Symbolic and Automatic Differentiation of Languages

ANONYMOUS AUTHOR(S)

Formal languages are usually defined in terms of set theory. Choosing type theory instead gives us languages as type-level predicates over strings. Applying a language to a string yields a type whose elements are language membership proofs describing *how* a string parses in the language. The usual building blocks of languages (including union, concatenation, and Kleene closure) have precise and compelling specifications uncomplicated by operational strategies and are easily generalized to a few general domain-transforming and codomain-transforming operations on predicates.

A simple characterization of languages (and indeed functions from lists to any type) captures the essential idea behind language "differentiation" as used for recognizing languages, leading to a collection of lemmas about type-level predicates. These lemmas are the heart of two dual parsing implementations—using (inductive) regular expressions and (coinductive) tries—each containing the same code but in dual arrangements (with representation and primitive operations trading places). The regular expression version corresponds to symbolic differentiation, while the trie version corresponds to automatic differentiation.

The relatively easy-to-prove properties of type-level languages transfer almost effortlessly to the decidable implementations. In particular, despite the inductive and coinductive nature of regular expressions and tries respectively, we need neither inductive nor coinductive/bisimulation arguments to prove algebraic properties.

ACM Reference Format:
Anonymous Author(s). 2021. Symbolic and Automatic Differentiation of Languages. 1, 1 (March 2021), 16 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SPECIFYING LANGUAGES

Languages are usually formalized either set-theoretically as sets of strings or operationally as parsers. Alternatively, one can use type theory, so that a language is a type-level predicate on "strings" (lists of an arbitrary type A of "characters"). The usual language operations are defined in Figure 1, including language union and intersection ($P \cup Q$ and $P \cap Q$) with their identities (the empty language $\varnothing$ and universal language $\mathscr{U}$), language concatenation ($P * Q$) and its identity (1, containing only the empty string), single-character languages ($` c$), Kleene star ($P^{\bigstar}$), and "scalar multiplication" ($s \cdot P$, which will prove useful later).

The definitions in Figure 1 reflect the logical interpretation of types under the Curry-Howard ("propositions as types") isomorphism, in which types represent propositions and elements (values) of a type represent proofs of the corresponding proposition [Wadler 2015]. The specific embodiment of Curry-Howard in Figure 1 and throughout this paper is the language Agda [Norell 2008; Bove et al. 2009], which is both programming language and proof assistant founded on Martin-Löf's intuitionistic type theory [Martin-Löf and Sambin 1984]. Specifically,

- Types have type Set ℓ for some universe level ℓ. These levels avoid logical inconsistencies but can be safely ignored in this paper (which is parametrized over ℓ).
- The types $\bot$ and $\top$ represent falsity and truth respectively, with $\bot$ having no elements (i.e., uninhabited) and $\top$ having a single element tt.

XXXX-XXXX/2021/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
Lang = A * → Set ℓ

∅ w = ⊥                    (P ∪ Q) w = P w ⊎ Q w
𝒰 w = ⊤                    (P ∩ Q) w = P w × Q w
1 w = w ≡ []               (P ∗ Q) w = ∃ λ (u , v) → (w ≡ u ++ v) × P u × Q v
‘ c w = w ≡ [ c ]          (P ☆) w = ∃ λ ws → (w ≡ concat ws) × All P ws
(s · P) w = s × P w
```

Fig. 1. Languages as type-level predicates

- The types $A \uplus B$ and $A \times B$ represent disjunction and conjunction respectively as non-dependent sum and products.
- For a function $F : A \to \mathsf{Set}\ \ell$, the types $\exists\ F$ and $\forall\ x \to F\ x$ represent existential and universal quantification as *dependent* product and function types.
- The type $x \equiv y$ represents *propositional* equality (rather than computable equality) as a data type with the single constructor $\mathsf{refl} : \forall\ x \to x \equiv x$.

In the definition of $P^{\bigstar}$, concat flattens a list of lists into a single list via a right fold. The predicate All P holds of a list ws when P holds for every element of ws:[1]

```
concat : (A *) * → A *
concat = foldr _++_ []

foldr : (B → X → X) → X → B * → X
foldr h x []       = x
foldr h x (b :: bs) = h b (foldr h x bs)

data All (P : B → Set ℓ) : B * → Set ℓ where
  []  : All P []
  _::_ : ∀ {b bs} → P b → All P bs → All P (b :: bs)
```

These three definitions are provided in the Agda standard library [Agda Team 2020].

Note that for a language P and string w, $P\ w$ is the type of *proofs* that $w \in P$, in other words *explanations* of membership, or *parsings*. (If the type $P\ w$ is uninhabited, then $w \notin P$.) For instance, every proof of $w \in P \cup Q$ contains a proof of $w \in P$ or of $w \in Q$ *and* the knowledge of which one was chosen (with different proofs reflecting different choices). Likewise, a proof of w $\in P * Q$ includes the choice of strings u and v, a proof that $w \equiv u +\!\!+ v$, and proofs that $u \in P$ and $v \in Q$.

The use of type-level predicates makes for simple, direct specification, including the use of existential quantification. As described in Section 6, it is also fairly easy to prove algebraic properties of predicates. These definitions do not give us decidable parsing, because general type inhabitation amounts to theorem proving. As shown in the rest of this paper, however, type-level predicates serve as a simple, precise, and non-operational basis for specification and reasoning, with correct and computable parsing implementations following systematically as corollaries.

This paper makes the following specific contributions:

- A collection of lemmas are stated and proved about functions over lists, capturing the semantic essence of Brzozowski's syntactic technique of derivatives of regular expressions (Section 2).

[1] The data type constructors [] and _::_ are overloaded here, referring to both lists (as indices) and All.

- A duality is revealed between regular expressions and tries (prefix trees) in the context of language differentiation (Section 4). Each implementation contains exactly the same code but in transposed forms. Both are corollaries of the semantic lemmas and are automatically proved correct by type-checking.
- The theory and algorithms are specified in terms of proof isomorphism rather than the usual, coarser relation of logical equivalence. As such, the lemmas for language concatenation and Kleene closure are more informative than in previous work, capturing *all* parsings distinctly, including closure of languages containing the empty string.
- The vocabulary of languages is generalized from predicates on lists to predicates on arbitrary types, and then re-specialized to the usual vocabulary (Section 6.1). This generalization consists of two useful covariant applicative functors on functions (the usual one on functions-from-X and another on functions-to-Set).
- Properties are stated first at the level of predicates, making them easy to understand and prove, thanks to elimination of operational details. These properties are then transported to the computable parsing implementations (Section 6.2) with no additional proofs needed.

Although many proofs are elided below, the source code for this paper is freely available and contains fully verified Agda code with no postulates. All type signatures and definitions in the paper are generated by the Agda compiler.

2 DECOMPOSING LANGUAGES

In order to convert languages into parsers, it will help to understand how language membership relates to the algebraic structure of lists, specifically the monoid formed by _++_ and []. Generalizing from languages to functions from lists to *any* type, define

