊥, we can prove any proposition A (populate any type A) by matching on p. As there are no constructors of ⊥, there is nothing further to show, indicated in Agda by the absurd pattern () which matches anything of empty type.

```
data ⊥ : Set where

⊥-elim : {A : Set} (p : ⊥) → A
⊥-elim ()
```

Truth ⊤ is modeled as record type with no fields. There is no information to extract from a proof of ⊤, a proof is simply an empty record.

```
record ⊤ : Set where

triv : ⊤
triv = record {}
```

Implication $A \to B$ coincides with the ordinary function space, and universal quantification $\forall x \to A$ with the dependent function space $(x : \_) \to A$.

We can define our own proposition, predicates, and relation as inductive (or coinductive, see Section 4.1) families. The prime example is *propositional equality* $x \equiv y$ of objects $x, y : A$, which is defined as a data type with a hidden parameter A : Set, a visible parameter $x : A$, and an index $y : A$. The only constructor refl—which has no arguments—fixes y to be identical to x, thus, witnesses that x and y are identical modulo Agda's internal notion of equality (which is called *definitional equality*).

```
data _≡_ {A : Set} (x : A) : (y : A) → Set where
  refl : x ≡ x
```

Since defined inductively, propositional equality is the smallest relation on A which is reflexive.

Proof of equality can be used by pattern matching. For instance, we can prove symmetry of equality, i. e., $x \equiv y$ implies $y \equiv x$ by pattern matching on the the proof of $x \equiv y$.

```
sym : ∀{A} {x y : A} → x ≡ y → y ≡ x
sym {A} {x} {.x} refl = refl
```

The only matching constructor refl forces y to be identical to x. This is indicated in Agda by the inaccessible pattern $.x$ which means that y has been instantiated by the term x. As a consequence, the goal becomes $x \equiv x$ which is simply proved by refl.

In a similar fashion, we prove transitivity of propositional equality. By matching both the proofs of $x \equiv y$ and $y \equiv z$ against refl, variables y and z become instantiated to x, and again, the goal becomes simply $x \equiv x$.

```
trans : ∀{A} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

In general, inductively defined propositions are inhabited by proof trees in the same way that inductive types are inhabited by trees. As an example, consider the proposition $\mathsf{Any}\,i\,P\,xs$ which states that predicate $P : A \to \mathsf{Set}$ holds on some element $x : A$ of (unbounded) list $xs : \mathsf{List}\ \infty\,A$. Parameter $i : \mathsf{Size}$ is an upper bound on the tree height of a proof $p : \mathsf{Any}\,i\,P\,xs$ of this proposition.

```
data Any (i : Size) {A} (P : A → Set) : List ∞ A → Set where
  here  : ∀{x xs}                (p : P x)         →  Any i P (x :: xs)
  there : ∀{x xs}  {j : Size< i}  (p : Any j P xs)  →  Any i P (x :: xs)
```

Constructor here establishes the proposition for a non-empty list $x :: xs$ given a proof $p : Px$ that predicate P holds on the head x of the list. The resulting proof tree is a leaf, and any size i is an upper bound on the height of this derivation. Constructor there takes a derivation $p : \mathsf{Any}\,j\,P\,xs$ stating that P holds on some element of list xs, and builds a proof of $\mathsf{Any}\,i\,P\,(x :: xs)$. The tree height of p, ordinal j, is necessarily strictly smaller than i.

Proofs in Type Theory naturally contain the necessary information to construct witnesses for existential propositions such as Any. In this case, a proof takes the form $\mathsf{there}^n\,(\mathsf{here}\,p)$ where n is the index of witnessing element x that satisfies P, and $p : P\,x$ is the evidence for the latter fact.

This concludes the short tutorial on Type Theory and Agda. In the next section, we introduce coinductive types for the example of infinitely deep trees. In the following, we will sometimes write $\mathsf{List}\ A$ for unbounded lists $\mathsf{List}\ \infty\,A$.

3. Decidable Languages, Coinductively

Given an alphabet A, a word $as \in \mathsf{List}\ A$ is a list of characters. A language over A is usually described as set of words, and a decidable language is one whose characteristic function $\mathsf{List}\ A \to \mathsf{Bool}$ is computable. We will work in the setting of Type Theory (Martin-Löf, 1975) where each function is computable, thus, we can identify decidable languages with functions of type $\mathsf{List}\ A \to \mathsf{Bool}$ where Bool is the two-element data type with constructors true and false.

A set of words with decidable membership can also be represented as *trie*. For our purposes, a trie is an A-branching tree whose nodes are labeled by booleans. Any word as is a path into the tree selecting a subtree. The root label of that subtree indicates the status of the word as. Label true means the word is member of the set, label false means it is not a member. Even though each word is finite, the language might be infinite, thus, tries have infinite depth in general.

For instance, let us consider the language E of even natural numbers in binary representation. Writing 0 as a and 1 as b, our language contains the words a, ba, baa, bba, $baaa$, $baba$, etc. Given the alphabet $A = \{a, b\}$, the language can be concisely described by the regular expression $a + b(a + b)^*a$.

Figure 1 shows an initial part of the trie of language E, where double-circled nodes denote membership of the word leading to that node, and single-circled ones non-membership. Observe that the subtree ba has only accepting nodes along its left-most path a^*, witnessing that ba^* is a sublanguage of E (representing 2^n for $n \geq 1$). Thus, a finite trie would be insufficient to represent E.

Generalizing Bool to B, the connection between the type T of A-branching B-labeled tries and the type $\mathsf{List}\, A \to B$ can also be derived by calculating type isomorphisms (Hinze, 2000; Altenkirch, 2001):

$$\begin{array}{rcl} \mathsf{List}\, A \to B & \cong & (1 + A \times \mathsf{List}\, A) \to B \\ & \cong & (1 \to B) \times ((A \times \mathsf{List}\, A) \to B) \\ & \cong & B \times (A \to (\mathsf{List}\, A \to B)) \end{array}$$

This means that $\mathsf{List}\, A \to B$ is a solution of the recursive equation

$$X \cong B \times (A \to X)$$

which also describes the decomposition of a trie X into its root label of type B and its A-indexed family of subtrees $A \to X$. Tries T are the greatest

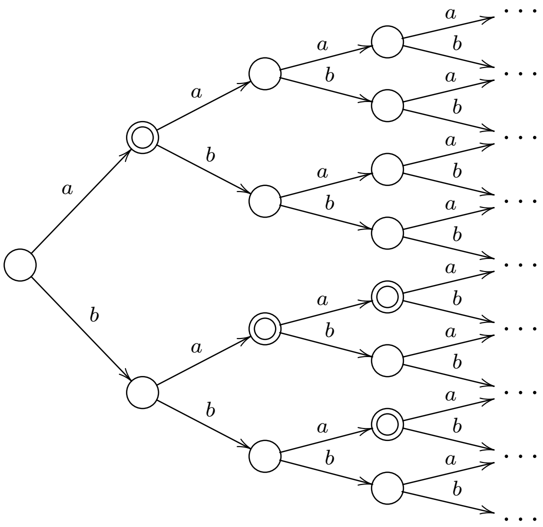


Figure 1: Trie of E

solution of this equation and we write $T = \nu X.\, B \times (A \to X)$. We will later establish the isomorphism between T and $\mathsf{List}\, A \to B$ more precisely.

3.1. Coinductive tries in Agda

In Agda, we represent the coinductive type $\nu X.\, \mathsf{Bool} \times (A \to X)$ of tries as a coinductive record type Lang with fields $\nu : \mathsf{Bool}$ for the root label and $\delta : A \to \mathsf{Lang}$ for the family of subtrees. If field ν is true then the language contains the empty word, which is sometimes called a *nullable* language, hence the field name ν.

The name δ is inspired by its role as *Brzozowski derivative*. Given a decidable language $f : \mathsf{List}\, A \to \mathsf{Bool}$, its a-derivative $\delta\, f\, a : \mathsf{List}\, A \to \mathsf{Bool}$ is defined as $(\delta\, f\, a)(as) = f\,(a :: as)$. This means that $\delta\, f\, a$ accepts the words of f that start with a, minus this first letter. In terms of tries t, we obtain the derivative $\delta\, t\, a$ simply by following the a-labeled edge from the root, thus, the derivation function is identical with the field δ.

There is one final twist to arrive at the Agda definition: In order to facilitate corecursive definitions of tries that are certified by Agda's productivity checker, we equip the type of tries $\mathsf{Lang}\, i$ with an index $i : \mathsf{Size}$. Index i denotes an ordinal $\leq \omega$ corresponding to the *definedness depth* of a trie

$t : \mathsf{Lang}\,i$. Ultimately, we are interested only in fully defined tries $t : \mathsf{Lang}\,\infty$, where ∞ is syntax for ordinal ω. This means we can query t's nodes at arbitrary depth. For a finite definedness level i we can only inspect nodes up to depth i. In particular $t : \mathsf{Lang}\,0$ allows us to look only at the root label $\nu\,t$, its subtrees via $\delta\,t\,a$ are undefined and Agda's type checker will object to such an expression.

If for an arbitrary ordinal i, a trie $t : \mathsf{Lang}\,i$ can be defined by reference to tries of type $\mathsf{Lang}\,j$ for $j < i$, written $j : \mathsf{Size}{<}\,i$, then t can be assigned type $\forall\{i\} \to \mathsf{Lang}\,i$. We say t is defined by well-founded recursion on ordinal i, which is our principle of corecursive definition.

