Generalized Convolution and Efficient Language Recognition

ANONYMOUS AUTHOR(S)

Convolution is a broadly useful operation with applications including signal processing, machine learning, probability, optics, polynomial multiplication, and efficient parsing. Usually, however, this operation is understood and implemented in more specialized forms, hiding commonalities and limiting usefulness. This paper formulates convolution in the common algebraic framework of semirings and semimodules and populates that framework with various representation types. One of those types is the grand abstract template and itself generalizes to the free semimodule monad. Other representations serve varied uses and performance trade-offs, with implementations calculated from simple and regular specifications.

Of particular interest is Brzozowski's method for regular expression matching. Uncovering the method's essence frees it from syntactic manipulations, while generalizing from boolean to weighted membership (such as multisets and probability distributions) and from sets to n-ary relations. The classic *trie* data structure then provides an elegant and efficient alternative to syntax.

Pleasantly, polynomial arithmetic requires no additional implementation effort, works correctly with a variety of representations, and handles multivariate polynomials and power series with ease. Image convolution also falls out as a special case.

ACM Reference Format:
Anonymous Author(s). 2019. Generalized Convolution and Efficient Language Recognition . 1, 1 (March 2019), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The mathematical operation of *convolution* combines two functions into a third—often written "$h = f * g$"—with each h value resulting from summing or integrating over the products of several pairs of f and g values according to a simple rule. This operation is at the heart of many important and interesting applications in a variety of fields [24].

- In image processing, convolution provides operations like blurring, sharpening, and edge detection [62].
- In machine learning convolutional neural networks (CNNs) allowed recognition of translation-independent image features [14, 34, 52].
- In probability, the convolution of the distributions of two independent random variables yields the distribution of their sum [20].
- In acoustics, reverberation results from convolving sounds and their echos [50]. Musical uses are known as "convolution reverb" [21, Chapter 4].
- The coefficients of the product of polynomials is the convolution of their coefficients [11].
- In formal languages, (generalized) convolution is language concatenation [12].

Usually, however, convolution is taught, applied, and implemented in more specialized forms, obscuring the underlying commonalities and unnecessarily limiting its usefulness. For instance,

XXXX-XXXX/2019/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Standard definitions rely on subtraction (which is unavailable in many useful settings) and are dimension-specific, while the more general form applies to any monoid [17, 59].
- Brzozowski's method of regular expression matching [8] appears quite unlike other applications and is limited to *sets* of strings (i.e., languages), leaving unclear how to generalize to variations like weighted membership (multisets and probability distributions) as well as n-ary *relations* between strings.
- Image convolution is usually tied to arrays and involves somewhat arbitrary semantic choices at image boundaries, including replication, zero-padding, and mirroring.

This paper formulates general convolution in the algebraic framework of semirings and semimodules, including a collection of types for which semiring multiplication is convolution. One of those types is the grand abstract template, namely the *monoid semiring*, i.e., functions from any monoid to any semiring. Furthermore, convolution reveals itself as a special case of an even more general notion—the *free semimodule monad*. The other types are specific representations for various uses and performance trade-offs, relating to the monoid semiring by simple denotation functions (interpretations). The corresponding semiring implementations are calculated from the requirement that these denotations be semiring homomorphisms, thus guaranteeing that the computationally efficient representations are consistent with their mathematically simple and general template.

An application of central interest in this paper is language specification and recognition, in which convolution specializes to language concatenation. Here, we examine a method by Brzozowski [8] for flexible and efficient regular expression matching, later extended to parsing context-free languages [42]. We will see that the essential technique is much more general, namely functions from lists to an arbitrary semiring. While Brzozowski's method involves repeated manipulation of syntactic representations (regular expressions or grammars), uncovering the method's essence frees us from such representations. Thue's tries provide a compelling alternative in simplicity and efficiency, as well as a satisfying confluence of classic techniques from the second and seventh decades of the twentieth century, as well as a modern functional programming notion: the cofree comonad.

Concretely, this paper makes the following contributions:

- Generalization of Brzozowski's algorithm from regular expressions representing sets of strings, to various representations of $[\,c\,] \to b$ where c is any type and b is any semiring, including n-ary functions and relations on lists (via currying).
- Demonstration that the subtle aspect of Brzozowski's algorithm (matching of concatenated languages) is an instance of generalized convolution.
- Specialization of the generalized algorithm to tries (rather than regular expressions), yielding a simple and apparently quite efficient implementation, requiring no construction or manipulation of syntactic representations.
- Observation that Brzozowski's key operations on languages generalize to the comonad operations of the standard function-from-monoid comonad and its various representations (including generalized regular expressions). The trie representation is the cofree comonad, which memoizes functions from the free monoid (lists).
- Application and evaluation of a simple memoization strategy encapsulated in a familiar functor, resulting in dramatic speed improvement.