```
ν : (A ⋆ → B) → B                    – "nullable"
ν f = f []

𝒟 : (A ⋆ → B) → A ⋆ → (A ⋆ → B)      – "derivative"
𝒟 f u = λ v → f (u ++ v)
```

For languages, $\mathcal{D}\ P\ u$ is the set of u-suffixes from P, i.e., the strings v such that $u \mathbin{+\!\!+} v \in P$.

Since $u \mathbin{+\!\!+} [] \equiv u$, it follows that[2,3]

```
ν∘𝒟 : ν ∘ 𝒟 f ≗ f
```

Moreover, the function $\mathcal{D}$ distributes over [] and _++_:[4]

```
𝒟[] : 𝒟 f [] ≡ f

𝒟++ : 𝒟 f (u ++ v) ≡ 𝒟 (𝒟 f u) v
```

These facts suggest that $\mathcal{D}$ can be computed one list element at a time via a more specialized operation, as is indeed the case:[5]

[2] Agda variable names can contain most characters other than spaces and parentheses, e.g., "ν∘𝒟" as declared here.
[3] For functions f and g, $f \stackrel{\circ}{=} g$ is extensional equality, i.e., $\forall\ x \to f\,x \equiv g\,x$.
[4] In algebraic terms, argument-flipped $\mathcal{D}$ is a monoid homomorphism from lists to endofunctions.
[5] Repeated δ application is expressed here via a standard left fold (with $X = A ^\star \to B$ for 𝒟foldl):

```
foldl : (X → A → X) → X → A ⋆ → X
foldl h x []       = x
foldl h x (a :: as) = foldl h (h x a) as
```

$$\nu\varnothing : \nu\ \varnothing \equiv \bot \qquad \delta\varnothing : \delta\ \varnothing\ a \equiv \varnothing$$

$$\nu\mathcal{U} : \nu\ \mathcal{U} \equiv \top \qquad \delta\mathcal{U} : \delta\ \mathcal{U}\ a \equiv \mathcal{U}$$

$$\nu\cup : \nu\ (P \cup Q) \equiv (\nu\ P \uplus \nu\ Q) \qquad \delta\cup : \delta\ (P \cup Q)\ a \equiv \delta\ P\ a \cup \delta\ Q\ a$$

$$\nu\cap : \nu\ (P \cap Q) \equiv (\nu\ P \times \nu\ Q) \qquad \delta\cap : \delta\ (P \cap Q)\ a \equiv \delta\ P\ a \cap \delta\ Q\ a$$

$$\nu\cdot : \nu\ (s \cdot P) \equiv (s \times \nu\ P) \qquad \delta\cdot : \delta\ (s \cdot P)\ a \equiv s \cdot \delta\ P\ a$$

$$\nu 1 : \nu\ 1 \leftrightarrow \top \qquad \delta 1 : \delta\ 1\ a \longleftrightarrow \varnothing$$

$$\nu* : \nu\ (P * Q) \leftrightarrow (\nu\ P \times \nu\ Q) \qquad \delta* : \delta\ (P * Q)\ a \longleftrightarrow \nu\ P \cdot \delta\ Q\ a \cup \delta\ P\ a * Q$$

$$\nu^{\bigstar} : \nu\ (P^{\bigstar}) \leftrightarrow (\nu\ P)^{\star} \qquad \delta^{\bigstar} : \delta\ (P^{\bigstar})\ a \longleftrightarrow (\nu\ P)^{\star} \cdot (\delta\ P\ a * P^{\bigstar})$$

$$\nu\text{`} : \nu\ (\text{`}\ c) \leftrightarrow \bot \qquad \delta\text{`} : \delta\ (\text{`}\ c)\ a \longleftrightarrow (a \equiv c) \cdot 1$$

Fig. 2. Properties of ν and δ for language operations

$$\delta : (A^{\star} \to B) \to A \to (A^{\star} \to B)$$
$$\delta\ f\ a = \mathscr{D}\ f\ [\ a\ ]$$

$$\mathscr{D}\text{foldl} : \mathscr{D}\ f \triangleq \text{foldl}\ \delta\ f$$

Each use of δ thus takes us one step closer to reducing general language membership to nullability. This simple observation is the semantic heart of the syntactic technique of *derivatives of regular expressions* as used for efficient recognition of regular languages [Brzozowski 1964], later extended to parsing general context-free languages [Might and Darais 2010]. Lemmas $\nu\circ\mathscr{D}$ and $\mathscr{D}$foldl liberate this technique from the assumption that languages are represented *symbolically*, by some form of grammar (e.g., regular or context-free), as will be exploited in Section 4.3.

In terms of automata theory, derived languages are states, with P being the initial state and $\mathscr{D}\ P\ u$ the state reached from P after consuming the string u; δ is the state transition function; and ν is the set of accepting states. Lemmas $\nu\circ\mathscr{D}$ and $\mathscr{D}$foldl capture the correctness of this state machine as a recognizer for the language P.

Given the definitions of ν and δ above and of the language operations in Section 1, one can prove properties about how they relate, as shown in Figure 2. The correct-by-construction parsing algorithms in Section 4 are corollaries of these properties.

There are three relations involved in Figure 2: propositional equality ("$\equiv$"), (type) isomorphism ("$\leftrightarrow$") and extensional isomorphism ("$\longleftrightarrow$"). Isomorphism relates types (propositions) whose inhabitants (proofs) are in one-to-one correspondence. *Extensional* (or "pointwise") isomorphism relates predicates isomorphic on every argument:

$$P \longleftrightarrow Q = \forall\ \{w\} \to P\ w \leftrightarrow Q\ w$$

The equalities in Figure 2 are all proved automatically by normalization (i.e., their proofs are simply refl), while the other relations require a bit more work. As with all lemmas in this paper

