Symbolic and Automatic Differentiation of Languages

CONAL ELLIOTT, USA

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.

CCS Concepts: • **Theory of computation → Type theory**; **Regular languages**; **Program verification**; *Parsing*.

Additional Key Words and Phrases: program calculation; language derivatives; tries

ACM Reference Format:
Conal Elliott. 2021. Symbolic and Automatic Differentiation of Languages. *Proc. ACM Program. Lang.* 5, ICFP, Article 78 (August 2021), 18 pages. https://oadoi.org/10.1145/3473583

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 ($\mathbf{1}$, containing only the empty string), single-character languages ($`\ c$), Kleene star ($P^{\star}$), 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,

Author's address: Conal Elliott, USA, conal@conal.net.

2475-1421/2021/8-ART78
https://oadoi.org/10.1145/3473583

$$\mathsf{Lang} = A^{\star} \to \mathsf{Set}\ \ell$$

$$\begin{array}{ll}
\varnothing\ w = \bot & (P \cup Q)\ w = P\ w \uplus Q\ w \\
\mathcal{U}\ w = \top & (P \cap Q)\ w = P\ w \times Q\ w \\
\mathbf{1}\ w = w \equiv [\,] & (P * Q)\ w = \exists\ \lambda\ (u\ ,\ v) \to (w \equiv u +\!\!+ v) \times P\ u \times Q\ v \\
`\ c\ w = w \equiv [\ c\ ] & (P^{\star})\ w = \exists\ \lambda\ ws \to (w \equiv \mathsf{concat}\ ws) \times \mathsf{All}\ P\ ws \\
(s \cdot P)\ w = s \times P\ w &
\end{array}$$

Fig. 1. Languages as type-level predicates

- 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.
- For types (propositions) A and B, 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 refl : $\forall\ x \to x \equiv x$.

In the definition of $P^{\star}$, 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 ✶              foldr : (B → X → X) → X → B ✶ → X
concat = foldr _++_ []               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 $v \in Q$. Figure 2 contains some examples of languages (left) and corresponding membership proofs (right).

The use of type-level predicates makes for simple, direct specification, including the use of existential quantification (as in Figure 1). 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,

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

```
a∪b : Lang                  _ : a∪b [ 'b' ]
a∪b = ‘ 'a' ∪ ‘ 'b'         _ = inj₂ refl

a∗b : Lang                  _ : a∗b ('a' :: 'b' :: [])
a∗b = ‘ 'a' ∗ ‘ 'b'         _ = ([ 'a' ] , [ 'b' ]) , refl , refl , refl

a∪b☆ : Lang                 _ : a∪b☆ ('a' :: 'b' :: 'a' :: [])
a∪b☆ = a∪b ☆                _ = [ 'a' ] :: [ 'b' ] :: [ 'a' ] :: []
                              , refl
                              , inj₁ refl :: inj₂ refl :: inj₁ refl :: []
```

Fig. 2. Language examples[2]

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).
- 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.

[2] Some explanatory comments:

- Agda variable names can contain most characters other than spaces and parentheses, e.g., "$a{\cup}b^{☆}$".
- The alphabet type (A in Figure 1) used for these examples is Char, i.e., characters.
- Single-item lists are notated with square brackets.
- Variables defined but not used can be replaced by the anonymous pattern "_" (as in proofs on the right).
- The inj₁ and inj₂ functions are left and right injections into a sum type, corresponding to the left and right introduction forms for logical disjunction.
- Pairing is right-associative. Since the second and third examples are existential propositions, their inhabiting proofs are dependent pairs. Each comprises a division into substrings and a proof that the substrings concatenate to the given test string (the language's argument), along with language membership proofs for each substring.
- Equality proofs in these examples are by normalization and hence given by refl.