```
record Lang i : Set where
  coinductive
  field ν  :  Bool
        δ  :  ∀{j : Size< i} → A → Lang j
```

As typing of the projections from tries we get

$$\begin{aligned} \nu &: \forall\{i : \mathsf{Size}\} \to \mathsf{Lang}\,i \to \mathsf{Bool} \\ \delta &: \forall\{i : \mathsf{Size}\} \to \mathsf{Lang}\,i \to \forall\{j : \mathsf{Size}{<}\,i\} \to A \to \mathsf{Lang}\,j \end{aligned}$$

with hidden size arguments which will, if type checking succeeds, be figured out by Agda's unifier and size constraint solver. When taking the derivative $\delta\,\{i\}\,t\,\{j\}\,a$ we are free to choose any j strictly below i. This expresses that if t was only defined up to depth i, then its a-subtree is less defined; and we are allowed to waste information and chose a smaller j than necessary. Wasting is fine since $\mathsf{Lang}\,i$ is a subtype of $\mathsf{Lang}\,j$ whenever $i \geq j$, as we can always use a *more* defined value $t : \mathsf{Lang}\,i$ when a value of $\mathsf{Lang}\,j$ is demanded. The antitone subtyping chain

$$\mathsf{Lang}\,\infty \leq \ldots \leq \mathsf{Lang}\,(\uparrow i) \leq \mathsf{Lang}\,i \leq \ldots \mathsf{Lang}\,0$$

can be justified by the equation $\mathsf{Lang}\,i \cong \mathsf{Bool} \times \bigcap_{j<i}(A \to \mathsf{Lang}\,j)$ which holds semantically.

The isomorphism $\mathsf{Lang}\,i \cong \mathsf{List}\ i\,A \to \mathsf{Bool}$ is witnessed by the following two functions. The first, $l \ni as$, checks membership of word as in language l represented as a trie. The empty word $[]$ is in the language if $\nu\,l$, i.e., if the language is nullable. The composite word $a :: as$ is accepted by l if as is accepted by the derivative $\delta\,l\,a$.

```
_∋_ : ∀{i} → Lang i → List i A → Bool
l ∋ []        = ν l
l ∋  a :: as  = δ l a ∋ as
```

Visually spoken, $l \ni as$ returns the root label of the subtree selected by path as.

The second function constructs a trie representation trie f from the functional representation f of a decidable language. The trie is constructed corecursively by copattern matching (Abel et al., 2013).