2 MONOIDS, SEMIRINGS AND SEMIMODULES

The ideas in this paper revolve around a small collection of closely related algebraic abstractions, so let's begin by introducing these abstractions along with examples.

2.1 Monoids

The simplest abstraction we'll use is the monoid, expressed in Haskell as follows:

$$\begin{array}{l}\textbf{class}\ \mathit{Monoid}\ a\ \textbf{where}\\ \quad \varepsilon \quad :: a\\ \quad (\diamond) :: a \to a \to a\\ \quad \textbf{infixr}\ 6\ \diamond\end{array}$$

The monoid laws require that $(\diamond)$ be associative and that ε is its left and right identity, i.e.,

$$\begin{array}{l}(u \diamond v) \diamond w = u \diamond (v \diamond w)\\ \varepsilon \diamond v = v\\ u \diamond \varepsilon = u\end{array}$$

One monoid especially familiar to functional programmers is lists with append:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ [a]\ \textbf{where}\\ \quad \varepsilon \quad = [\,]\\ \quad (\diamond) = (+\!\!+)\end{array}$$

Natural numbers form a monoid under addition and zero. These two monoids are related via the function $\mathit{length} :: [a] \to \mathbb{N}$, which not only maps lists to natural numbers, but does so in a way that preserves monoid structure. This pattern is common and useful enough to have a name [60]:

Definition 1. A function h from one monoid to another is called a *monoid homomorphism* when it satisfies the following properties:

$$\begin{array}{l}h\ \varepsilon = \varepsilon\\ h\ (u \diamond v) = h\ u \diamond h\ v\end{array}$$

A fancier monoid example is functions from a type to itself, also known as *endofunctions*, for which ε is the identity function, and $(\diamond)$ is composition:

$$\textbf{newtype}\ \mathit{Endo}\ a = \mathit{Endo}\ (a \to a)$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ (\mathit{Endo}\ a)\ \textbf{where}\\ \quad \varepsilon = \mathit{Endo}\ \mathit{id}\\ \quad \mathit{Endo}\ g \diamond \mathit{Endo}\ f\ = \mathit{Endo}\ (g \circ f)\end{array}$$

A modest generalization of Cayley's theorem states that every monoid is isomorphic to a monoid of endofunctions [4]. This embedding is useful for turning quadratic-time algorithms linear [25, 58].

$$\begin{array}{ll}\mathit{toEndo} :: \mathit{Monoid}\ a \Rightarrow a \to \mathit{Endo}\ a & \mathit{fromEndo} :: \mathit{Monoid}\ a \Rightarrow \mathit{Endo}\ a \to a\\ \mathit{toEndo}\ a = \mathit{Endo}\ (\lambda\, z \to a \diamond z) & \mathit{fromEndo}\ (\mathit{Endo}\ f) = f\ \varepsilon\end{array}$$

The *toEndo* embedding provides another example of a monoid homomorphism [3].

2.2 Additive Monoids

While $(\diamond)$ must be associative, it needn't be commutative. Commutative monoids, however, will play an important role in this paper as well. For clarity and familiarity, it will be convenient to use the name "$(+)$" instead of "$(\diamond)$" and refer to such monoids as "additive":

$$\begin{array}{l}\textbf{class}\ \mathit{Additive}\ b\ \textbf{where}\\ \quad 0 \quad :: b\end{array}$$

```
    (+) :: b → b → b
    infixl 6 +
```

The *Additive* laws are the same as for *Monoid* (translating ε and $(\diamond)$ to 0 and $(+)$), together with commutativity:

$$
\begin{aligned}
&(u + v) + w = u + (v + w)\\
&0 + v = v\\
&u + 0 = u\\
&u + v = v + u
\end{aligned}
$$

Unlike lists with append, natural numbers form a *additive* monoid. Another example is functions with pointwise addition, with any domain and with any *additive* codomain:

```
instance Additive b ⇒ Additive (a → b) where
  0 = λ a → 0
  f + g = λ a → f a + g a
```

Additive monoids have their form of homomorphism:

Definition 2. A function h from one additive monoid to another is an *additive monoid homomorphism* if it satisfies the following properties:

$$
\begin{aligned}
&h\ 0 = 0\\
&h\ (u + v) = h\ u + h\ v
\end{aligned}
$$

Curried function types of *any number* of arguments (and additive result type) are additive, thanks to repeated application of this instance. In fact,

Theorem 1 (Proved in [3, Appendix A])**.** Currying and uncurrying are additive monoid homomorphisms.