```
⊥? : Dec ⊥
⊥? = no (λ ())

⊤? : Dec ⊤
⊤? = yes tt

_⊎?_ : Dec A → Dec B → Dec (A ⊎ B)
no ¬a ⊎? no ¬b = no [ ¬a , ¬b ]
yes a ⊎? no ¬b = yes (inj₁ a)
_     ⊎? yes b  = yes (inj₂ b)

_×?_ : Dec A → Dec B → Dec (A × B)
yes a ×? yes b  = yes (a , b)
no ¬a ×? yes b  = no (¬a ∘ proj₁)
_     ×? no ¬b = no (¬b ∘ proj₂)

_✶? : Dec A → Dec (A ✶)
_ ✶? = yes []
```

Fig. 3. Compositional decidability

shown with signatures only, full proofs are in the paper's source code and are formally verified by the Agda compiler (including during typesetting of this paper).

3 DECIDABILITY

For effective implementations, we must bridge the gap between a type (of membership proofs) and its decidable inhabitation. Fortunately, there is a convenient and compositional way to do so. For type A, the type Dec A contains proof of A or a proof of $\neg\ A$ (defined as usual to mean $A \to \bot$):[6]

```
data Dec (A : Set ℓ) : Set ℓ where
  yes :    A → Dec A
  no  : ¬ A → Dec A
```

For compositionality, we will use a few operations that lift decidability of types to decidability of constructions on those types, as shown in Figure 3.[7] Moreover, decidability lifts naturally from types to predicates:

```
Decidable P = ∀ x → Dec (P x)
```

With these definitions, we can formulate the problem of decidable language recognition: given a language P, construct a term of type Decidable P. To accomplish this transformation, we will look for decidable building blocks that mirror the predicate vocabulary defined in Section 1.

Type isomorphisms (such as those in Figure 2) play an important role in decidability, namely that isomorphic types or predicates are equivalently decidable:

```
_◃_ : B ↔ A → Dec A → Dec B

_◂_ : Q ⟷ P → Decidable P → Decidable Q
```

[6] The Agda standard library contains a more efficient version of Dec [Agda Team 2020, Relation.Nullary].

[7] A few explanatory remarks:

- λ () is the unique function from $\bot$ to any type (here also $\bot$), sometimes called "absurd" or "$\bot$-elim".
- inj_1 and inj_2 are left and right injections into a sum type.
- proj_1 and proj_2 are left and right projections out of a product type.
- For functions $f : A \to C$ and $g : B \to C$, the function $[\ f\ ,\ g\ ] : A \uplus B \to C$ maps $\text{inj}_1\ a$ to $f\ a$ and $\text{inj}_2\ b$ to $g\ b$.
- $\_^{\star ?}$ reflects the fact that the empty list exists for every element type (even $\bot$).

One direction of the isomorphism proves B from A, while the other proves $\neg$ B from $\neg$ A. In fact, logical *equivalence* suffices for the results in this paper, but the stronger condition of isomorphism enables variations such as generating many parses/proofs rather than just one.

4 FROM LANGUAGES TO PARSING

4.1 Reflections

The lemmas in Figure 2 tell us how to decompose languages defined in terms of the vocabulary from Figure 1, while the definitions in Figure 3 tell us how compute inhabitation of the resulting types, resulting in decidable parsing. These lemmas and definitions cannot be applied *automatically* in their present form, however, because languages are functions, as are ν and δ, and so are not subject to pattern-matching. An automatic solution would need some form of *reflection* of language-constructing operations or of ν and δ as inspectable data.

This situation is exactly as in differential calculus, since differentiation in that setting is also defined on functions rather than on symbolic representations. Fortunately, there are two standard solutions for *computable* differentiation, commonly referred to as "symbolic" and "automatic" differentiation. In the former, functions are represented symbolically in some suitable vocabulary, and pattern-matching is used to apply rules of differentiation to those symbolic representations. The latter solution involves re-interpreting the function-building vocabulary to construct not just the usual functions, but also their derivatives [Griewank 1989; Griewank and Walther 2008; Elliott 2018]. This latter option is often much more efficient than the former, since it easily avoids a good deal of work duplicated between a function and its derivative.

Sections 4.2 and 4.3 applies the symbolic and automatic strategies respectively to languages. Both algorithms are corollaries of the lemmas in Figure 2 and are automatically proved correct by type checking.

4.2 Symbolic Differentiation

The "symbolic differentiation" style reflects language-building vocabulary (Figure 1) into an inductive data type (of modestly extended regular expressions), while ν and δ become functions defined by pattern-matching on that data type. For convenience, we can use the same names as in Section 1 for these new counterparts, while referring to the original versions via the module prefix "$_\diamond$". The result is shown in Figure 4 along with a function $[\![\_]\!]$ that converts a symbolic language representation to a computable parser.

Decidable parsing relies only on ν and δ (via $[\![\_]\!]$), which are defined in Figure 5, as systematically derived from the lemmas of Figure 2. These clauses are meant to be read in "column-major" order, i.e., each column is a complete definition. Correctness is guaranteed by the types of ν and δ and so is proved automatically by type-checking Figure 5. (We could thus use *any* definitions of ν and δ that type- and termination-check, although not many definitions would do so.)

In Figure 5, note the role of the isomorphisms from Figure 2. Since those isomorphisms relate ν and δ on the language-building operations to types and languages expressed in that *same* vocabulary, we end up with a recursive algorithm.

4.3 Automatic Differentiation

Section 4.2 embodies one choice of reflecting functions into a data representation. Now consider the dual strategy of "automatic differentiation". This time, reflect ν and δ into a *coinductive* data