```
trie : ∀{i} (f : List i A → Bool) → Lang i
ν (trie f)     =  f []
δ (trie f) a   =  trie (λ as → f (a :: as))
```

The root label ν (trie f) is determined by whether f accepts the empty word $[\,]$. The a-derivative δ (trie f) a is constructed by corecursion on the a-derivative of f. The justification of the recursive call to trie is apparent once we make the hidden size arguments visible:

$$\delta\ \{i\}\ (\mathsf{trie}\ \{i\}\ f)\ \{j\}\ a = \mathsf{trie}\ \{j\}\ (\lambda as \to f\ (a :: as))$$

By typing of the projection δ, we have j : Size< i, thus, the definition of trie $\{i\}$ only rests on trie $\{j\}$ with a smaller size index. Well-founded induction on sizes guarantees that the equation system has a unique solution.

The corecursive definition by copattern matching is sometimes likened to differential equations (Hansen et al., 2016). In the definition of trie, the second equation (δ) is the differential equation, and the first equation (ν) determines the initial value.

3.2. Constructing decidable languages by coiteration

In the following, we implement some standard constructions on formal languages by copattern matching. These operations will allow us to compute the trie of any regular expression. Throughout the rest of this article, we assume a type A of characters with a decidable equality $\lfloor\ a \overset{?}{=} b\ \rfloor$: Bool for a, b : A.

The empty language $\emptyset$ is the trie where each node label is false. Naturally, each subtree of $\emptyset$ is again $\emptyset$.

$$
\begin{aligned}
&\emptyset : \forall\{i\} \to \mathsf{Lang}\ i\\
&\nu\ \emptyset \quad\ = \mathsf{false}\\
&\delta\ \emptyset\ x = \emptyset
\end{aligned}
$$

The language ε accepting only the empty word has root label true but all other labels are false. Hence, any derivative is the empty language.

$$
\begin{aligned}
&\varepsilon : \forall\{i\} \to \mathsf{Lang}\ i\\
&\nu\ \varepsilon \quad\ = \mathsf{true}\\
&\delta\ \varepsilon\ x = \emptyset
\end{aligned}
$$

The language char a accepting the one-letter word $a :: [\,]$ is not nullable, its a-derivative is ε and all other derivatives are $\emptyset$.

$$
\begin{aligned}
&\mathsf{char} : \forall\{i\}\ (a : \mathsf{A}) \to \mathsf{Lang}\ i\\
\\
&\nu\ \ (\mathsf{char}\ a) \quad\ = \mathsf{false}\\
&\delta\ \ (\mathsf{char}\ a)\ \ x = \mathsf{if}\ \lfloor\, a \overset{?}{=} x \,\rfloor\ \mathsf{then}\ \ \varepsilon\ \ \mathsf{else}\ \ \emptyset
\end{aligned}
$$

We obtain the language complement compl l of a language l by flipping all labels. This is accomplished by recursing over the whole tree.

$$
\begin{aligned}
&\mathsf{compl} : \forall\{i\}\ (l : \mathsf{Lang}\ i) \to \mathsf{Lang}\ i\\
&\nu\ (\mathsf{compl}\ l) \quad\ = \mathsf{not} \quad\ (\nu\ l)\\
&\delta\ (\mathsf{compl}\ l)\ x = \mathsf{compl}\ \ (\delta\ l\ x)
\end{aligned}
$$

Complement compl is a special instance of mapping a function pointwise over all tree labels.

For the union $k \cup l$ of two languages l and k, we overlay the two tries and perform the Boolean disjunction operation on corresponding node labels.

$$
\begin{aligned}
&\_\cup\_ : \forall\{i\}\ (k\ l : \mathsf{Lang}\ i) \to \mathsf{Lang}\ i\\
&\nu\ (k \cup l) \quad\ = \nu\ k \quad \vee\ \ \nu\ l\\
&\delta\ (k \cup l)\ x = \delta\ k\ x\ \cup\ \delta\ l\ x
\end{aligned}
$$

The intersection could be defined analogously, using Boolean conjunction.

Both operations are instance of a general $\mathsf{zipWith}$-function that applies a binary operation pointwise to a pair of tries.

All recursive definitions of tries so far have followed a specific pattern: in the right hand sides of the recursive equations, the recursive call was outermost, i. e., the equation had the form $\delta\,(g\,\vec{y})\,x = g\,\vec{t}$ for some variables $x, \vec{y}$ and some terms $\vec{t}$. With the non-recursive equation being $\nu\,(g\,\vec{y}) = o$, this form is an instance of the commutative diagram for terminal coalgebras and sometimes called *coiteration* (Geuvers, 1992).

For a functor $F : \mathsf{Set} \to \mathsf{Set}$, an F-coalgebra is a pair (S, t) with $S : \mathsf{Set}$ and $t : S \to F\,S$. An F-coalgebra morphism between coalgebras (S_1, t_1) and (S_2, t_2) is a function $f : S_1 \to S_2$ such that $t_2\,(f\,s_1)$ is equal to $F\,f\,(t\,s_1)$ for all $s_1 : S_1$. An F-coalgebra is terminal if it is the target of a coalgebra morphism from every F-coalgebra. Besides establishing the connection between coiteration and coalgebras, we will not dwell on coalgebras in this article, thus, we do not go into more details here.

Here is the diagram for a $(\mathsf{Bool} \times (A \to \_))$-coalgebra (Γ, h) mapping into the terminal coalgebra $(\mathsf{Lang}, \langle\nu, \delta\rangle)$:

$$\begin{tikzcd}
\Gamma \arrow[r, "h"] \arrow[d, dotted, "\mathsf{coit}\,h"'] & \mathsf{Bool} \times (A \to \Gamma) \arrow[d, "\mathsf{id} \times (\mathsf{coit}\,h \circ \_)"] \\
\mathsf{Lang} \arrow[r, "\langle\nu, \delta\rangle"] & \mathsf{Bool} \times (A \to \mathsf{Lang})
\end{tikzcd}$$

With $g := \mathsf{coit}\,h$, the commutative law

$$\langle\nu, \delta\rangle \circ g = \mathsf{id} \times (g \circ \_) \circ h$$

can be applied to points $\vec{y} : \Gamma$ to yield

$$\langle\nu, \delta\rangle(g\,\vec{y}) = (\mathsf{id} \times (g \circ \_))(h\,\vec{y}).$$

For our instance, $h\,\vec{y} = (o,\ \lambda x \to \vec{t})$ with $\vec{y}{:}\Gamma \vdash o : \mathsf{Bool}$ and $\vec{y}{:}\Gamma, x{:}A \vdash t : \mathsf{Lang}$, thus,

$$\langle\nu, \delta\rangle(g\,\vec{y}) = (o,\ g \circ (\lambda x \to \vec{t})).$$

This can be split into the two equations

$$\begin{array}{lcl}
\nu\,(g\,\vec{y}) & = & o \\
\delta\,(g\,\vec{y})\,x & = & g\,\vec{t}
\end{array}$$

that form the laws of a function $g = \mathsf{coit}\ (\lambda\vec{y} \to (o,\ \lambda x \to \vec{t}))$ defined by coiteration (modulo some tupling and (un)currying).

The type/context Γ can be interpreted as the set of states of an automaton h with a coupled presentation of the accepting state set $\Gamma \to \mathsf{Bool}$ and the transition function $\Gamma \to (A \to \Gamma)$. Function $\mathsf{coit}\,h$ maps a state $s : \Gamma$ to the language $\mathsf{coit}\,h\,s$ accepted by h starting from state s. The discussed language constructions correspond to constructions of (possibly infinite) automata with references to existing automata as oracles. The reader is invited to confirm this by expressing the given constructions through coiteration. Note however, that the state type Γ might involve Lang and is, thus, not guaranteed to be finite!

3.3. Constructing decidable languages by well-founded corecursion

To complete the constructions of languages as supported by regular expressions, we are missing language concatenation and the Kleene star. These can be constructed by *corecursion up-to* which can be reduced to primitive corecursion into a trie with an extended alphabet (Traytel, 2016). However, using sized types we can naturally define these operations by their derivative laws, using well-founded recursion on sizes.

Language concatenation $k \cdot l$ is our first non-trivial operation on languages. The intuition $(k \ni as) \land (l \ni bs) \implies (k \cdot l) \ni (as \mathbin{+\!\!+} bs)$ leads to the specification $(k \cdot l) \ni cs \iff \exists n \in \mathbb{N}.\ k \ni (\mathsf{take}\,n\,cs) \land l \ni (\mathsf{drop}\,n\,cs)$.[1] However, this specification does not directly suggest a pretty implementation of $k \cdot l$ (Doczkal et al., 2013).

We can instead try to understand language concatenation as an operation on the tries k and l. If we think about accepting a word cs in $k \cdot l$ by following paths in k and l, the following procedure applies: We start by following branches in k. Whenever we reach an accepting node in k we may decide that we have reached the boundary between the words as in k and bs in l that make up the word $cs = (as \mathbin{+\!\!+} bs)$ in $k \cdot l$. Hence, we start following branches in l. However, since we are not sure we already reached the boundary, we simultaneously continue to follow branches in k. At each accepting node in k we spawn off a run in l. Thus, a trie for $k \cdot l$ may be constructed by the following operation on all accepting nodes of k:

[1] We write $as \mathbin{+\!\!+} bs$ for the concatenation of lists as and bs; we write $\mathsf{take}\,n\,cs$ for the largest prefix of cs of length $\leq n$, and $\mathsf{drop}\,n\,cs$ for the remainder. Note that $cs = \mathsf{take}\,n\,cs \mathbin{+\!\!+} \mathsf{drop}\,n\,cs$ for any $n \in \mathbb{N}$.

make the node non-accepting but then union the subtree starting here with l. This transformation is achieved by the following corecursive definition of concatenation:

```
_·_ : ∀{i} (k l : Lang i) → Lang i
ν (k · l)     =  ν k ∧ ν l
δ (k · l) x   =  let  k′l = δ k x · l  in  if  ν k  then  k′l ∪ δ l x  else  k′l
```

The concatenation of two languages is nullable iff both are nullable. For the x-derivative, we follow the x-branch in k via $\delta\,k\,x \cdot l$ in any case. If the node is accepting, i. e., $\nu\,k$ is true, we may in addition follow the x-branch in l via $\delta\,l\,x$. As before, the equations for language concatenation correspond to the derivation laws of regular expressions (Brzozowski, 1964), but we arrived there by the trie intuition.

The above definition is not an instance of coiteration for two reasons: First, the outermost call is to if_then_else_ rather than the recursive call $k'l$. Even if we consider if_then_else_ to be special (rather than just an arbitrary Agda function), there is still a recursive call $k'l$ in the then-branch which is not at top-level, but under the union-operator. This problem is usually fixed by defining a scheme for corecursion up to union. However, looking at the involved sizes we can accept the definition in the present form as an instance of well-founded corecursion. Crucial here is the sized typing of the union

$$\_\cup\_ : \forall\{i\}(k\,l : \mathsf{Lang}\,i) \to \mathsf{Lang}\,i$$

which asserts that the arguments are no deeper analyzed than the definedness depth of the result. If we make all hidden size arguments visible—having to switch to prefix operators instead of infix ones—we can see the propagation of definedness depth levels to the recursive call $k'l$.

```
_·_ : ∀{i} (k l : Lang i) → Lang i

δ (_·_{i} k l) {j} x =
   let  k′l : Lang j
        k′l = _·_{j} (δ k {j} x) l
   in   if  ν k  then  _∪_{j} k′l (δ l {j} x)  else  k′l
```

Since the recursive call happens at smaller index $j < i$, it is justified. Note also that in the definition of $k'l$, last letter, $l : \mathsf{Lang}\, i$ is cast to $\mathsf{Lang}\, j$ which is valid since $j < i$.

The iteration l^* of a language l, aka *Kleene star*, can be informally described as "zero or more repetitions of l". If for some $n \geq 0$ we have words $as_1, as_2, \ldots as_n \in l$, then $(as_1 ++ as_2 ++ \ldots ++ as_n) \in l$. In terms of tries, l^* is obtained from l by making the root accepting and unioning l with any subtree of l that has an accepting root. Intuitively, this means that at each accepting node we may "jump back" to the root. The corecursive definition