2.3 Semirings

The natural numbers form a monoid in two familiar ways: addition and zero, and multiplication and one. Moreover, these monoids interact usefully in two ways: multiplication distributes over addition, and multiplication by zero (the additive identity) yields zero (i.e., "annihilates"). Similarly, *linear* endofunctions and their various representations (e.g., square matrices) forms a monoid via addition and via composition, with composition distributing over addition, and composition with zero yielding zero. In both examples, addition commutes; but while natural number multiplication commutes, composition does not. The vocabulary and laws these examples share is called a *semiring* (distinguished from a ring by dropping the requirement of additive inverses):

```
class Additive b ⇒ Semiring b where
  1   :: b
  (∗) :: b → b → b
  infixl 7 ∗
```

The laws, in addition to those for *Additive* above, include multiplicative monoid, distribution, and annihilation:

$$
\begin{aligned}
&u * 0 = 0 && (u * v) * w = u * (v * w)\\
&0 * v = 0 && \\
&1 * v = v && p * (q + r) = p * q + p * r\\
&u * 1 = u && (p + q) * r = p * r + q * r
\end{aligned}
$$

Definition 3. A function h from one semiring to another is a *semiring homomorphism* if it is an additive monoid homomorphism (Definition 2) and satisfies the following additional properties:

$$h\ 1 = 1$$
$$h\ (u * v) = h\ u * h\ v$$

As mentioned, numbers and various linear endofunction representations form semirings. A simpler example is the semiring of booleans, with disjunction as addition and conjunction as multiplication (though we could reverse roles):

```
instance Additive Bool where        instance Semiring Bool where
  0   = False                         1   = True
  (+) = (∨)                           (∗) = (∧)
```

An example of a semiring homomorphism is testing natural numbers for positivity [3]. There is a more fundamental example we will have use for later:

Theorem 2 (Proved in [3, Appendix A])**.** Currying and uncurrying are semiring homomorphisms.

2.4 Star Semirings

The semiring operations allow all *finite* combinations of addition, zero, multiplication, and one. It's often useful, however, to form infinite combinations, particularly in the form of Kleene's "star" (or "closure") operation:

$$p^* = \sum_i p^i \quad \text{-- where } p^0 = 1 \text{, and } p^{n+1} = p * p^n.$$

Another characterization is as a solution to either of the following semiring equations:

$$p^* = 1 + p * p^* \qquad\qquad p^* = 1 + p^* * p$$

which we will take as a laws for a new abstraction, as well as a default recursive implementation:

```
class Semiring b ⇒ StarSemiring b where
  ·* :: b → b
  p* = 1 + p ∗ p*
```

Sometimes there are more appealing alternative implementations. For instance, when subtraction and division are available, we can instead define $p^* = (1 - p)^{-1}$ [11].

Predictably, there is a notion of homomorphisms for star semirings:

Definition 4. A function h from one star semiring to another is a *star semiring homomorphism* if it is a semiring homomorphism (Definition 3) and satisfies the additional property $h\ (p^*) = (h\ p)^*$.

One simple example of a star semiring (also known as a "closed semiring" [11, 35]) is booleans:

```
instance StarSemiring Bool where b* = 1   -- = 1 ∨ (b ∧ b*)
```

A useful property of star semirings is that recursive affine equations have solutions [3, 11]:

Lemma 3. In a star semiring, the affine equation $p = b + m * p$ has solution $p = m^* * b$.

2.5 Semimodules

As fields are to vector spaces, rings are to modules, and semirings are to *semimodules*. For any semiring s, a *left* s*-semimodule* b is a additive monoid whose values can be multiplied by s values on the left. Here, s plays the role of "scalars", while b plays the role of "vectors".

```
class (Semiring s, Additive b) ⇒ LeftSemimodule s b | b → s where
  (·) :: s → b → b
```

In addition to the laws for the additive monoid b and the semiring s, we have the following laws specific to left semimodules: [17]:

$$(s * t) \cdot b = s \cdot (t \cdot b) \qquad 1 \cdot b = b$$
$$(s + t) \cdot b = s \cdot b + t \cdot b \qquad 0 \cdot b = 0$$
$$s \cdot (b + c) = s \cdot b + s \cdot c$$

As usual, we have a corresponding notion of homomorphism, which is more commonly referred to as "linearity":

Definition 5. A function h from one left s-semimodule to another is a *left s-semimodule homomorphism* if it is an additive monoid homomorphism (Definition 2) and satisfies the additional property $h\ (s \cdot b) = s \cdot h\ b$.