```
data Lang : ∗Lang → Set (suc ℓ) where
  ∅   : Lang ∗∅
  𝒰   : Lang ∗𝒰
  _∪_ : Lang P → Lang Q → Lang (P ∗∪ Q)
  _∩_ : Lang P → Lang Q → Lang (P ∗∩ Q)
  _·_ : Dec  s → Lang P → Lang (s ∗·  P)
  1   : Lang (∗1)
  _*_ : Lang P → Lang Q → Lang (P ∗*  Q)
  _☆  : Lang P → Lang (P ∗☆)
  `   : (a : A) → Lang (∗` a)
  _◂_ : (Q ⟷ P) → Lang P → Lang Q

ν : Lang P → Dec (∗ν P)
δ : Lang P → (a : A) → Lang (∗δ P a)

⟦_⟧ : Lang P → Decidable P
⟦ p ⟧ []       = ν p
⟦ p ⟧ (a :: w) = ⟦ δ p a ⟧ w
```

Fig. 4. Regular expressions (inductive)

```
ν ∅ = ⊥?                    δ ∅ a = ∅
ν 𝒰 = ⊤?                    δ 𝒰 a = 𝒰
ν (p ∪ q) = ν p ⊎? ν q      δ (p ∪ q) a = δ p a ∪ δ q a
ν (p ∩ q) = ν p ×? ν q      δ (p ∩ q) a = δ p a ∩ δ q a
ν (s · p) = s ×? ν p        δ (s · p) a = s · δ p a
ν 1 = ν1 ◃ ⊤?               δ 1 a = δ1 ◂ ∅
ν (p ∗ q) = ν∗ ◃ (ν p ×? ν q)   δ (p ∗ q) a = δ∗ ◂ (ν p · δ q a ∪ δ p a ∗ q)
ν (p ☆) = ν☆ ◃ (ν p ✶?)     δ (p ☆) a = δ☆ ◂ (ν p ✶? · (δ p a ∗ p ☆))
ν (` a) = ν` ◃ ⊥?           δ (` c) a = δ` ◂ ((a ≟? c) · 1)
ν (f ◂ p) = f ◃ ν p          δ (f ◂ p) a = f ◂ δ p a
```

Fig. 5. Symbolic differentiation (column-major/patterns)

type of tries[8], while redefining the language-building vocabulary as functions on that data type defined by *copatterns* [Abel and Pientka 2016]. (This program duality pattern has been noted and

[8]The classic trie ("prefix tree") data structure was introduced by Thue [1912] to represent sets of strings [Knuth 1998, Section 6.3], later generalized to arbitrary non-nested algebraic data types [Connelly and Morris 1995], and then from sets to functions [Hinze 2000].

```
record Lang (P : ₀Lang) : Set (suc ℓ) where
  coinductive
  field
    ν : Dec (₀ν P)
    δ : (a : A) → Lang (₀δ P a)

⟦_⟧ : Lang P → Decidable P
⟦ p ⟧ []       = ν p
⟦ p ⟧ (a :: w) = ⟦ δ p a ⟧ w

∅   : Lang ₀∅
𝒰   : Lang ₀𝒰
_∪_ : Lang P → Lang Q → Lang (P ₀∪ Q)
_∩_ : Lang P → Lang Q → Lang (P ₀∩ Q)
_·_ : Dec  s → Lang P → Lang (s ₀· P)
1   : Lang (₀1)
_*_ : Lang P → Lang Q → Lang (P ₀* Q)
_☆  : Lang P → Lang (P ₀☆)
‘   : (a : A) → Lang (₀‘ a)
_◂_ : (Q ⟷ P) → Lang P → Lang Q
```

Fig. 6. Tries (coinductive)

$$\nu\ \varnothing = \bot^?$$
$$\delta\ \varnothing\ a = \varnothing$$
$$\nu\ \mathcal{U} = \top^?$$
$$\delta\ \mathcal{U}\ a = \mathcal{U}$$
$$\nu\ (p \cup q) = \nu\ p \uplus^?\ \nu\ q$$
$$\delta\ (p \cup q)\ a = \delta\ p\ a \cup \delta\ q\ a$$
$$\nu\ (p \cap q) = \nu\ p \times^?\ \nu\ q$$
$$\delta\ (p \cap q)\ a = \delta\ p\ a \cap \delta\ q\ a$$
$$\nu\ (s \cdot p) = s \times^?\ \nu\ p$$
$$\delta\ (s \cdot p)\ a = s \cdot \delta\ p\ a$$
$$\nu\ 1 = \nu 1 \triangleleft \top^?$$
$$\delta\ 1\ a = \delta 1 \blacktriangleleft \varnothing$$
$$\nu\ (p * q) = \nu * \triangleleft (\nu\ p \times^?\ \nu\ q)$$
$$\delta\ (p * q)\ a = \delta * \blacktriangleleft (\nu\ p \cdot \delta\ q\ a \cup \delta\ p\ a * q)$$
$$\nu\ (p^{\bigstar}) = \nu^{\bigstar} \triangleleft (\nu\ p\ ^{\star?})$$
$$\delta\ (p^{\bigstar})\ a = \delta^{\bigstar} \blacktriangleleft (\nu\ p\ ^{\star?} \cdot (\delta\ p\ a * p^{\bigstar}))$$
$$\nu\ (‘\ a) = \nu‘ \triangleleft \bot^?$$
$$\delta\ (‘\ c)\ a = \delta‘ \blacktriangleleft ((a \stackrel{?}{=} c) \cdot 1)$$
$$\nu\ (f \blacktriangleleft p) = f \triangleleft \nu\ p$$
$$\delta\ (f \blacktriangleleft p)\ a = f \blacktriangleleft \delta\ p\ a$$

Fig. 7. Automatic differentiation (row-major/copatterns)

investigated by Ostermann and Jabs [2018].) Again, we will use the same names as in Section 1, referring to the original versions via the module prefix "₀". The result is shown in Figure 6.

Decidable recognition again relies only on ν and δ, which are defined for each language operation in Figure 7 (though with a problem to be noted and fixed in the next paragraph). These clauses

```
record Lang i (P : ∗Lang) : Set (suc ℓ) where
  coinductive
  field
    ν : Dec (∗ν P)
    δ : ∀ {j : Size< i} → (a : A) → Lang j (∗δ P a)