Although many proofs are elided below, the source code for this paper is freely available and contains fully verified Agda code with no postulates [Elliott 2021]. The type signatures and definitions displayed in the paper are extracted from this source code 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, $\mathscr{D}\ P\ u$ 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[3,4]

```
ν∘𝒟 : ν ∘ 𝒟 f ≗ f
```

Moreover, the function 𝒟 distributes over [] and _++_:[5]

```
𝒟[] : 𝒟 f [] ≡ f

𝒟++ : 𝒟 f (u ++ v) ≡ 𝒟 (𝒟 f u) v
```

These facts suggest that 𝒟 can be computed one list element at a time via a more specialized operation, as is indeed the case:[6]

```
δ : (A ✶ → B) → A → (A ✶ → B)
δ f a = 𝒟 f [ a ]

𝒟foldl : 𝒟 f ≗ foldl δ 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 ν∘𝒟 and 𝒟foldl 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 ν∘𝒟 and 𝒟foldl capture the correctness of this state machine as a recognizer for the language P.

[3]Recall that Agda variable names can contain most characters other than spaces and parentheses, e.g., “ν∘𝒟” as declared here.

[4]For functions f and g, $f \doteq g$ is extensional equality, i.e., $\forall\ x \to f\ x \equiv g\ x$.

[5]In algebraic terms, argument-flipped 𝒟 is a monoid homomorphism from lists to endofunctions.

[6]Repeated δ application is expressed here via a standard left fold (with $X = A^{✶} \to B$ for 𝒟foldl):

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

$$
\begin{array}{llll}
\nu\varnothing & : \nu\ \varnothing \equiv \bot & \delta\varnothing & : \delta\ \varnothing\ a \equiv \varnothing \\
\nu\mathcal{U} & : \nu\ \mathcal{U} \equiv \top & \delta\mathcal{U} & : \delta\ \mathcal{U}\ a \equiv \mathcal{U} \\
\nu\cup & : \nu\ (P \cup Q) \equiv (\nu\ P \uplus \nu\ Q) & \delta\cup & : \delta\ (P \cup Q)\ a \equiv \delta\ P\ a \cup \delta\ Q\ a \\
\nu\cap & : \nu\ (P \cap Q) \equiv (\nu\ P \times \nu\ Q) & \delta\cap & : \delta\ (P \cap Q)\ a \equiv \delta\ P\ a \cap \delta\ Q\ a \\
\nu\cdot & : \nu\ (s \cdot P) \equiv (s \times \nu\ P) & \delta\cdot & : \delta\ (s \cdot P)\ a \equiv s \cdot \delta\ P\ a \\
\nu\mathbf{1} & : \nu\ \mathbf{1} \leftrightarrow \top & \delta\mathbf{1} & : \delta\ \mathbf{1}\ a \longleftrightarrow \varnothing \\
\nu\ast & : \nu\ (P \ast Q) \leftrightarrow (\nu\ P \times \nu\ Q) & \delta\ast & : \delta\ (P \ast Q)\ a \longleftrightarrow \nu\ P \cdot \delta\ Q\ a \cup \delta\ P\ a \ast Q \\
\nu^{\star} & : \nu\ (P^{\star}) \leftrightarrow (\nu\ P)^{\star} & \delta^{\star} & : \delta\ (P^{\star})\ a \longleftrightarrow (\nu\ P)^{\star} \cdot (\delta\ P\ a \ast P^{\star}) \\
\nu` & : \nu\ (`\ c) \leftrightarrow \bot & \delta` & : \delta\ (`\ c)\ a \longleftrightarrow (a \equiv c) \cdot \mathbf{1}
\end{array}
$$

Fig. 3. Properties of ν and δ for language operations

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 3. The correct-by-construction parsing algorithms in Section 4 are corollaries of these properties.