```
_* : ∀{i} (l : Lang i) → Lang i
ν (l *)    =  true
δ (l *) x  =  δ l x · (l *)
```

relies on the sized typing of concatenation to justify the recursive call.

This concludes our set of language operations defined by well-founded corecursion. These operations allow us to give an executable semantics for regular expressions (leaving aside efficiency questions). It may be remarked that, thanks to sized typing, all the definitions are concise and direct counterparts of the derivative laws for regular expressions (Brzozowski, 1964).

4. Proving the Kleene Algebra Laws

In this section, we prove that decidable languages as introduced in Section 3 form a *Kleene algebra*.

4.1. A family of equivalence relations over languages

Equality of tries, sometimes called *strong bisimilarity*, is defined coinductively as follows. Two tries are strongly bisimilar if they have the same root and corresponding subtries are strongly bisimilar in turn. In Agda, this amounts to the following coinductive definition:

```
record _≅⟨_⟩≅_ (l : Lang ∞) i (k : Lang ∞) : Set where
  coinductive
  field ≅ν  :  ν l ≡ ν k
        ≅δ  :  ∀{j : Size< i} (a : A) → δ l a ≅⟨ j ⟩≅ δ k a
```

For one, note that we are relating tries $l, k : \mathsf{Lang}\,\infty$ whose definedness depth is unbounded (∞). This means that any subtrie such as $\delta\, l\, a$ is defined and in turn has type $\mathsf{Lang}\,\infty$.

However, the relation itself is indexed by a definedness depth i. In fact we are defining a family of types such that $l \cong\langle\, j \,\rangle\cong k$ is a subtype of $l \cong\langle\, i \,\rangle\cong k$ whenever $i \leq j$. The depth is a lower bound on how far the proof of equality of l and k is constructed. In particular, we can only inspect the derivative $\cong\!\delta\, p\, a$ of a proof $p : l \cong\langle\, i \,\rangle\cong k$ if $i > 0$. As for coinductive types like $\mathsf{Lang}\, i$, the size index i is just a tool for the corecursive construction of derivations. Ultimately, we are only interested in fully defined equality proofs $p : l \cong\langle\, \infty \,\rangle\cong k$. In particular, our size-index relation is not to be confused with *ordered families of equivalences* (OFEs) (Gianantonio and Miculan, 2003) $l \equiv_n k$ which refine the notion of equality itself. (There, $l \equiv_0 k$ would hold always and $l \equiv_{n+1} k$ would hold if l and k have equal roots and their immediate subtries are $\equiv_n$-related.) OFEs are a different approach to justifying corecursive definitions.

Each of the coinductive relations forms an equivalence relation, proven for the whole family by coiteration. For reflexivity, we have to prove that given a trie l, we can construct a derivation that l is strongly bisimilar to itself l, up to arbitrary depth i.

```
≅refl : ∀{i} {l : Lang ∞} → l ≅⟨ i ⟩≅ l
≅ν  ≅refl    =  refl
≅δ  ≅refl a  =  ≅refl
```

The proof ≅refl of $l \cong\langle\, i \,\rangle\cong l$ is constructed lazily. If we are asking for its first component $\cong\!\nu$ ≅refl we get a proof that the root $\nu\, l$ is identical to itself, namely refl $: \nu\, l \equiv \nu\, l$. If we are asking for the a-branch of its second component, $\cong\!\delta$ ≅refl a at depth $j < i$, it computes ≅refl $: l \cong\langle\, j \,\rangle\cong l$ corecursively.

Symmetry is defined in a similar fashion. To compute a proof of $k \cong l$ up to depth i, we only need a derivation of $l \cong k$ up to depth i; thus, the type of ≅sym is $l \cong\langle\, i \,\rangle\cong k \to k \cong\langle\, i \,\rangle\cong l$.

```
≅sym : ∀{i} {k l : Lang ∞} (p : l ≅⟨ i ⟩≅ k) → k ≅⟨ i ⟩≅ l
≅ν  (≅sym p)    =  sym   (≅ν p)
≅δ  (≅sym p) a  =  ≅sym  (≅δ p a)
```

Transitivity is likewise depth preserving. Depth-preservation is crucial to combine reasoning by transitivity and the coinductive hypothesis in a natural way, as we will see below.

```
≅trans : ∀{i} {k l m : Lang ∞}
   (p : k ≅⟨ i ⟩≅ l) (q : l ≅⟨ i ⟩≅ m) → k ≅⟨ i ⟩≅ m
≅ν  (≅trans p q)    =  trans   (≅ν p)    (≅ν q)
≅δ  (≅trans p q) a  =  ≅trans  (≅δ p a)  (≅δ q a)
```

Taken together, each _≅⟨ i ⟩≅_ is an equivalence relation, and forms a setoid Bis i with carrier Lang ∞.

```
≅isEquivalence : ∀(i : Size) → IsEquivalence (λ l k → l ≅⟨ i ⟩≅ k)
≅isEquivalence i = record { refl = ≅refl; sym = ≅sym; trans = ≅trans }

Bis : ∀(i : Size) → Setoid _ _
Setoid.Carrier         (Bis i)  =  Lang ∞
Setoid._≈_             (Bis i)  =  λ l k → l ≅⟨ i ⟩≅ k
Setoid.isEquivalence   (Bis i)  =  ≅isEquivalence i
```

Later, we will use these setoids to reason by equality chains. Equality chains are not a built-in feature of Agda, but a module of its standard library. An equality chain allows to write down equational reasoning in a human-readable way, and is basically a nice interface to reasoning by transitivity. In general, it works for any preorder, i. e., any reflexive-transitive relation.

Just for the sake of demonstration, we prove transitivity of bisimilarity again, using the old transitivity proof in form of an equality chain.

```
≅trans′ : ∀ i (k l m : Lang ∞)
   (p : k ≅⟨ i ⟩≅ l) (q : l ≅⟨ i ⟩≅ m) → k ≅⟨ i ⟩≅ m
≅trans′ i k l m p q = begin
   k  ≈⟨ p ⟩
   l  ≈⟨ q ⟩
   m  ∎ where open EqR (Bis i)
```

As a prerequisite, we bring the primitives of equality chains into scope by opening module EqR (short for EquationalReasoning) instantiated to the setoid Bis i. A chain then starts with begin followed with the first term of the chain (k). Then follows a justification ($p : k \cong\langle i \rangle\cong l$) for equality with the second term (l). This may repeat for a while, in our case, there is only another justification ($q : l \cong\langle\, i \,\rangle\cong m$) and a final term ($m$). The chain closes with an end-of-proof maker (■).

4.2. Laws of language union

Decidable languages form an idempotent commutative monoid under union. The individual laws, like associativity, commutativity, idempotency, and unit, follow from the corresponding laws of the boolean disjunction, which are pointwise applied at all the corresponding nodes of the involved tries. In Agda, these are direct proofs by coiteration.

```
union-assoc : ∀{i} (k {l m} : Lang ∞) → (k ∪ l) ∪ m ≅⟨ i ⟩≅ k ∪ (l ∪ m)
≅ν  (union-assoc k)      =  ∨-assoc (ν k) _ _
≅δ  (union-assoc k) a    =  union-assoc (δ k a)

union-comm : ∀{i} (l k : Lang ∞) → l ∪ k ≅⟨ i ⟩≅ k ∪ l
≅ν  (union-comm l k)      =  ∨-comm (ν l) _
≅δ  (union-comm l k) a    =  union-comm (δ l a) (δ k a)

union-idem : ∀{i} {l : Lang ∞} → l ∪ l ≅⟨ i ⟩≅ l
≅ν  union-idem      =  ∨-idempotent _
≅δ  union-idem a    =  union-idem

union-emptyˡ : ∀{i} {l : Lang ∞} → ∅ ∪ l ≅⟨ i ⟩≅ l
≅ν  union-emptyˡ      =  refl
≅δ  union-emptyˡ a    =  union-emptyˡ
```

Finally, union preserves equality, which is again proven by coiteration. The sized typing will be crucial to apply a coinductive hypothesis under union-cong later.

```
union-cong : ∀{i}{k k′ l l′ : Lang ∞}
   (p : k ≅⟨ i ⟩≅ k′) (q : l ≅⟨ i ⟩≅ l′) → k ∪ l ≅⟨ i ⟩≅ k′ ∪ l′
≅ν  (union-cong p q)      =  cong₂ _∨_  (≅ν p)    (≅ν q)
≅δ  (union-cong p q) a    =  union-cong  (≅δ p a)  (≅δ q a)
```

A derived law we require later is that union distributes over itself. Now that we have established that union fulfills the laws of an idempotent commutative monoid, we can use a solver to prove this law automatically by reflection.