Familiar s-semimodule examples include various containers of s values, including single- or multi-dimensional arrays, lists, infinite streams, sets, multisets, and trees. Another, of particular interest in this paper, is functions from any type to any semiring:

```
instance LeftSemimodule s (a → s) where s · f = λ a → s ∗ f a
```

If we think of $a \to s$ as a "vector" of s values, indexed by a, then $s \cdot f$ scales each component of the vector f by s.

There is an important optimization to be made for scaling. When $s = 0$, $s \cdot p = 0$, so we can discard p entirely. This optimization applies quite often in practice, for instance with languages, which tend to be sparse. Another optimization (though less dramatically helpful) is $1 \cdot p = p$. Rather than burden each *LeftSemimodule* instance with these two optimizations, let's define $(\cdot)$ via a more primitive $(\hat{\cdot})$ method:

```
class (Semiring s, Additive b) ⇒ LeftSemimodule s b | b → s where
  (·̂) :: s → b → b

infixr 7 ·
(·) :: (Additive b, LeftSemimodule s b, IsZero s, IsOne s) ⇒ s → b → b
s · b | isZero s    = 0
      | isOne s     = b
      | otherwise = s ·̂ b
```

The *IsZero* and *IsOne* classes:

```
class Additive  b ⇒ IsZero b where isZero :: b → Bool
class Semiring b ⇒ IsOne  b where isOne  :: b → Bool
```

As with star semirings (Lemma 3), recursive affine equations in semimodules *over* star semirings also have solutions [3].

Lemma 4. In a left semimodule over a star semiring, the affine equation $p = b + m \cdot p$ has solution $p = m^* \cdot b$

2.6 Function-like Types and Singletons

Most of the representations used in this paper behave like functions, and it will be useful to use a standard vocabulary. An “indexable” type x with domain a and codomain b represents $a \to b$: Sometimes we will need to restrict a or b.

$$
\begin{array}{l}
\textbf{class } \mathit{Indexable}\ a\ b\ x \mid x \to a\ b \textbf{ where} \\
\quad \textbf{infixl } 9\ ! \\
\quad (!) :: x \to a \to b
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Indexable}\ a\ b\ (a \to b) \textbf{ where} \\
\quad f\ !\ k = f\ k
\end{array}
$$

Sections 2.1 through 2.5 provides a fair amount of vocabulary for combining values. We’ll also want an operation that constructs a “vector” (e.g., language or function) with a single nonzero component:

$$
\begin{array}{l}
\textbf{class } \mathit{Indexable}\ a\ b\ x \Rightarrow \mathit{HasSingle}\ a\ b\ x \textbf{ where} \\
\quad \textbf{infixr } 2 \mapsto \\
\quad (\mapsto) :: a \to b \to x
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } (\mathit{Eq}\ a, \mathit{Additive}\ b) \Rightarrow \mathit{HasSingle}\ a\ b\ (a \to b) \textbf{ where} \\
\quad a \mapsto b = \lambda\, a' \to \textbf{if } a = a' \textbf{ then } b \textbf{ else } 0
\end{array}
$$

Two specializations of $a \mapsto b$ will come in handy: one for $a = \varepsilon$, and the other for $b = 1$.

$$
\begin{array}{l}
\mathit{single} :: (\mathit{HasSingle}\ a\ b\ x, \mathit{Semiring}\ b) \Rightarrow a \to x \\
\mathit{single}\ a = a \mapsto 1
\end{array}
$$

$$
\begin{array}{l}
\mathit{value} :: (\mathit{HasSingle}\ a\ b\ x, \mathit{Monoid}\ a) \Rightarrow b \to x \\
\mathit{value}\ b = \varepsilon \mapsto b
\end{array}
$$

In particular, $\varepsilon \mapsto 1 = \mathit{single}\ \varepsilon = \mathit{value}\ 1$.

The $(\mapsto)$ operation gives a way to decompose arbitrary functions:

Lemma 5 (Proved in [3, Appendix A])**.** For all $f :: a \to b$ where b is an additive monoid,

$$f = \sum_a a \mapsto f\ a$$

For the uses in this paper, f is often “sparse”, i.e., nonzero on a relatively small (e.g., finite or at least countable) subset of its domain.