There are three relations involved in Figure 3: 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:

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

The equalities in Figure 3 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 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 $A \rightarrow \bot$):[7]

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

For compositionality, we will use a few operations that lift decidability of types to decidability of constructions on those types, as shown in Figure 4. Moreover, decidability lifts naturally from types to predicates:

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

$$\bot^? : \mathsf{Dec}\ \bot \qquad\qquad \top^? : \mathsf{Dec}\ \top$$
$$\bot^? = \mathsf{no}\ (\lambda\ ()) \qquad\qquad \top^? = \mathsf{yes}\ \mathsf{tt}$$

$$\begin{aligned}
&\_\uplus^?\_ : \mathsf{Dec}\ A \to \mathsf{Dec}\ B \to \mathsf{Dec}\ (A \uplus B) \\
&\mathsf{no}\ \neg a \uplus^? \mathsf{no}\ \neg b = \mathsf{no}\ [\ \neg a\ ,\ \neg b\ ] \\
&\mathsf{yes}\ a\ \ \uplus^? \mathsf{no}\ \neg b = \mathsf{yes}\ (\mathsf{inj}_1\ a) \\
&\_\qquad\ \ \uplus^? \mathsf{yes}\ b\ \ = \mathsf{yes}\ (\mathsf{inj}_2\ b)
\end{aligned}
\qquad
\begin{aligned}
&\_\times^?\_ : \mathsf{Dec}\ A \to \mathsf{Dec}\ B \to \mathsf{Dec}\ (A \times B) \\
&\mathsf{yes}\ a\ \ \times^? \mathsf{yes}\ b\ \ = \mathsf{yes}\ (a\ ,\ b) \\
&\mathsf{no}\ \neg a \times^? \mathsf{yes}\ b\ \ = \mathsf{no}\ (\neg a \circ \mathsf{proj}_1) \\
&\_\qquad\ \ \times^? \mathsf{no}\ \neg b = \mathsf{no}\ (\neg b \circ \mathsf{proj}_2)
\end{aligned}$$

$$\_\star^? : \mathsf{Dec}\ A \to \mathsf{Dec}\ (A\ \star)$$
$$\_\ \star^? = \mathsf{yes}\ []$$

Fig. 4. Compositional decidability[8]

$$\mathsf{Decidable}\ P = \forall\ x \to \mathsf{Dec}\ (P\ x)$$

With these definitions, we can succinctly 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 3) play an important role in decidability, namely that isomorphic types or predicates are equivalently decidable:

$$\_\triangleleft\_ : B \leftrightarrow A \to \mathsf{Dec}\ A \to \mathsf{Dec}\ B$$

$$\_\blacktriangleleft\_ : Q \longleftrightarrow P \to \mathsf{Decidable}\ P \to \mathsf{Decidable}\ Q$$

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 3 tell us how to decompose languages defined in terms of the vocabulary from Figure 1, while the definitions in Figure 4 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 structural inspection. 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.

[8] A few explanatory remarks:

- $\lambda\ ()$ is the unique function from $\bot$ to any type (here also $\bot$), sometimes called "absurd" or "$\bot$-elim".
- Recall that 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 $\mathsf{inj}_1\ a$ to $f\ a$ and $\mathsf{inj}_2\ b$ to $g\ b$.
- $\_\star^?$ reflects the fact that the empty list exists for every element type (even $\bot$).