```
union-union-distr : ∀{i} (k {l m} : Lang ∞) →
  (k ∪ l) ∪ m ≅⟨ i ⟩≅ (k ∪ m) ∪ (l ∪ m)
```

Concretely, the solver checks that both sides of the equation have the same set of atoms, by normalizing both sides to the set $\{k, l, m\}$. This solver is implemented in Agda itself, but we will not describe it further here.[2]

4.3. Laws of language concatenation

In this section, we prove laws of language concatenation $k \cdot l$. Since it is defined by cases on whether k is nullable, we will make the same case distinction in most proofs. To this end, we use Agda's with construct, as for example in:

```
withExample : (P : Bool → Set) (p : P true) (q : P false) →
  {A : Set} (g : A → Bool) (x : A) → P (g x)
withExample P p q g x with g x
... | true  =  p
... | false =  q
```

It can be roughly seen as a case distinction over $g\,x$, but it also abstracts $g\,x$ in the goal $P\,(g\,x)$ so that we can solve it by $p : P$ true in the first clause and $q : P$ false in the second clause.

Further, we use Agda's rewrite construct, which can be applied on an equation $l \equiv r$ to rewrite subterms l in a goal to r. For example:

```
rewriteExample : {A : Set} {P : A → Set} {x : A} (p : P x)
  {g : A → A} (e : g x ≡ x) → P (g x)
rewriteExample p e rewrite e = p
```

Here, the goal is changed from $P\,(g\,x)$ to $P\,x$ using equation e, and subsequently solved by p.

Concatenation distributes over union, for instance, $k \cdot (l \cup m) \cong (k \cdot l) \cup (k \cdot m)$. Naturally, we would like to prove this by coinduction. The case for

[2] https://github.com/agda/agda-stdlib/blob/1c78e4e/src/Algebra/IdempotentCommutativeMonoidSolver.agda implements this solver.

ν follows by the boolean distributivity law $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$. In the case for δ, we would like to reason by the following equality chain. We consider the subcase that k is nullable, and underline the subterms that have changed from the last line (unless the whole expression has changed).

$$\begin{array}{ccr}
\delta\,(k \cdot (l \cup m))\,a & \cong & \text{by definition} \\
\delta\,k\,a \cdot (l \cup m) \cup \delta\,(l \cup m)\,a & \cong & \text{by definition} \\
\delta\,k\,a \cdot (l \cup m) \cup \underline{(\delta\,l\,a \cup \delta\,m\,a)} & \cong & \text{by coinduction hypothesis} \\
\underline{(\delta\,k\,a \cdot l \cup \delta\,k\,a \cdot m)} \cup (\delta\,l\,a \cup \delta\,m\,a) & \cong & \text{by union laws} \\
(\delta\,k\,a \cdot l \cup \underline{\delta\,l\,a}) \cup (\underline{\delta\,k\,a \cdot m} \cup \delta\,m\,a) & \cong & \text{by definition} \\
\underline{\delta\,(k \cdot l)\,a} \cup \underline{\delta\,(k \cdot m)\,a} & \cong & \text{by definition} \\
\delta\,(k \cdot l \cup k \cdot m)\,a & &
\end{array}$$

This proof does not follow the scheme of (primitive) coinduction. The coinduction hypothesis is applied under uses of transitivity (for connecting the equations) and under the congruence law for union. This becomes especially clear if we fully write out the justifications as in the corresponding Agda proof in Figure 2. However, the continuity of transitivity and union-cong^{l} as witnessed by the sized typing justifies the use of the coinduction hypothesis.

The other distributivity law is proven by coinduction and case distinction over the nullability of l and k.

```
concat-union-distribˡ : ∀{i} (k {l m} : Lang ∞) →
  (k ∪ l) · m ≅⟨ i ⟩≅ (k · m) ∪ (l · m)
```

Congruence laws for concatenation follow by coinduction and congruence of union.

```
concat-congˡ : ∀{i} {m l k : Lang ∞}
  → l       ≅⟨ i ⟩≅ k
  → l  ·  m ≅⟨ i ⟩≅ k  ·  m

concat-congʳ : ∀{i} {m l k : Lang ∞}
  →       l ≅⟨ i ⟩≅       k
  → m  ·  l ≅⟨ i ⟩≅ m  ·  k
```

```
concat-union-distribʳ : ∀{i} (k {l m} : Lang ∞) →
  k · (l ∪ m) ≅⟨ i ⟩≅ (k · l) ∪ (k · m)

≅ν (concat-union-distribʳ k) = ∧-∨-distribˡ (ν k) _ _
≅δ (concat-union-distribʳ k) a with ν k

≅δ (concat-union-distribʳ k {l} {m}) a | true = begin
    δ k a · (l ∪ m) ∪ (δ l a ∪ δ m a)
  ≈⟨ union-congˡ (concat-union-distribʳ (δ k a)) ⟩
    (δ k a · l ∪ δ k a · m) ∪ (δ l a ∪ δ m a)
  ≈⟨ union-swap24 ⟩
    (δ k a · l ∪ δ l a) ∪ (δ k a · m ∪ δ m a)
  ∎
  where open EqR (Bis _)

≅δ (concat-union-distribʳ k) a | false = concat-union-distribʳ (δ k a)
```

Figure 2: Concatenation distributes over union.

The coinductive proof of associativity relies on distributivity and congruence and associativity of union.

```
concat-assoc : ∀{i} (k {l m} : Lang ∞) → (k · l) · m ≅⟨ i ⟩≅ k · (l · m)
```

Finally, the empty language is a zero and the language of the empty word a unit for language composition:

```
concat-emptyˡ  : ∀{i} l → ∅ · l ≅⟨ i ⟩≅ ∅
concat-emptyʳ  : ∀{i} l → l · ∅ ≅⟨ i ⟩≅ ∅

concat-unitˡ  : ∀{i} l → ε · l ≅⟨ i ⟩≅ l
concat-unitʳ  : ∀{i} l → l · ε ≅⟨ i ⟩≅ l
```

4.4. Laws of the Kleene star

The language of the empty word is the iteration of the empty language.

```
star-empty : ∀{i} → ∅ * ≅⟨ i ⟩≅ ε
```

To prove that iteration is idempotent, we first prove that concatenation of iterated languages is idempotent.

```
star-concat-idem : ∀{i} (l : Lang ∞) → l * · l * ≅⟨ i ⟩≅ l *
≅ν (star-concat-idem l)     =  refl
≅δ (star-concat-idem l) a  =  begin
      δ l a · l * · l * ∪ δ l a · l *
   ≈⟨ union-congˡ (concat-assoc _) ⟩
      δ l a · (l * · l *) ∪ δ l a · l *
   ≈⟨ union-congˡ (concat-congʳ (star-concat-idem _)) ⟩
      δ l a · l * ∪ δ l a · l *
   ≈⟨ union-idem ⟩
      δ l a · l *
   ∎ where open EqR (Bis _)
```

This lets us prove idempotency of the Kleene star:

```
star-idem : ∀{i} (l : Lang ∞) → (l *) * ≅⟨ i ⟩≅ l *
≅ν (star-idem l)     =  refl
≅δ (star-idem l) a  =  begin
   δ l a · l * · (l *) *  ≈⟨ concat-congʳ (star-idem l) ⟩
   δ l a · l * · l *      ≈⟨ concat-assoc (δ l a) ⟩
   δ l a · (l * · l *)    ≈⟨ concat-congʳ (star-concat-idem l) ⟩
   δ l a · l *
   ∎ where open EqR (Bis _)
```

The Kleene star obeys the following recursive equation:

```
star-rec : ∀{i} (l : Lang ∞) → l * ≅⟨ i ⟩≅ ε ∪ (l · l *)
```

Finally, we prove Arden's rule (1961), which would allow us to solve linear equations over regular expressions.