Singletons also curry handily and provide another useful homomorphism:

Lemma 6 (Proved in [3, Appendix A])**.**

$$(a \mapsto b \mapsto c) = \mathit{curry}\ ((a, b) \mapsto c)$$

Lemma 7. For $(\to)\ a$, partial applications $(a\ \mapsto)$ are left semi-module (and hence additive) homomorphisms. Moreover, $\mathit{single} = (\varepsilon\ \mapsto)$ is a semiring homomorphism.

Proof. Straightforward from the definition of $(\mapsto)$. □

3 CALCULATING INSTANCES FROM HOMOMORPHISMS

So far, we've started with instance definitions and then noted and proved homomorphisms where they arise. We can instead invert the process, taking homomorphisms as specifications and *calculating* instance definitions that satisfy them. This process of calculating instances from homomorphisms is the central guiding principle of this paper, so let's see how it works.

Consider a type "$\mathcal{P}\ a$" of mathematical *sets* of values of some type a. Are there useful instances of the abstractions from Section 2 for sets? Rather than guessing at such instances and then trying to prove the required laws, let's consider how sets are related to a type for which we already know instances, namely functions.

Sets are closely related to functions-to-booleans ("predicates"):

$$\begin{array}{ll} \mathit{pred} :: \mathcal{P}\ a \to (a \to \mathit{Bool}) & \mathit{pred}^{-1} :: (a \to \mathit{Bool}) \to \mathcal{P}\ a \\ \mathit{pred}\ \mathit{as} = \lambda\, a \to a \in \mathit{as} & \mathit{pred}^{-1}\ f = \{\, a \mid f\ a \,\} \end{array}$$

This pair of functions forms an isomorphism, i.e., $\mathit{pred}^{-1} \circ \mathit{pred} = \mathit{id}$ and $\mathit{pred} \circ \mathit{pred}^{-1} = \mathit{id}$, as can be checked by inlining definitions and simplifying. Moreover, for sets p and q, $p = q \Longleftrightarrow \mathit{pred}\ p = \mathit{pred}\ q$, by the *extensionality* axiom of sets and of functions. Now let's also require that *pred* be an *additive monoid homomorphism*. The required homomorphism properties:

$$\begin{array}{l} \mathit{pred}\ 0 = 0 \\ \mathit{pred}\ (p + q) = \mathit{pred}\ p + \mathit{pred}\ q \end{array}$$

We already know definitions of *pred* as well as the function versions of 0 and (+) (on the RHS) but not yet the set versions of 0 and (+) (on the LHS). We thus have two algebra problems in two unknowns. Since only one unknown appears in each homomorphism equation, we can solve them independently. The $\mathit{pred}/\mathit{pred}^{-1}$ isomorphism makes it easy to solve these equations, and removes all semantic choice, allowing only varying implementations of the same meaning.

$$\begin{array}{lll} & \mathit{pred}\ 0 = 0 & \\ \Longleftrightarrow & \mathit{pred}^{-1}\ (\mathit{pred}\ 0) = \mathit{pred}^{-1}\ 0 & \text{-- } \mathit{pred}^{-1} \text{ injectivity} \\ \Longleftrightarrow & 0 = \mathit{pred}^{-1}\ 0 & \text{-- } \mathit{pred}^{-1} \circ \mathit{pred} = \mathit{id} \end{array}$$

$$\begin{array}{lll} & \mathit{pred}\ (p + q) = \mathit{pred}\ p + \mathit{pred}\ q & \\ \Longleftrightarrow & \mathit{pred}^{-1}\ (\mathit{pred}\ (p + q)) = \mathit{pred}^{-1}\ (\mathit{pred}\ p + \mathit{pred}\ q) & \text{-- } \mathit{pred}^{-1} \text{ injectivity} \\ \Longleftrightarrow & p + q = \mathit{pred}^{-1}\ (\mathit{pred}\ p + \mathit{pred}\ q) & \text{-- } \mathit{pred}^{-1} \circ \mathit{pred} = \mathit{id} \end{array}$$