```
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
```

```
                                           ⟦_⟧? : Lang P → Decidable P
ν : Lang P → Dec (◇ν P)                    ⟦ p ⟧?    []    = ν p
δ : Lang P → (a : A) → Lang (◇δ P a)       ⟦ p ⟧? (a ∷ w) = ⟦ δ p a ⟧? w
```

Fig. 5. Regular expressions (inductive)

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 apply the symbolic and automatic strategies respectively to languages. Both algorithms are corollaries of the lemmas in Figure 3 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, use the same names as in Section 1 for these new counterparts, and refer to the original versions via the module prefix "◇". The result is shown in Figure 5 along with a computable function ⟦_⟧? that converts a symbolic language representation to a computable parser. Note that the _◂_ operation from Section 3 has been reified as a new constructor alongside the more conventional language operators. The reason _◂_ must be part of the inductive representation is the same as the other constructors, namely so that the core lemmas (Figure 3) translate into an implementation in terms of that representation.

The *meaning* of any r : Lang P is defined simply as P, ignoring the representation r itself:

```
⟦_⟧ : Lang P → ◇Lang
⟦_⟧ {P} r = P
```

Decidable parsing relies only on ν and δ (via ⟦_⟧?), which are defined in Figure 6, as systematically derived from the lemmas of Figure 3. These clauses are meant to be read in column-major order, i.e., each column is a complete definition (two definitions of ten pattern-matching clauses

$$
\begin{aligned}
&\nu\ \varnothing = \bot^{?} && \delta\ \varnothing\ a = \varnothing \\
&\nu\ \mathcal{U} = \top^{?} && \delta\ \mathcal{U}\ a = \mathcal{U} \\
&\nu\ (p \cup q) = \nu\ p \uplus^{?} \nu\ q && \delta\ (p \cup q)\ a = \delta\ p\ a \cup \delta\ q\ a \\
&\nu\ (p \cap q) = \nu\ p \times^{?} \nu\ q && \delta\ (p \cap q)\ a = \delta\ p\ a \cap \delta\ q\ a \\
&\nu\ (s \cdot p) = s \times^{?} \nu\ p && \delta\ (s \cdot p)\ a = s \cdot \delta\ p\ a \\
&\nu\ \mathbf{1} = \nu\mathbf{1} \blacktriangleleft \top^{?} && \delta\ \mathbf{1}\ a = \delta\mathbf{1} \blacktriangleleft \varnothing \\
&\nu\ (p * q) = \nu{*} \blacktriangleleft (\nu\ p \times^{?} \nu\ q) && \delta\ (p * q)\ a = \delta{*} \blacktriangleleft (\nu\ p \cdot \delta\ q\ a \cup \delta\ p\ a * q) \\
&\nu\ (p^{\star}) = \nu^{\star} \blacktriangleleft (\nu\ p\ {\ast}^{?}) && \delta\ (p^{\star})\ a = \delta^{\star} \blacktriangleleft (\nu\ p\ {\ast}^{?} \cdot (\delta\ p\ a * p^{\star})) \\
&\nu\ (`\ a) = \nu` \blacktriangleleft \bot^{?} && \delta\ (`\ c)\ a = \delta` \blacktriangleleft ((a \stackrel{?}{=} c) \cdot \mathbf{1}) \\
&\nu\ (f \blacktriangleleft p) = f \blacktriangleleft \nu\ p && \delta\ (f \blacktriangleleft p)\ a = f \blacktriangleleft \delta\ p\ a
\end{aligned}
$$

Fig. 6. Symbolic differentiation (column-major/patterns)

each). Correctness is guaranteed by the types of ν and δ and so is proved automatically by type-checking Figure 6. (Recall from Section 3 that correctness is defined as constructing a term of type Decidable P for a given language P, as satisfied by the type of $[\![\_]\!]^{?}$.) We could thus use *any* definitions of ν and δ that type- and termination-check, although not many definitions would do so.

In Figure 6, note the role of the isomorphisms from Figure 3. 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 type of tries[9], while redefining the language-building vocabulary as functions on that data type defined by *copatterns* [Abel et al. 2013; Abel and Pientka 2016].[10] (This program duality pattern has been noted and investigated by Ostermann and Jabs [2018].) Again, we will use the same names

[9]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].

[10]Inductive types ("data") describe finitely large values and are processed recursively (as least fixed points) by pattern-matching clauses that decompose arguments. Dually, coinductive types ("codata") describe infinitely large values and are processed corecursively (as greatest fixed points) by copattern-matching clauses that compose results. Programs on inductive types are often proved by induction, while programs on coinductive types are often proved by coinduction. (As we will see in Section 6.2, however, the simple relationship to the specification in Section 2 allows many trivial correctness proofs, needing neither induction or coinduction.) See Gordon [1995] for a tutorial on theory and techniques of coinduction. Tries in general represent functions, with each trie datum position corresponding to a domain value. Even when each domain value is finitely large, there are often infinitely many of them (e.g., lists), so tries will naturally be infinite and thus more amenable to coinductive than inductive analysis.

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

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

```
∅    : 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
```

Fig. 7. Tries (coinductive)

```
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. 8. Automatic differentiation (row-major/copatterns)

as in Section 1, referring to the original versions via the module prefix "◇". The result is shown in Figure 7.

Decidable recognition again relies only on ν and δ, which are defined for each language operation in Figure 8 (though with a problem to be noted and fixed in the next paragraph). These clauses are meant to be read in row-major order, i.e., each row is a complete definition (ten definitions of

```
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. 9. Sized tries (coinductive)

two copattern-matching clauses each). Note that the clauses in Figure 8 are syntactically identical to those in Figure 6 but are organized dually. (The compiler-generated syntax coloring differs to reflect the changed interpretation of the definitions.)

The proof of correctness (defined by ⟦_⟧ and ⟦_⟧?, defined as before) is *almost* accomplished by type- and termination-checking, but a technical problem arises. The δ (p ⋆ q) clause does not satisfy Agda's termination checker, which cannot see that the argument δ p a in the recursive use of _⋆_ is in some sense smaller than p. (Figure 8 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 : 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 9. Decidable recognition is defined exactly as with unsized tries (Figure 8), with the sole exception of removing the compiler pragma that suppressed termination checking. This time the compiler successfully proves *total* correctness (including termination).

To see the parallel with automatic differentiation (AD) more clearly, 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
```