```
star-from-rec : ∀{i} (k {l m} : Lang ∞)
   → ν k ≡ false
   → l ≅⟨ i ⟩≅ k · l ∪ m
   → l ≅⟨ i ⟩≅ k * · m
```

```
≅ν (star-from-rec k n p) with ≅ν p
... | b rewrite n = b

≅δ (star-from-rec k {l} {m} n p) a with ≅δ p a
... | q rewrite n = begin
      (δ l a)
    ≈⟨ q ⟩
      δ k a · l ∪ δ m a
    ≈⟨ union-congˡ (concat-congʳ (star-from-rec k {l} {m} n p)) ⟩
      (δ k a · (k * · m) ∪ δ m a)
    ≈⟨ union-congˡ (≅sym (concat-assoc (δ k a))) ⟩
      (δ k a · k * · m ∪ δ m a)
    ∎ where open EqR (Bis _)
```

All the proofs about decidable languages in this section could be carried out rather mechanically using:

1. coinduction,
2. equality chains,
3. already proven lemmata.

We did not require any up-to techniques or creative insight such as finding bisimulation relations to carry out our proofs. Thus, it is likely that after initiating coinduction, standard first-order theorem provers could fill in the remaining steps.

5. Constructing Automata

In this section, we show that deterministic automata form a Kleene algebra like decidable languages do. We show how to construct union, concatenation, and Kleene star of automata, in a recapitulation of the classic theory of formal languages. Our message is that the corresponding correctness proofs can be carried out by the same means as in the last section: coinduction and equational reasoning.

In our presentation of deterministic automata (DA) we follow Rutten (1998): A not necessarily finite automaton over a state set S is given by a transition function $\delta : S \to A \to S$ and a characteristic function $\nu : S \to \mathsf{Bool}$ for the set of accepting (or final) states. These two functions could also be bundled as $S \to \mathsf{Bool} \times (A \to S)$, making apparent that an automaton $(S : \mathsf{Set},\ da : \mathsf{DA}\, S)$ is just a $\mathsf{Bool} \times (A \to \_)$-coalgebra.

```
record DA (S : Set) : Set where
  field ν  :  (s : S) → Bool
        δ  :  (s : S) (a : A) → S

  νs : ∀{i} (ss : List i S) → Bool
  νs ss = List.any ν ss

  δs : ∀{i} (ss : List i S) (a : A) → List i S
  δs ss a = List.map (λ s → δ s a) ss
```

In anticipation of power automata we lift the coalgebra to lists of states $\mathsf{List}\ i\, S \to \mathsf{Bool} \times (A \to \mathsf{List}\ i\, S)$. A lists of states is accepting ($\nu\mathsf{s}$) if it contains at least one final state. And we step ($\delta\mathsf{s}$) to a new list of states by pointwise applying the transition function.

The initial state is not contained in the automaton definition; each state s induces a language $\mathsf{lang}\ da\, s$ accepted by an automaton da, which can be defined by simple coiteration:

```
lang : ∀{i} {S} (da : DA S) (s : S) → Lang i
Lang.ν (lang da s)     =  DA.ν da s
Lang.δ (lang da s) a   =  lang da (DA.δ da s a)
```

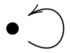
For each automaton (S, da) the function $\mathsf{lang}\ da : S \to \mathsf{Lang}\,\infty$ is the terminal morphism.

$$\begin{tikzcd}
S \arrow[r, "\langle \mathsf{DA}.\nu\ da,\ \mathsf{DA}.\delta\ da \rangle"] \arrow[d, dotted, "\mathsf{lang}\ da"'] & \mathsf{Bool} \times (A \to S) \arrow[d, "\mathsf{id} \times (\mathsf{lang}\ da \circ \_)"] \\
\mathsf{Lang}\,\infty \arrow[r, "\langle \mathsf{Lang}.\nu, \mathsf{Lang}.\delta \rangle"] & \mathsf{Bool} \times (A \to \mathsf{Lang}\,\infty)
\end{tikzcd}$$

5.1. Simple constructions on automata

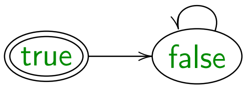
An automaton for the empty language can be constructed with a single non-accepting state inhabiting Agda's unit type $\top$.

```
∅A : DA ⊤
ν  ∅A  s     =  false
δ  ∅A  s  a  =  s
```



To recognize the language of the empty word, we take need two states, accepting true and non-accepting false : Bool.

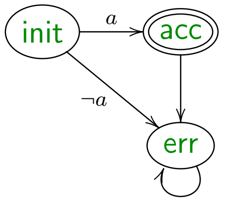
```
εA : DA Bool
ν  εA  b     =  b
δ  εA  b  a  =  false
```



For accepting a the single letter word a, we have three states: an initial state init, an accepting state acc, and a rejecting error state err.

```
data 3States : Set where
  init acc err : 3States

charA : (a : A) → DA 3States
ν  (charA a)  init     =  false
ν  (charA a)  acc      =  true
ν  (charA a)  err      =  false
δ  (charA a)  init  x  =
  if ⌊ a ≟ x ⌋ then acc else err
δ  (charA a)  acc   x  =  err
δ  (charA a)  err   x  =  err
```



Given an automaton da, we construct the automaton complA da for the complement language by switching accepting and non-accepting states.

```
complA : ∀{S} (da : DA S) → DA S
ν  (complA da)  s     =  not (ν da s)
δ  (complA da)  s  a  =  δ da s a
```

Given an automaton da_1 over state set S_1 accepting language ℓ_1 and an automaton da_2 over S_2 for ℓ_2, we can recognize the union $\ell_1 \cup \ell_2$ by the following *product* automaton $da_1 \oplus da_2$ over state set $S_1 \times S_2$. A state in the product automaton is a pair of states (s_1, s_2), one from each original automaton. Transitions are done in lock-step, and for acceptance at least one of the original automata must be in a final state.

```
_⊕_ : ∀{S₁ S₂} (da₁ : DA S₁) (da₂ : DA S₂) → DA (S₁ × S₂)
ν  (da₁ ⊕ da₂)  (s₁ , s₂)     =  ν da₁ s₁     ∨  ν da₂ s₂
δ  (da₁ ⊕ da₂)  (s₁ , s₂)  a  =  δ da₁ s₁ a  ,   δ da₂ s₂ a
```

5.2. Automaton composition for language concatenation

In preparation for automaton constructions for language concatenation and iteration, we define the power automaton, which allows us to be in a set of states at the same time. It is actually sufficient to consider finite sets of states, which we represent a bit redundantly as lists.

```
powA : ∀{S} (da : DA S) → DA (List ∞ S)
ν  (powA da)  ss     =  νs da ss
δ  (powA da)  ss  a  =  δs da ss a
```

If we start the power automaton in state $[s_1, \ldots, s_n]$, the accepted language will be the $\bigcup_{i=1}^{n} \mathsf{lang}\, da\, s_i$. We prove this in two steps: First, if we start out in no states, the accepted language is empty.

```
powA-nil : ∀{i S} (da : DA S) →

   lang (powA da) [] ≅⟨ i ⟩≅ ∅

≅ν  (powA-nil da)     = refl
≅δ  (powA-nil da)  a  = powA-nil da
```

If we start in the non-empty list s :: ss, we accept the union of the accepted language of da from s and the accepted language of powA da from ss.

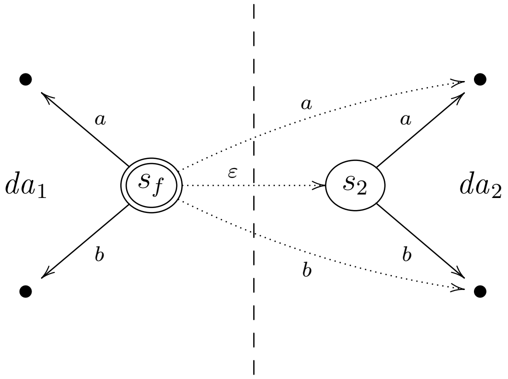
```
powA-cons : ∀{i S} (da : DA S) {s : S} {ss : List ∞ S} →