We thus have sufficient (and in this case semantically necessary) definitions for 0 and (+) on sets. Now let's simplify to get more direct definitions:

$$\begin{array}{ll} \quad \mathit{pred}^{-1}\ 0 & \\ = \mathit{pred}^{-1}\ (\lambda\, a \to 0) & \text{-- 0 on functions} \\ = \mathit{pred}^{-1}\ (\lambda\, a \to \mathit{False}) & \text{-- 0 on } \mathit{Bool} \\ = \{\, a \mid \mathit{False} \,\} & \text{-- } \mathit{pred}^{-1} \text{ definition} \\ = \emptyset & \end{array}$$

$$\begin{array}{ll} \quad \mathit{pred}^{-1}\ (\mathit{pred}\ p + \mathit{pred}\ q) & \\ = \mathit{pred}^{-1}\ ((\lambda\, a \to a \in p) + (\lambda\, a \to a \in q)) & \text{-- } \mathit{pred} \text{ definition (twice)} \\ = \mathit{pred}^{-1}\ (\lambda\, a \to (a \in p) + (a \in q)) & \text{-- (+) on functions} \\ = \mathit{pred}^{-1}\ (\lambda\, a \to a \in p \vee a \in q) & \text{-- (+) on } \mathit{Bool} \end{array}$$

$$
\begin{aligned}
&= \{\, a \mid a \in p \vee a \in q \,\} && \text{-- } \mathit{pred}^{-1} \text{ definition} \\
&= p \cup q && \text{-- } (\cup) \text{ definition}
\end{aligned}
$$

Without applying any real creativity, we have discovered the desired *Semiring* instance for sets:

```
instance Additive (𝒫 a) where
  0   = ∅
  (+) = (∪)
```

Next consider a *LeftSemimodule* instance for sets. We might be tempted to define $s \cdot p$ to multiply s by each value in p, i.e.,

```
instance LeftSemimodule s (𝒫 s) where s ·̂ p = { s ∗ x | x ∈ p }   -- wrong
```

This definition, however, would violate the semimodule law that $0 \cdot p = 0$, since $0 \cdot p$ would be $\{\, 0 \,\}$, but 0 for sets is $\emptyset$. Both semimodule distributive laws fail as well. There is an alternative choice, necessitated by requiring that pred^{-1} be a left *Bool*-semimodule homomorphism. The choice of *Bool* is inevitable from the type of pred^{-1} and the fact that $a \to b$ is a b-semimodule for all semirings b, so $a \to \mathit{Bool}$ is a *Bool*-semimodule. The necessary homomorphism property:

$$\mathit{pred}\ (s \cdot p) = s \cdot \mathit{pred}\ p$$

Equivalently [3],

```
instance LeftSemimodule Bool (𝒫 a) where
  s ·̂ p = if s then p else 0
```

While perhaps obscure at first, this alternative will prove useful later on.

Note that the left s-semimodule laws specialized to $s = \mathit{Bool}$ require *True* (1) to preserve and *False* (0) to annihilate the second $(\cdot)$ argument. *Every* left *Bool*-semimodule instance must therefore agree with this definition.

4 LANGUAGES AND THE MONOID SEMIRING

A *language* is a set of strings over some alphabet, so the *Additive* and *LeftSemimodule* instances for sets given above apply directly. Conspicuously missing, however, are the usual notions of language concatenation and closure (Kleene star), which are defined as follows for languages U and V:

$$U\ V = \{\, u \diamond v \mid u \in U \wedge v \in V \,\}$$

$$U^* = \bigcup_i U^i \quad \text{-- where } U^0 = 1 \text{, and } U^{n+1} = U\,U^n.$$

Intriguingly, this U^* definition would satisfy the *StarSemiring* laws if $(*)$ were language concatenation. A bit of reasoning shows that all of the semiring laws would hold as well:

- Concatenation is associative and has as identity the language $\{\, \varepsilon \,\}$.
- Concatenation distributes over union, both from the left and from the right.
- The 0 (empty) language annihilates (yields 0) under concatenation, both from the left and from the right.

All we needed from strings is that they form a monoid, so we may as well generalize:

```
instance Monoid a ⇒ Semiring (P a) where
  1 = { ε }   -- = ε ↦ 1 = single ε = value 1 (Section 2.6)
  p ∗ q = { u ⋄ v | u ∈ p ∧ v ∈ q }

instance StarSemiring (𝒫 a)   -- use default ·* definition (Section 2.4).
```

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Semiring}\ b, \mathit{Monoid}\ a) \Rightarrow \mathit{Semiring}\ (a \rightarrow b)\ \textbf{where}\\
&\quad 1 = \mathit{single}\ \varepsilon\\
&\quad f * g = \sum_{u,v} u \diamond v \mapsto f\ u * g\ v\\
&\qquad\quad = \lambda\ w \rightarrow \sum_{\substack{u,v\\ u\diamond v = w}} f\ u * g\ v\\[1ex]
&\textbf{instance}\ (\mathit{Semiring}\ b, \mathit{Monoid}\ a) \Rightarrow \mathit{StarSemiring}\ (a \rightarrow b) \quad \text{-- default } \cdot^*
\end{aligned}
$$