$$\mathsf{Pred}\ A = A \to \mathsf{Set}\ \ell$$

$$\begin{array}{ll}
\mathsf{pure}^o : \mathsf{Set}\ \ell \to \mathsf{Pred}\ A & \mathsf{pure}^i : A \to \mathsf{Pred}\ A \\
\mathsf{pure}^o\ x\ a = x & \mathsf{pure}^i\ x\ a = a \equiv x \\
\mathsf{map}^o : (\mathsf{Set}\ \ell \to \mathsf{Set}\ \ell) \to (\mathsf{Pred}\ A \to \mathsf{Pred}\ A) & \mathsf{map}^i : (A \to B) \to (\mathsf{Pred}\ A \to \mathsf{Pred}\ B) \\
\mathsf{map}^o\ g\ P\ a = g\ (P\ a) & \mathsf{map}^i\ g\ P\ b = \exists\ \lambda\ a \to b \equiv g\ a \times P\ a \\
\mathsf{map}^o{}_2 : (\mathsf{Set}\ \ell \to \mathsf{Set}\ \ell \to \mathsf{Set}\ \ell) \to & \mathsf{map}^i{}_2 : (A \to B \to C) \to \\
\quad (\mathsf{Pred}\ A \to \mathsf{Pred}\ A \to \mathsf{Pred}\ A) & \quad (\mathsf{Pred}\ A \to \mathsf{Pred}\ B \to \mathsf{Pred}\ C) \\
\mathsf{map}^o{}_2\ h\ P\ Q\ a = h\ (P\ a)\ (Q\ a) & \mathsf{map}^i{}_2\ h\ P\ Q\ c = \exists\ \lambda\ (a\ ,\ b) \to c \equiv h\ a\ b \times P\ a \times Q\ b
\end{array}$$

Fig. 10. Predicate operations

The potential work-sharing version exploits the close relationship between f and $\mathscr{D}\ f$:

$$\begin{array}{l}
\hat{\mathscr{D}} : (A^{\star} \to B) \to A^{\star} \to B \times (A^{\star} \to B) \\
\hat{\mathscr{D}}\ f\ u = \mathsf{let}\ f' = \mathsf{foldl}\ \delta\ f\ u\ \mathsf{in}\ \nu\ f'\ ,\ f'
\end{array}$$

Equivalence follows from a simple inductive argument, thanks to lemmas $\nu{\circ}\mathscr{D}$ and $\mathscr{D}\mathsf{foldl}$ of Section 2:

$$\begin{array}{l}
\mathscr{D}'{\equiv}\hat{\mathscr{D}} : \mathscr{D}'\ f \doteq \hat{\mathscr{D}}\ f \\
\mathscr{D}'{\equiv}\hat{\mathscr{D}}\ \ \ [\,]\ \ \ \ \ \ = \mathsf{refl} \\
\mathscr{D}'{\equiv}\hat{\mathscr{D}}\ (a :: as) = \mathscr{D}'{\equiv}\hat{\mathscr{D}}\ as
\end{array}$$

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): $\varnothing$, $\mathscr{U}$, $\_\cup\_$, and $\_\cap\_$. The other transforms the domain (lists): $\mathbf{1}$, $\_{*}\_$, $\_^{☆}$, and $`$. The first category applies to predicates over all types (not just over lists). Within the second category, $\mathbf{1}$, $\_{*}\_$, and $\_^{☆}$ can apply to any monoid, while $`$ is specific to lists.

Figure 10 shows two parallel collections of predicate operations, each covariantly transforming the codomain (left) or domain (right).[11] These generalized operations then specialize to language operations as in Figure 11, which adds language complement ($\complement$).[12]

The ν and δ lemmas in Figure 3 for codomain (but not domain) operations are easily generalized as shown in Figure 12, 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.

[11]The types of operations on the left can be further relaxed from $\mathsf{Set}\ \ell$ 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.