   lang (powA da) (s :: ss) ≅⟨ i ⟩≅ lang da s ∪ lang (powA da) ss

≅ν  (powA-cons da)     = refl
```

$$\cong\delta \;\; (\textsf{powA-cons}\ da)\;\; a \;\; = \textsf{powA-cons}\ da$$

For language concatenation, given two automata da_1 and da_2, we will construct a composition automaton composeA $da_1\, s_2\, da_2$ such that its accepted language from state s_1 is the language concatenation lang $da_1\, s_1$ · lang $da_2\, s_2$. The key insight is that whenever we reach a final state s_f in da_1, we non-deterministically jump to the initial state s_2 of da_2. In some formulations of non-deterministic automata this would be an ε-transition from s_f to s_2, consuming no input. We will instead add transitions from s_f to the successor states of s_2.



This means for the composition that we are in one state of da_1 and in zero or more states of da_2 at the same time. Thus, the type of states is $S_1 \times$ List $\infty\, S_2$ and we consider the power of the second automaton.

$$\textsf{composeA} : \forall\{S_1\ S_2\}$$
$$(da_1 : \textsf{DA}\ S_1)\ (s_2 : S_2)\ (da_2 : \textsf{DA}\ S_2) \to \textsf{DA}\ (S_1 \times \textsf{List}\ \infty\ S_2)$$

A state (s_1, ss_2) of the composition automation is final if any of ss_2 is final, or if s_1 is final and the initial state s_2 of da_2 is also a final state. (The latter means that the second language is nullable, so any word of the first language is contained in the composition.)

$$\nu\;\; (\textsf{composeA}\ da_1\ s_2\ da_2)\;\; (s_1\ ,\ ss_2)\;\; =$$
$$(\nu\ da_1\ s_1 \wedge \nu\ da_2\ s_2) \vee \nu\textsf{s}\ da_2\ ss_2$$

To step from state (s_1, ss_2) we consider two cases. First, if s_1 is not final, we simply transition pointwise, from s_1 with δ da_1, and from each state in ss_2 with δ da_2. However, if s_1 is final, we imagine us also in the initial state s_2 of da_2, thus, we add to this the transition we can make from s_2 in the second automaton.

```
δ (composeA da₁ s₂ da₂) (s₁ , ss₂) a =
  δ da₁ s₁ a , δs da₂ (if ν da₁ s₁ then s₂ ∷ ss₂ else ss₂) a
```

The composition automaton is a non-trivial construction, thus, it makes sense to look at its correctness proof. We have to generalize the correctness statement to arbitrary initial states (s_1, ss) in the composition automaton. If ss is not empty, the accepted language of the composition automaton contains the union of the accepted languages from each state in ss as well.

```
composeA-gen : ∀{i S₁ S₂} (da₁ : DA S₁) (da₂ : DA S₂) →
  ∀ (s₁ : S₁) (s₂ : S₂) (ss : List ∞ S₂) →

    lang (composeA da₁ s₂ da₂) (s₁ , ss)
  ≅⟨ i ⟩≅
    lang da₁ s₁ · lang da₂ s₂ ∪ lang (powA da₂) ss
```

The proof is by coinduction, using lemma powA-cons in case s_1 is final.

```
≅ν (composeA-gen da₁ da₂ s₁ s₂ ss) = refl
≅δ (composeA-gen da₁ da₂ s₁ s₂ ss) a with ν da₁ s₁
... | false = composeA-gen da₁ da₂ (δ da₁ s₁ a) s₂ (δs da₂ ss a)

... | true  = begin

    lang (composeA da₁ s₂ da₂)
      (δ da₁ s₁ a , δ da₂ s₂ a ∷ δs da₂ ss a)

  ≈⟨  composeA-gen da₁ da₂ (δ da₁ s₁ a) s₂ (δs da₂ (s₂ ∷ ss) a) ⟩

    lang da₁ (δ da₁ s₁ a) · lang da₂ s₂ ∪
      lang (powA da₂) (δs da₂ (s₂ ∷ ss) a)

  ≈⟨  union-congʳ (powA-cons da₂) ⟩
```

```
    lang da₁ (δ da₁ s₁ a) · lang da₂ s₂ ∪
      (lang da₂ (δ da₂ s₂ a) ∪ lang (powA da₂) (δs da₂ ss a))

  ≈⟨    ≅sym (union-assoc _) ⟩

    lang da₁ (δ da₁ s₁ a) · lang da₂ s₂ ∪
      lang da₂ (δ da₂ s₂ a) ∪ lang (powA da₂) (δs da₂ ss a)

  ■ where open EqR (Bis _)
```

As a corollary for empty ss, we obtain the correctness of automaton composition:

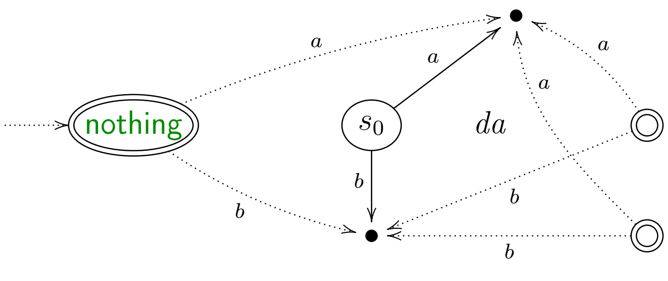
```
    composeA-correct : ∀{i S₁ S₂} (da₁ : DA S₁) (da₂ : DA S₂) s₁ s₂ →

      lang (composeA da₁ s₂ da₂) (s₁ , []) ≅⟨ i ⟩≅ lang da₁ s₁ · lang da₂ s₂
```

5.3. Automaton construction for language iteration

Finally, we construct from an automaton da accepting language ℓ from state $s_0 : S$ an automaton starA da for the iterated language ℓ^*. We do this in two steps:

1. acceptingInitial: Add a new final state nothing : Maybe S with the same successors as s_0. State nothing will serve as the new initial state. Its finality guarantees that the empty word is accepted.
2. finalToInitial: Add the successors of s_0 to each final state. This enables iteration. At this point, the automaton becomes "non-deterministic", i. e., we switch to List ∞ (Maybe S).



The first step embeds states $s : S$ of da as just s : Maybe S.

```
acceptingInitial : ∀{S} (s₀ : S) (da : DA S) → DA (Maybe S)
ν (acceptingInitial s₀ da) (just s)     = ν da s
δ (acceptingInitial s₀ da) (just s) a  = just (δ da s a)
```

It adds the new accepting state nothing : Maybe S with the successors of s_0.

```
ν (acceptingInitial s₀ da) nothing     = true
δ (acceptingInitial s₀ da) nothing a  = just (δ da s₀ a)
```

The second step constructs the power automaton and add transitions from the final states to the successors of the initial state.

```
finalToInitial : ∀{S} (da : DA (Maybe S)) → DA (List ∞ (Maybe S))
ν (finalToInitial da) ss     = νs da ss
δ (finalToInitial da) ss a =
  let ss′ = δs da ss a
  in  if νs da ss then δ da nothing a :: ss′ else ss′
```

Composing these steps leads to the automaton for language iteration.

```
starA : ∀{S} (s₀ : S) (da : DA S) → DA (List ∞ (Maybe S))
starA s₀ da = finalToInitial (acceptingInitial s₀ da)
```

To verify the construction, we first note some properties of the first step. For one, embedding the states of da via just : $S \to$ Maybe S does not change the accepted language.

```
acceptingInitial-just : ∀{i S} (s₀ : S) (da : DA S) {s : S} →

  lang (acceptingInitial s₀ da) (just s) ≅⟨ i ⟩≅ lang da s
```

This lemma is proven directly by coinduction. Further, is accepted language by the new state nothing : Maybe S is the language accepted by s_0 enriched with the empty word.

```
acceptingInitial-nothing : ∀{i S} (s₀ : S) (da : DA S) →
```

```
lang (acceptingInitial s₀ da) nothing ≅⟨ i ⟩≅ ε ∪ lang da s₀
```

The proof by coinduction uses acceptingInitial-just.

The main lemma characterizes the language accepted by starA s_0 da from an arbitrary state ss.

```
starA-lemma : ∀{i S} (da : DA S) (s₀ : S) (ss : List ∞ (Maybe S)) →

    lang (starA s₀ da) ss
  ≅⟨ i ⟩≅
    lang (powA (acceptingInitial s₀ da)) ss · (lang da s₀) *
```

The proof by coinduction uses powA-cons, acceptingInitial-just, and some laws of decidable languages as proven in Section 3.

Finally, we prove correctness of the starA-construction: If we start in the new initial state nothing (only), the recognized language is the Kleene star of the language recognized by da from s_0.