Fig. 1. The monoid semiring

These new instances indeed satisfy the laws for additive monoids, semimodules, semirings, and star semirings. They seem to spring from nothing, however, which is disappointing compared with the way the *Additive* and *LeftSemimodule* instances for sets follow inevitably from the requirement that *pred* be a homomorphism for those classes (Section 3). Let's not give up yet, however. Perhaps there's a *Semiring* instance for $a \rightarrow b$ that specializes with $b = \mathit{Bool}$ to bear the same relationship to $\mathcal{P}\ a$ that the *Additive* and *LeftSemimodule* instances bear. The least imaginative thing we can try is to require that *pred* be a *semiring* homomorphism. If we apply the same sort of reasoning as in Section 3 and then generalize from *Bool* to an arbitrary semiring, we get the definitions in Figure 1. With this instance, $a \rightarrow b$ type is known as the *monoid semiring*, and its $(*)$ operation as *convolution* [17, 59].

Theorem 8 (Proved in [3, Appendix A])**.** Given the instance definitions in Figure 1, *pred* is a star semiring homomorphism.

For some monoids, we can also express the product operation in a more clearly computable form via *splittings*:

$$f * g = \lambda\ w \rightarrow \sum_{(u,v) \in \mathit{splits}\ w} f\ u * g\ v$$

where *splits w* yields all pairs (u, v) such that $u \diamond v = w$:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monoid}\ t \Rightarrow \mathit{Splittable}\ t\ \textbf{where}\\
&\quad \mathit{splits} :: t \rightarrow [(t,t)] \quad \text{-- multi-valued inverse of } (\diamond)
\end{aligned}
$$

Examples of splittable monoids include natural numbers and lists [3].

While simple, general, and (assuming *Splittable* domain) computable, the definitions of $(+)$ and $(*)$ above for the monoid semiring make for quite inefficient implementations, primarily due to naive backtracking. As a simple example, consider the language *single* `"pickles"` + *single* `"pickled"`, and suppose that we want to test the word "pickling" for membership. The $(+)$ definition from Section 2.2 will first try "pickles", fail near the end, and then backtrack all the way to the beginning to try "pickled". The second attempt redundantly discovers that the prefix "pickl" is also a prefix of the candidate word and that "pickle" is not. Next consider the language *single* `"ca"` $*$ *single* `"ts"` $*$ *single* `"up"`, and suppose we want to test "catsup" for membership. The $(*)$ implementation above will try all possible three-way splittings of the test string.

5 FINITE MAPS

One representation of *partial* functions is the type of finite maps, *Map a b* from keys of type *a* to values of type *b*, represented is a key-ordered balanced tree [2, 43, 54]. To model *total* functions

```
instance (Ord a, Additive b) ⇒ Indexable a b (Map a b) where
  m ! a = M.findWithDefault 0 a m

instance (Ord a, Additive b) ⇒ HasSingle a b (Map a b) where
  (↦) = M.singleton

instance (Ord a, Additive b) ⇒ Additive (Map a b) where
  0 = M.empty
  (+) = M.unionWith (+)

instance (Ord a, Additive b) ⇒ IsZero (Map a b) where isZero = M.null

instance Semiring b ⇒ LeftSemimodule b (Map a b) where
  (·) b = fmap (b ∗)

instance (Ord a, Monoid a, Semiring b) ⇒ Semiring (Map a b) where
  1 = ε ↦ 1
  p ∗ q = sum [u ⋄ v ↦ p ! u ∗ q ! v | u ← M.keys p, v ← M.keys q]
```

Fig. 2. Finite maps

instead, we can treat unassigned keys as denoting zero. Conversely, merging two finite maps can yield a key collision, which can be resolved by addition. Both interpretations require *b* to be an additive monoid. Given the definitions in Figure 2, (!) is a homomorphism with respect to each instantiated class. (The "*M*." module qualifier indicates names coming from the finite map library [36].) The finiteness of finite maps prevents giving a useful *StarSemiring* instance.

6 DECOMPOSING FUNCTIONS FROM LISTS

For functions from *lists* specifically, we can decompose in a way that lays the groundwork for more efficient implementations than the ones in previous sections.