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

$$\mathsf{Lang} = \mathsf{Pred}\ (A^{\star})$$

$$\begin{aligned}
\varnothing &= \mathsf{pure}^{o}\ \bot & \mathbf{1} &= \mathsf{pure}^{i}\ \varepsilon \\
\mathcal{U} &= \mathsf{pure}^{o}\ \top & & \\
\_\cup\_ &= \mathsf{map}^{o}{}_{2}\ \_\uplus\_ & \_{*}\_ &= \mathsf{map}^{i}{}_{2}\ \_\bullet\_ \\
\_\cap\_ &= \mathsf{map}^{o}{}_{2}\ \_\times\_ & P^{\star} &= \mathsf{map}^{i}\ (\mathsf{foldr}\ \_\bullet\_\ \varepsilon)\ (\mathsf{All}\ P) \\
\_\cdot\_\ s &= \mathsf{map}^{o}\ (s \times\_) & \text{`}\ c &= \mathsf{pure}^{i}\ [\ c\ ] \\
\complement &= \mathsf{map}^{o}\ \neg\_ & &
\end{aligned}$$

Fig. 11. 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. 12. Properties of ν and δ for predicate codomain operations

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 providing "one" and "multiplication", and (c) the relationship between them, namely that multiplication distributes over addition and zero.[13] In the first predicate semiring, which is *commutative* (i.e., multiplication commutes), zero and addition are $\varnothing$ and $\_\cup\_$, while one and multiplication are $\mathcal{U}$ 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 Agda's standard library.[14] They are both closed as well. For booleans, closure maps both false

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

[14]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

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 to the identity $\lambda\ a \to \varepsilon$. All of the laws transfer from B to $A \to B$. 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 : Pred A is a vector in the semimodule 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 $\text{pure}^{i}\ a$ 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 11).

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 11. 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 11. 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 formal statements and proofs of these properties are included in this paper's Agda source code [Elliott 2021].

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—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 from Specification to Implementations

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

accounts for the difference between the ν and δ lemmas for $P * Q$ and $P^{\star}$ in Figure 3 and the corresponding rules in previous work, as mentioned in Section 7.

```
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. 13. Existential wrappers

typed. For example, each instance of the Monoid abstraction involves a single type τ with a binary operation $\_\bullet\_ : \tau \to \tau \to \tau$ and identity value $\varepsilon : \tau$. This recipe accommodates the language operations in Figures 1 and 11 and their generalizations in Figure 10, but not the dependently typed, *indexed* versions of languages and their operations in Figures 5 and 9, nor decidable types and their vocabulary in Figure 4.

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 $F : A \to$ Set ℓ, 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 5:

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

Referring to the type and operations of Figures 1 and 5 (for the inductive language representation) 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
```

Likewise for the coinductive language representation in Figure 9. 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 13, 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 *semantic*—i.e., 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 types of $[\![\_]\!]$ and $[\![\_]\!]^?$ in Figures 5 and 9. (Similarly, for the decidable type constructors in Figure 4, 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

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

existential types, as shown in Figure 14. (These definitions are partially inferable as well, e.g., from $\mathsf{prop}_3{}'\ R\,\_\,\_\,\_ = R\,\_\,\_\,\_$.) 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 4 in Section 3, and lift both language semirings to both decidable implementations in Section 4, as in Figure 15.

7 RELATED WORK

The shift from languages as sets of strings to type-level predicates (i.e., proof relevance or "parsing") is akin to weighted automata [Schützenberger 1961; Droste and Kuske 2019] and more generally to semiring-based parsing [Chomsky and Schützenberger 1959; Goodman 1998, 1999; Liu 2004], noting that types ("Set" in Agda) form a (commutative) semiring (up to isomorphism). In particular, Lombardy and Sakarovitch [2005] investigated Brzozowski-style derivatives in this more general setting, formalizing and generalizing the work of Antimirov [1996], who decomposed derivatives into simpler components ("partial derivatives") that sum to the full derivative and lead to efficient recognizers in the form of nondeterministic automata.

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.

The main contributions of this paper are to formalization and mechanization of language recognition and parsing. There has 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 on 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 3. As mentioned at the end of Section 3, one can simplify the development above somewhat by replacing proof isomorphism with the coarser notion of logical equivalence. Doing so restores union idempotence (modulo equivalence), losing proof relevance and thus replacing parsing by recognition. One might then also prove that the set of derivatives of a regular language is finite (again, modulo equivalence).