⟦_⟧ : Lang ∞ P → Decidable P
⟦ p ⟧ []       = ν p
⟦ p ⟧ (a :: w) = ⟦ δ p a ⟧ w

∅   : Lang i ∗∅
𝒰   : Lang i ∗𝒰
_∪_ : Lang i P → Lang i Q → Lang i (P ∗∪ Q)
_∩_ : Lang i P → Lang i Q → Lang i (P ∗∩ Q)
_·_ : Dec   s → Lang i P → Lang i (s ∗· P)
1   : Lang i (∗1)
_*_ : Lang i P → Lang i Q → Lang i (P ∗* Q)
_☆  : Lang i P → Lang i (P ∗☆)
`   : (a : A) → Lang i (∗` a)
_◂_ : (Q ⟷ P) → Lang i P → Lang i Q
```

Fig. 8. Sized tries (coinductive)

are meant to be read in "row-major" order, i.e., each row is a complete definition. Note that the clauses in Figure 7 are syntactically identical to those in Figure 5 but are organized dually. (The compiler-generated syntax coloring differs to reflect the changed interpretation of the definitions.)

The correctness proof is *almost* accomplished by type- and termination-checking, but a technical problem arises. The $\delta\ (p * q)$ clause does not satisfy Agda's termination checker, which cannot see that the argument $\delta\ p\ a$ in the recursive use of $\_*\_$ is in some sense smaller than p. (Figure 7 compiles only due to suppressing termination checking for the problematic clause with a compiler pragma.) Fortunately, this issue was already identified and solved by Abel [2016]—also in the setting of trie-based language recognition—by using *sized types* [Abel 2008; Abel and Pientka 2016]. This solution only requires giving Lang (now tries) an index $i : \mathsf{Size}$ corresponding to the maximum depth to which a trie can be searched, or equivalently the longest string that can be matched. In practice, we will work with arbitrarily deep tries, i.e., ones having index ∞, as in the type of $[\![\_]\!]$. The modified representation is shown in Figure 8. Decidable recognition is defined exactly as with unsized tries (Figure 7), with the sole exception of removing the compiler pragma that suppressed termination checking. This time the compiler successfully proves *total* correctness (including termination).

To more clearly see the parallel with automatic differentiation (AD), note that AD implementations sample a function *and* its derivative together to exploit the fact that these two computations typically share much common work. This work sharing also applies when differentiating languages. Without this optimization we have

```
𝒟′ : (A ⋆ → B) → A ⋆ → B × (A ⋆ → B)
𝒟′ f u = f u , 𝒟 f u
```

The potential work-sharing version:

```
Pred A = A → Set ℓ

pure^o : Set ℓ → Pred A                 pure^i : A → Pred A
pure^o x a = x                          pure^i x a = a ≡ x

map^o : (Set ℓ → Set ℓ) → (Pred A → Pred A)
map^o g P a = g (P a)

map^i : (A → B) → (Pred A → Pred B)
map^i g P b = ∃ λ a → b ≡ g a × P a

map^o₂ : (Set ℓ → Set ℓ → Set ℓ) →
         (Pred A → Pred A → Pred A)
map^o₂ h P Q a = h (P a) (Q a)

map^i₂ : (A → B → C) →
         (Pred A → Pred B → Pred C)
map^i₂ h P Q c = ∃ λ (a , b) → c ≡ h a b × P a × Q b
```

Fig. 9. Predicate operations

```
𝒟̂ : (A ⋆ → B) → A ⋆ → B × (A ⋆ → B)
𝒟̂ f u = ν f′ , f′ where f′ = foldl δ f u
```

Equivalence follows from lemmas ν∘𝒟 and 𝒟foldl of Section 2:

```
𝒟′≡𝒟̂ : 𝒟′ f ≐ 𝒟̂ f
```

The trie representation exploits this sharing potential by weaving ν and δ into a single structure based on common prefixes.

5 GENERALIZING FROM LANGUAGES TO PREDICATES

The language-building vocabulary defined in Section 1 can be simplified and generalized. First note that there are two categories of operations. One category transforms the codomain (types): $\varnothing$, $\mathcal{U}$, _∪_, and _∩_. The other transforms the domain (lists): 1, _∗_, $\_^{\star}$, and ‘. The first category applies to predicates over all types (not just over lists). Within the second category, 1, _∗_, and $\_^{\star}$ can apply to any monoid, while ‘ is specific to lists.

Figure 9 shows two parallel collections of predicate operations, each covariantly transforming the codomain (left) or domain (right).[9] These generalized operations then specialize to language operations as in Figure 10, which adds language complement ($\complement$).[10]

The ν and δ lemmas in Figure 2 for codomain (but not domain) operations are easily generalized as shown in Figure 11, all proved automatically by normalization. Since the decidable language implementations in Section 4 are corollaries of ν and δ lemmas, those implementations adapt easily to the generalized codomain operations (pure^o, map^o, and $\mathsf{map}^o{}_2$), resulting in a somewhat smaller implementation and broader coverage.

6 PROPERTIES

6.1 Predicate Algebra

The basic building blocks of type-level predicates—and languages in particular—form the vocabulary of a *closed semiring* in two different ways. The semiring abstraction has three aspects: (a) a commutative monoid providing “zero” and “addition”, (b) a (possibly non-commutative) monoid

[9] The types of operations on the left can be further relaxed from Set ℓ to any codomain type, while keeping A fixed, resulting in the usual covariant applicative functor of functions *from* a fixed type [McBride and Paterson 2008]. Dually, the operations on the right (in their current generality) form a second covariant applicative functor of functions *to* types.

[10] The list-specific operations _+_, [], and concat used in Figure 1 have been generalized to monoid operations _•_ and ε. The *definitions* of 1, _∗_, and $\_^{\star}$ do not require the monoid properties, but their properties will (Section 6.1).