```
starA-correct : ∀{i S} (da : DA S) (s₀ : S) →

  lang (starA s₀ da) (nothing :: []) ≅⟨ i ⟩≅ (lang da s₀) *
```

The proof is direct, instantiating the starA-lemma, using correctness of the powA-construction, and lemma acceptingInitial-nothing.

6. Conclusions and Too Much Related Work

In this article, we have demonstrated that well-founded coinduction communicated by sized types and copattern matching allows for elegant definitions of decidable languages, language operations, and correctness proofs for automata constructions. All definitions and proofs could be carried out formally in the Agda proof assistant, using standard tools like equation chains and a simple monoid solver.

Beyond the material presented in this article, we have also formalized regular expressions and their equivalence to regular (Chomsky type 3) grammars, using the same proof techniques.

Being one of the oldest topics of computer science and taught to every student, there is an abundance of related work we are not able to review here. We just wish to mention a recent and comprehensive Coq formalization of classic automata theory by Doczkal et al. (2013). In contrast to us, they properly restrict to *finite* automata, using the support for finite types given by the SSReflect library (Gonthier and Mahboubi, 2010).

Acknowledgments. The author acknowledges support from Vetenskapsrådet (Swedish Research Council) through project 621-2014-4864/E0486401 *Termination Certification for Dependently-Typed Programs and Proofs via Refinement Types* and from the COST Action CA15123 *European research network on types for programming and verification* (EUTYPES). He is grateful to Ichiro Hasuo and the program committee of the IFIP WG 1.3 International Workshop on Coalgebraic Methods in Computer Science, CMCS 2016, for an invitation to present his findings at this workshop.

References

Abel, A., 2008. Semi-continuous sized types and termination. Logical Methods in Computer Science 4 (2:3), 1–33, CSL'06 special issue.
URL http://dx.doi.org/10.2168/LMCS-4(2:3)2008

Abel, A., Pientka, B., 2013. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: Morrisett, G., Uustalu, T. (Eds.), Proceedings of the Eighteenth ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA, September 25-27, 2013. ACM Press, pp. 185–196.
URL http://doi.acm.org/10.1145/2500365.2500591

Abel, A., Pientka, B., Thibodeau, D., Setzer, A., 2013. Copatterns: Programming infinite structures by observations. In: Giacobazzi and Cousot (2013), pp. 27–38.
URL http://dl.acm.org/citation.cfm?id=2429069

AgdaTeam, 2016. The Agda Wiki.
URL http://wiki.portal.chalmers.se/agda

Altenkirch, T., 2001. Representations of first order function types as terminal coalgebras. In: Abramsky, S. (Ed.), Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May

2-5, 2001, Proceedings. Vol. 2044 of Lecture Notes in Computer Science. Springer, pp. 8–21.
URL http://dx.doi.org/10.1007/3-540-45413-6_5

Amadio, R. M., Coupet-Grimal, S., 1998. Analysis of a guard condition in type theory (extended abstract). In: Nivat, M. (Ed.), Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Vol. 1378 of Lecture Notes in Computer Science. Springer, pp. 48–62.
URL http://dx.doi.org/10.1007/BFb0053541

Arden, D. N., 1961. Delayed-logic and finite-state machines. In: 2nd Annual Symposium on Switching Circuit Theory and Logical Design, Detroit, Michigan, USA, October 17-20, 1961. IEEE Computer Society Press, pp. 133–151.
URL http://dx.doi.org/10.1109/FOCS.1961.13

Barthe, G., Frade, M. J., Giménez, E., Pinto, L., Uustalu, T., 2004. Type-based termination of recursive definitions. Mathematical Structures in Computer Science 14 (1), 97–141.
URL http://dx.doi.org/10.1017/S0960129503004122

Bizjak, A., Grathwohl, H. B., Clouston, R., Møgelberg, R. E., Birkedal, L., 2016. Guarded dependent type theory with coinductive types. In: Jacobs, B., Löding, C. (Eds.), Foundations of Software Science and Computation Structures - 19th International Conference, FoSSaCS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Vol. 9634 of Lecture Notes in Computer Science. Springer, pp. 20–35.
URL http://dx.doi.org/10.1007/978-3-662-49630-5_2

Brzozowski, J. A., 1964. Derivatives of regular expressions. Journal of the Association of Computing Machinery 11 (4), 481–494.
URL http://doi.acm.org/10.1145/321239.321249

Coquand, T., 1994. Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (Eds.), Types for Proofs and Programs, International Workshop

TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers. Vol. 806 of Lecture Notes in Computer Science. Springer, pp. 62–78.
URL http://dx.doi.org/10.1007/3-540-58085-9_72

Doczkal, C., Kaiser, J., Smolka, G., 2013. A constructive theory of regular languages in Coq. In: Gonthier, G., Norrish, M. (Eds.), Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings. Vol. 8307 of Lecture Notes in Computer Science. Springer, pp. 82–97.
URL http://dx.doi.org/10.1007/978-3-319-03545-1_6

Dybjer, P., 1994. Inductive families. Formal Aspects of Computing 6 (4), 440–465.

Geuvers, H., 1992. Inductive and coinductive types with iteration and recursion. In: Nordström, B., Pettersson, K., Plotkin, G. (Eds.), Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992. pp. 193–217.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.9758

Giacobazzi, R., Cousot, R. (Eds.), 2013. The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013. ACM Press.
URL http://dl.acm.org/citation.cfm?id=2429069

Gianantonio, P. D., Miculan, M., 2003. A unifying approach to recursive and co-recursive definitions. In: Geuvers, H., Wiedijk, F. (Eds.), Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers. Vol. 2646 of Lecture Notes in Computer Science. Springer.

Gonthier, G., Mahboubi, A., 2010. An introduction to small scale reflection in coq. J. Formalized Reasoning 3 (2), 95–152.
URL http://dx.doi.org/10.6092/issn.1972-5787/1979

Hansen, H. H., Kupke, C., Rutten, J., 2016. Stream differential equations: Specification formats and solution methods. The Computing Research Repository (CoRR) abs/1609.08367.
URL http://arxiv.org/abs/1609.08367

Hinze, R., 2000. Generalizing generalized tries. Journal of Functional Programming 10 (4), 327–351.

Hughes, J., Pareto, L., Sabry, A., 1996. Proving the correctness of reactive systems using sized types. In: Boehm, H.-J., Steele Jr., G. L. (Eds.), Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996. ACM Press, pp. 410–423.
URL http://doi.acm.org/10.1145/237721.240882

Hur, C., Neis, G., Dreyer, D., Vafeiadis, V., 2013. The power of parameterization in coinductive proof. In: Giacobazzi and Cousot (2013), pp. 193–206.
URL http://doi.acm.org/10.1145/2429069.2429093

INRIA, 2016. The Coq Proof Assistant Reference Manual. INRIA, version 8.6 Edition.
URL http://coq.inria.fr/

Martin-Löf, P., 1975. An intuitionistic theory of types: Predicative part. In: Rose, H. E., Shepherdson, J. C. (Eds.), Logic Colloquium '73. North-Holland, pp. 73–118.

Pous, D., Sangiorgi, D., 2012. Enhancements of the bisimulation proof method. In: Sangiorgi, D., Rutten, J. (Eds.), Advanced Topics in Bisimulation and Coinduction. Cambridge University Press.

Rutten, J. J. M. M., 1998. Automata and coinduction (an exercise in coalgebra). In: Sangiorgi, D., de Simone, R. (Eds.), CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings. Vol. 1466 of Lecture Notes in Computer Science. Springer, pp. 194–218.
URL http://dx.doi.org/10.1007/BFb0055624

Sacchini, J. L., 2013. Type-based productivity of stream definitions in the calculus of constructions. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013. IEEE Computer Society Press, pp. 233–242.
URL http://dx.doi.org/10.1109/LICS.2013.29

Tarski, A., 1955. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309.

Traytel, D., 2016. Formal languages, formally and coinductively. In: Kesner, D., Pientka, B. (Eds.), 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal. Vol. 52 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 31:1–31:17.
URL http://dx.doi.org/10.4230/LIPIcs.FSCD.2016.31