REFERENCES

Andreas Abel. 2008. Semi-continuous sized types and termination. *Logical Methods in Computer Science* 4, 2 (Apr 2008). https://arxiv.org/abs/0804.0876

Andreas Abel. 2016. Equational reasoning about formal languages in coalgebraic style. http://www.cse.chalmers.se/~abela/jlamp17.pdf draft.

Andreas Abel and Brigitte Pientka. 2016. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming* 26 (2016). https://www.cs.mcgill.ca/~bpientka/papers/jfp15.pdf

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming infinite structures by observations *(POPL '13)*. 27–38. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.295.8056

Agda Team. 2020. The Agda standard library. https://github.com/agda/agda-stdlib

Alexandre Agular and Bassel Mannaa. 2009. *Regular expressions in Agda.* Technical Report. Chalmers University. https://itu.dk/people/basm/report.pdf

Valentin Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science* 155, 2 (1996), 291–319. https://www.sciencedirect.com/science/article/pii/0304397595001824

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*. https://arxiv.org/abs/2005.00197

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*. http://www.cse.chalmers.se/~ulfn/papers/tphols09/tutorial.pdf

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). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.902.7768

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).* https://www.ps.uni-saarland.de/~doczkal/regular/ConstructiveRegularLanguages.pdf

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. https://bura.brunel.ac.uk/bitstream/2438/12133/1/Fulltext.pdf

Manfred Droste and Dietrich Kuske. 2019. Weighted automata. (Jan 2019). https://eiche.theoinf.tu-ilmenau.de/person/kuske/public_html/weiterleitung.html?Submitted/weighted.pdf unpublished.

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 4 (Sept. 2018), 29 pages. http://conal.net/papers/essence-of-ad/

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

Conal Elliott. 2021. Source repository for "Symbolic and automatic differentiation of languages". https://github.com/conal/paper-2021-language-derivatives

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. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.571.724

Jonathan S. Golan. 2005. Some recent applications of semiring theory. In *International Conference on Algebra in Memory of Kostia Beidar*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.318.6696

Joshua Goodman. 1998. *Parsing Inside-Out*. Ph.D. Dissertation. Harvard University. https://arxiv.org/abs/cmp-lg/9805007

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics* 25, 4 (Dec. 1999), 573–605. http://www.aclweb.org/anthology/J99-4004

Andrew D. Gordon. 1995. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*. 78–95. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.3914

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. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.4069

Graham Hutton and Erik Meijer. 1996. Monadic parser combinators. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.1678

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). https://dlicata.wescreates.wesleyan.edu/pubs/ktl16regexp/ktl16regexp.pdf unpublished draft.

Daan Leijen and Erik Meijer. 2001. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 1–20. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.5200

Yudong Liu. 2004. *Algebraic Foundation of Statistical Parsing Semiring Parsing*. Ph.D. Dissertation. Simon Fraser University. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.167.6977

Sylvain Lombardy and Jacques Sakarovitch. 2005. Derivatives of rational expressions with multiplicity. *Theoretical Computer Science* 332, 1 (2005), 141–177. https://www.sciencedirect.com/science/article/pii/S0304397504007054

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. http://www.staff.city.ac.uk/~ross/papers/Applicative.html

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

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)*. http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf

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.). https://handin-ps.informatik.uni-tuebingen.de/publications/ostermann18dualizing.pdf

M.P. Schützenberger. 1961. On the definition of a family of automata. *Information and Control* 4, 2 (1961), 245–270. https://www.sciencedirect.com/science/article/pii/S001999586180020X

S Doaitse Swierstra. 2008. Combinator parsing: A short tutorial. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*. 252–300. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.184.7953

Axel Thue. 1912. *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen*. Jacob Dybwad. https://archive.org/details/skrifterutgitavv121chri

Dmitriy Traytel. 2017. Formal languages, formally and coinductively. *Logical Methods in Computer Science* Volume 13, Issue 3 (Sept. 2017). https://arxiv.org/abs/1611.09633

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015). https://homepages.inf.ed.ac.uk/wadler/topics/history.html#propositions-as-types