```
Lang = Pred (A ⋆)

∅     = pure^o ⊥          1 = pure^i ε
𝒰     = pure^o ⊤          _*_ = map^i 2 _•_
_∪_   = map^o 2 _⊎_       P ☆ = map^i (foldr _•_ ε) (All P)
_∩_   = map^o 2 _×_       ‘ c = pure^i [ c ]
_·_ s = map^o (s ×_)
∁     = map^o ¬_
```

Fig. 10. Languages via predicate operations

$$
\begin{aligned}
&\nu\mathsf{pure}^o &&: \nu\ (\mathsf{pure}^o\ x) && \equiv x\\
&\nu\mathsf{map}^o &&: \nu\ (\mathsf{map}^o\ g\ P) && \equiv g\ (\nu\ P)\\
&\nu\mathsf{map}^o{}_2 &&: \nu\ (\mathsf{map}^o{}_2\ h\ P\ Q) && \equiv h\ (\nu\ P)\ (\nu\ Q)
\end{aligned}
$$

$$
\begin{aligned}
&\delta\mathsf{pure}^o &&: \delta\ (\mathsf{pure}^o\ x) && a \equiv \mathsf{pure}^o\ x\\
&\delta\mathsf{map}^o &&: \delta\ (\mathsf{map}^o\ g\ P) && a \equiv \mathsf{map}^o\ g\ (\delta\ P\ a)\\
&\delta\mathsf{map}^o{}_2 &&: \delta\ (\mathsf{map}^o{}_2\ h\ P\ Q) && a \equiv \mathsf{map}^o{}_2\ h\ (\delta\ P\ a)\ (\delta\ Q\ a)
\end{aligned}
$$

Fig. 11. Properties of ν and δ for predicate codomain operations

providing "one" and "multiplication", and (c) the relationship between them, namely that multiplication distributes over addition and zero.[11] In the first predicate semiring, which is *commutative* (i.e., multiplication commutes), zero and addition are $\varnothing$ and $\_\cup\_$, while one and multiplication are $\mathscr{U}$ and $\_\cap\_$. Closure adds a star/closure operation $\_^{\star}$ with star law: $x^{\star} \approx 1 + x * x^{\star}$.

Conveniently, booleans and types form commutative semirings with all necessary proofs already in the standard library.[12] They are both closed as well. For booleans, closure maps both false and true to true, with the star law holding definitionally. For types, the closure of A is $A^{\star}$ (the usual inductive list type) with a simple, non-inductive proof of star.

This first closed semiring for predicates follows from a much more general pattern. Given any two types A and B, if B is a monoid then $A \to B$ is as well. The monoid operation $\_\bullet\_$ lifts to the function-level binary operation $\lambda\ (f\ g : A \to B) \to \lambda\ a \to f\ a \bullet g\ a$. The monoid identity $\varepsilon : B$ lifts

[11] Distribution of multiplication over zero is also known as "annihilation".

[12] The equivalence relation used for types is isomorphism rather than equality. The boolean semiring is idempotent, while the type semiring is not (in order to distinguish multiple parsings in ambiguous languages). The latter non-idempotence accounts for the difference between the ν and δ lemmas for $P * Q$ and $P^{\star}$ in Figure 2 and the corresponding rules in previous work, as mentioned in Section 7.

to the identity $\lambda\ a \longrightarrow \varepsilon$. All of the laws transfer from B to $A \longrightarrow B$. Likewise for other algebraic structures. (As always, compiler-verified proofs are in this paper's source code.)

Looking more closely, additional algebraic structure emerges on predicates: (full) *semimodules* generalize vector spaces by relaxing the associated types of "scalars" from fields to commutative semirings, i.e., dropping the requirements of multiplicative and additive inverses. Left and right semimodules further generalize scalars to arbitrary semirings, dropping the commutativity requirement for scalar multiplication. Again, nothing is needed from the codomain type B beyond the properties of a semiring, so again predicates get these algebraic structures for free (already paid for by types). Every $P : \mathsf{Pred}\ A$ is a vector in the semimodule $\mathsf{Pred}\ A$, including the special case of "languages", for which A is a list type. The dimension of the semimodule is the cardinality of the domain type A (typically infinite), with "basis vectors" having the form $\mathsf{pure}^i\ a$ for $a : A$. A general vector (predicate) is a linear combination (often infinite) of these basis vectors, with addition being union and scaling defined by pairing membership proofs (as in Figure 10).

There is still more algebraic structure to be found and exploited. When our *domain* type A is a monoid (as with languages, infinite streams and grids, and functions of continuous space and time), predicates over A form a second semiring, known as "the monoid semiring" [Golan 2005], in which zero and addition are as in the first predicate commutative semiring and semimodule, while one and multiplication are 1 and $\_*\_$, defined in Figures 1 and 10. In this setting, language concatenation is subsumed by a very general form of *convolution* [Golan 2005; Dongol et al. 2016; Elliott 2019]. The semimodule structure of predicates provides the additive aspect and is built entirely from the *codomain*-transforming predicate operations defined in Figure 1 and redefined in Figure 10. The multiplicative aspect (convolution) comes entirely from the *domain*-transforming operations applied to any domain monoid. The two needed distributive properties hold for *any* domain-transforming operation, whether associative or not. The proofs of these properties, while reasonably straightforward, are beyond the scope of this paper but are available in the paper's Agda source code.

Not only do types form a closed semiring, the *only* properties needed from types in the monoid semiring come from the laws of closed semirings. One can thus apply the ideas in this paper to many other useful semirings, including natural numbers (e.g., number of parses), probability distributions, and the tropical semirings (max/plus and min/plus) for optimization. When the domain monoid commutes (multiplicatively)—e.g., for functions of space and time rather than languages—the monoid semiring does as well. Applications include power series (univariate and multivariate) and image processing [Elliott 2019]. Research on semirings in parsing began with Chomsky and Schützenberger [1959] and was further explored by Goodman [1998, 1999] and by Liu [2004].

6.2 Transporting Properties

We have seen that type-level languages have useful and illuminating algebraic properties and that the decidable implementations in Section 4 are tightly connected to languages. Must we prove these properties again for each implementation—where there are more operational details to manage—or do the algebraic properties somehow transfer to those implementations? This question immediately raises a technical obstacle. Algebraic abstractions and their properties are simply typed. For instance, each instance of the Monoid abstraction involves a single type τ with a binary operation $\_\bullet\_ : \tau \longrightarrow \tau \longrightarrow \tau$ and identity value $\varepsilon : \tau$. This recipe accommodates the language operations in Figures 1 and 10 and their generalizations in Figure 9 but not the dependently typed, *indexed* versions of languages and their operations in Figures 4 and 8, nor decidable types and their vocabulary in Figure 3.

```
inj₀ : ∀ {s : A} (s′ : F s) → ∃ F
inj₀ {s} s′ = s , s′

inj₁ : ∀ {g : A → A} (g′ : ∀ {a} → F a → F (g a)) → (∃ F → ∃ F)
inj₁ {g} g′ (a , x) = g a , g′ x

inj₂ : ∀ {h : A → A → A} (h′ : ∀ {a b} → F a → F b → F (h a b)) → (∃ F → ∃ F → ∃ F)
inj₂ {h} h′ (a , x) (b , y) = h a b , h′ x y
```

Fig. 12. Existential wrappers

Fortunately, there is a simple way to encapsulate indexed types into non-indexed types, namely existential quantification. For instance, ∃ Dec and ∃ (Lang ∞) are non-indexed types. For a type family $F : A \longrightarrow \mathsf{Set}\ \ell$, values of type ∃ F are pairs $(a\ ,\ b)$ with $a : A$ and $b : F\ a$. We can also encapsulate dependently typed *operations* on indexed types into simply typed operations on the (simply typed) existentially wrapped versions of those types. For instance, consider language union from Figure 4:

```
_∪_ : Lang P → Lang Q → Lang (P ◈∪ Q)
```

Referring to the type and operations of Figures 1 and 4 via the module prefixes "◈" and "▪" respectively, define

```
Lang = ∃ ▪Lang
```

Then wrap union as follows:

```
_∪_ : Lang → Lang → Lang
(P , p) ∪ (Q , q) = P ◈∪ Q , p ▪∪ q
```

The type-level language components of the arguments and result are given explicitly here but could instead be inferred by the compiler. Indexed operations of all arities can be systematically wrapped in this style, as shown in Figure 12, again with automatically inferable components given explicitly for clarity. Wrapped union can then be defined simply as `_∪_ = inj₂ ▪_∪_` and likewise for the other operations.

In addition to transporting types and operations as just described, we also need to transport *properties*. A crucial question is the choice of equivalence relation for the new properties. If we choose equivalence on these dependent pairs to be equivalence of *first* projections (type *indices*), then all algebraic properties of those indices will hold for the existentially wrapped versions as well. This choice may at first seem like cheating, since the entire operational aspect of the representation is ignored. The correctness of that aspect, however, is guaranteed by its dependent typing, which anchors its meaning (no matter how cleverly implemented) to the non-operational type index. For the language implementations of Section 4, this anchoring is guaranteed by the type of ⟦_⟧ in Figures 4 and 8. (Similarly, for the decidable type constructors in Figure 3, correctness is guaranteed by the types of the yes and no constructors in the definition of Dec in Section 3.) Given this choice of equivalence, properties of index types easily lift to properties of existential types, as shown in Figure 13. (These definitions are partially inferable as well, e.g., from `prop₃′ R _ _ _ = R _ _ _`.) These definitions ease specification of algebraic instances for existentially wrapped types from corresponding instances for their indices. As examples, we can lift the commutative semiring of types to wrappings of the decidable counterparts from Figure 3 in Section 3, and lift both language semirings to both decidable implementations in Section 4, as in Figure 14.

```
prop₁ : (∀ a → P a) → ∀ ((a , _) : ∃ F) → P a
prop₁ P (a , _) = P a

prop₂ : (∀ a b → Q a b) → ∀ ((a , _) (b , _) : ∃ F) → Q a b
prop₂ Q (a , _) (b , _) = Q a b

prop₃ : (∀ a b c → R a b c) → ∀ ((a , _) (b , _) (c , _) : ∃ F) → R a b c
prop₃ R (a , _) (b , _) (c , _) = R a b c
```

Fig. 13. Liftings of index properties to existential types

```
decCCS = mkClosedCommutativeSemiring ×-⊎-closedCommutativeSemiring
                                     Dec _⊎?_ _×?_ ⊥? ⊤? _✶?

symbolicCS₁ = mkCommutativeSemiring (∩-∪-commutativeSemiring _) Lang _∪_ _∩_ ∅ 𝒰
  where open Symbolic

symbolicCS₂ = mkClosedSemiring ∗-∪-closedSemiring Lang _∪_ _∗_ ∅ 1 _☆
  where open Symbolic

automaticCS₁ = mkCommutativeSemiring (∩-∪-commutativeSemiring _) (Lang ∞) _∪_ _∩_ ∅ 𝒰
  where open Automatic

automaticCS₂ = mkClosedSemiring ∗-∪-closedSemiring (Lang ∞) _∪_ _∗_ ∅ 1 _☆
  where open Automatic
```

Fig. 14. Examples of algebraic instances transported from indices to existential wrappings

7 RELATED WORK

Most work on formal languages in functional programming has been in parsing, particularly via parsing combinators [Hutton and Meijer 1996; Leijen and Meijer 2001; Swierstra 2008]. Such work typically concentrates on usability, composability, and sometimes efficiency, rather than on simple semantic specification and verifiable correctness.

There has also been some work using type theory to capture languages as type-level predicates much as in Section 1:

- Agular and Mannaa [2009] also defined a non-indexed algebraic type of regular expressions. They also *defined* indexed types for ν and δ on regular expressions rather than defining the meanings of those operations in terms of languages and then proving properties of them. They proved a consistency property about δ, but apparently not about ν, and they note their method's "inability to prove the non-membership of some string in a given language".
- Doczkal et al. [2013] formalized regular languages constructively in Coq, also in terms of a type-level membership predicate. They then proved several classic results about finite automata, including minimization and relationships to Nerode and Myhill partitions.
- Firsov and Uustalu [2013] formalized regular expressions in Agda with a corresponding inductive type of language membership proofs and related regular expressions to nondeterministic finite automata (NFAs) represented as boolean matrices. They also defined operations

on that representation corresponding to those on regular expressions and proved soundness and completeness of the NFA representation with respect to regular language membership.

- Korkut et al. [2016] defined a non-indexed algebraic data type of regular expressions together with an indexed inductive type of proofs of membership of strings in the language denoted by the regular expressions and a corresponding function that matches an implicitly concatenated stack of regular expressions. Special attention was paid to ensuring and proving termination, particularly in regard to languages that contain the empty string. Addressing both involved defunctionalization and translating between general regular expressions and a restricted "standard" form.
- Traytel [2017] defined formal languages as tries, with operations via primitive corecursion and reasoning via coinduction, all in the Isabelle/HOL proof assistant. There appears to have been no specification higher level (less operational) than tries, leading to some rather complex definitions (which, as *definitions*, cannot be proved correct). Language concatenation was particularly delicate. For the same reason, proofs of even basic properties involved coinduction. The authors pointed out, however, that the coinductive trie formulation enabled better proof automation than with sets of strings.
- Abel [2016] also formulated languages via tries and in particular noted the termination issue mentioned in Section 4.3 and provided the sized-types solution adopted in this paper. Relationships to automata theory and coalgebraic notions were also explored in depth. Trie look-up yielded boolean values rather than propositions, and there was no simpler specification of languages with respect to which the trie implementation could be specified and proved consistent, so every operation was *defined* coinductively. The concatenation and closure (star) cases were especially delicate. Equality on tries was defined by strong bisimilarity, and thus properties required corresponding machinery to state and prove.
- Baanen and Swierstra [2020] also explored correctness of regular expression matching in Agda, but from the perspective of *effects* and their predicate transformer semantics. The authors shared a common starting point with Korkut et al. [2016] and addressed differentiation on regular expressions, while separating termination from partial correctness. Their regular expression matching function targets a free monad (later to be interpreted with suitable effect semantics) rather than propositions.

Of the previous work involving language derivatives, none appear to have based their formal specification and proofs on the essential, non-inductive, nearly-trivial definition in terms of functions of lists from Section 2. Abel [2016] mentioned this definition informally as motivation for the more complex, coinductively defined operations on tries. Brzozowski [1964] also made clear mention of the underlying meaning on sets of strings in his original work on regular expression derivatives. Moreover, none of these previous investigations appear to have addressed proof isomorphism (distinguishing multiple parsings/explanations), and correspondingly all use transformations based the less precise laws originally discovered by Brzozowski [1964]. Those laws were based on the assumption that union ("addition" in both language semirings of Section 6.1) is idempotent, which is not true for type-level language predicates related by proof isomorphism (rather than mere logical equivalence). The lack of idempotence is crucial for going beyond recognizers to parsers (which are essentially *proof-relevant* recognizers), in which we might want to extract more than one proof (parsing). The difference is mainly visible in the ν and δ lemmas for $P * Q$ and $P^{\star}$ in Figure 2.

REFERENCES

Andreas Abel. 2008. Semi-continuous sized types and termination. *Logical Methods in Computer Science* 4, 2 (Apr 2008).
Andreas Abel. 2016. Equational reasoning about formal languages in coalgebraic style. draft.

Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* 26 (2016).

Agda Team. 2020. The Agda standard library.

Alexandre Agular and Bassel Mannaa. 2009. *Regular expressions in Agda.* Technical Report. Chalmers University.

Anne Baanen and Wouter Swierstra. 2020. Combining predicate transformer semantics for effects: A case study in parsing regular languages. In *Proceedings Eighth Workshop on Mathematically Structured Functional Programming.*

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda — A functional language with dependent types. In *Theorem Proving in Higher Order Logics.*

Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *J. ACM* 11 (1964), 481–494.

N. Chomsky and M.P. Schützenberger. 1959. The algebraic theory of context-free languages. In *Computer Programming and Formal Systems.* Studies in Logic and the Foundations of Mathematics, Vol. 26. 118–161.

Richard H. Connelly and F. Lockwood Morris. 1995. A generalization of the trie data structure. *Mathematical Structures in Computer Science* 5, 3 (1995).

Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. 2013. A constructive theory of regular languages in Coq. In *Certified Programs and Proofs, Third International Conference (CPP 2013).*

Brijesh Dongol, Ian J. Hayes, and Georg Struth. 2016. Convolution as a unifying concept: Applications in separation logic, interval calculi, and concurrency. *ACM Transactions on Computational Logic* (Feb. 2016), 15:1–15:25.

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 4 (Sept. 2018), 29 pages.

Conal Elliott. 2019. Generalized convolution and efficient language recognition. *CoRR* abs/1903.10677 (2019).

Denis Firsov and Tarmo Uustalu. 2013. Certified parsing of regular languages. In *Proceedings of the Third International Conference on Certified Programs and Proofs, Volume 8307.* Springer-Verlag.

Jonathan S. Golan. 2005. Some recent applications of semiring theory. In *International Conference on Algebra in Memory of Kostia Beidar.*

Joshua Goodman. 1998. *Parsing Inside-Out.* Ph.D. Dissertation. Harvard University.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics* 25, 4 (Dec. 1999), 573–605.

Andreas Griewank. 1989. On Automatic Differentiation. In *In Mathematical Programming: Recent Developments and Applications.*

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics.

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

Graham Hutton and Erik Meijer. 1996. Monadic parser combinators.

Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc.

Joomy Korkut, Maksim Trifunovski, and Daniel R. Licata. 2016. Intrinsic verification of a regular expression matcher. (2016). unpublished draft.

Daan Leijen and Erik Meijer. 2001. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 1–20.

Yudong Liu. 2004. *Algebraic Foundation of Statistical Parsing Semiring Parsing.* Ph.D. Dissertation. Simon Fraser University.

Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory.* Vol. 9. Bibliopolis Naples.

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (Jan. 2008), 1–13.

Matthew Might and David Darais. 2010. Yacc is dead. *CoRR* abs/1010.5023 (2010).

Ulf Norell. 2008. Dependently typed programming in Agda. In *Revised Lectures of the Sixth International Spring School on Advanced Functional Programming (Lecture Notes in Computer Science).*

Klaus Ostermann and Julian Jabs. 2018. Dualizing generalized algebraic data types by matrix transposition. In *Programming Languages and Systems—27th European Symposium on Programming, ESOP 2018, Proceedings*, Amal Ahmed (Ed.).

S Doaitse Swierstra. 2008. Combinator parsing: A short tutorial. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development.* 252–300.

Axel Thue. 1912. *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen.* Jacob Dybwad.

Dmitriy Traytel. 2017. Formal languages, formally and coinductively. *Logical Methods in Computer Science* Volume 13, Issue 3 (Sept. 2017).

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015).