Lemma 9 (Proved in [3, Appendix A])**.** Any $f :: [c] \to b$ can be decomposed as follows:

$$f = at_\epsilon\ f \triangleleft \mathcal{D}\ f$$

Moreover, for all *b* and *h*,

$$\begin{aligned} at_\epsilon\ (b \triangleleft h) &= b \\ \mathcal{D}\ \ (b \triangleleft h) &= h \end{aligned}$$

where

```
at_ε :: ([c] → b) → b
at_ε f = f ε

𝒟 :: ([c] → b) → c → ([c] → b)
𝒟 f = λ c cs → f (c : cs)

infix 1 ◃
(◃) :: b → (c → ([c] → b)) → ([c] → b)
b ◃ h = λ case { [ ] → b ; c : cs → h c cs }
```

Considering the isomorphism $\mathcal{P}\ [c] \simeq [c] \to \mathit{Bool}$, this decomposition generalizes the δ and $\mathcal{D}$ operations used by Brzozowski [8] mapping languages to languages (as sets of strings), the latter of which he referred to as the "derivative".[1] Brzozowski used differentiation with respect to single symbols to implement a more general form of language differentiation with respect to a *string* of symbols, where the *derivative* $\mathcal{D}^*\ u\ p$ of a language p with respect to a prefix string u is the set of u-suffixes of strings in p, i.e.,

$$\mathcal{D}^*\ p\ u = \{\, v \mid u \diamond v \in p \,\}$$

so that

$$u \in p \iff \varepsilon \in \mathcal{D}^*\ p\ u$$

Further, he noted that

$$\begin{aligned} \mathcal{D}^*\ p\ \varepsilon &= p \\ \mathcal{D}^*\ p\ (u \diamond v) &= \mathcal{D}^*\ (\mathcal{D}^*\ p\ u)\ v \end{aligned}$$

Thanks to this decomposition property and the fact that $\mathcal{D}\ p\ c = \mathcal{D}^*\ p\ [c]$, one can successively differentiate with respect to single symbols.

Generalizing from sets to functions,

$$\mathcal{D}^*\ f\ u = \lambda\, v \to f\ (u \diamond v)$$

so that

$$\begin{aligned} f &= \lambda\, u \to \mathcal{D}^*\ f\ u\ \varepsilon \\ &= \lambda\, u \to \mathit{at}_\epsilon\ (\mathcal{D}^*\ f\ u) \\ &= \mathit{at}_\epsilon \circ \mathcal{D}^*\ f \\ &= \mathit{at}_\epsilon \circ \mathit{foldl}\ \mathcal{D}\ f \end{aligned}$$

where *foldl* is the usual left fold on lists:

$$\begin{aligned} &\mathit{foldl} :: (c \to b \to b) \to b \to [c] \to b \\ &\mathit{foldl}\ h\ e\ [\,] = e \\ &\mathit{foldl}\ h\ e\ (c : cs) = \mathit{foldl}\ h\ (h\ e\ c)\ cs \end{aligned}$$

Intriguingly, at_ϵ and $\mathcal{D}^*$ correspond to *coreturn* and *cojoin* for the function-from-monoid comonad, also called the "exponent comonad" [56].

Understanding how at_ϵ and $\mathcal{D}$ relate to the semiring vocabulary will help us develop efficient implementations in later sections.

Lemma 10 (Proved in [3, Appendix A])**.** The at_ϵ function is a star semiring and left semimodule homomorphism, i.e.,

$$\begin{aligned} \mathit{at}_\epsilon\ 0 &= 0 \\ \mathit{at}_\epsilon\ 1 &= 1 \\ \mathit{at}_\epsilon\ (p + q) &= \mathit{at}_\epsilon\ p + \mathit{at}_\epsilon\ q \\ \mathit{at}_\epsilon\ (p * q) &= \mathit{at}_\epsilon\ p * \mathit{at}_\epsilon\ q \\ \mathit{at}_\epsilon\ (p^*) &= (\mathit{at}_\epsilon\ p)^* \end{aligned}$$

[1] Brzozowski wrote "$\mathcal{D}^*_c\ p$" instead of "$\mathcal{D}\ p\ c$", but the latter will prove more convenient below.

Moreover,[2]

$$at_\epsilon\ (s \cdot p) = s * at_\epsilon\ p$$

$$\begin{aligned} at_\epsilon\ (\ [\ ]\ \ \mapsto b) &= b \\ at_\epsilon\ (c : cs \mapsto b) &= 0 \end{aligned}$$

Lemma 11 (Proved in [3, Appendix A], generalizing Lemma 3.1 of Brzozowski [8])**.** Differentiation has the following properties:

$$\begin{aligned} \mathcal{D}\ 0\ c &= 0 \\ \mathcal{D}\ 1\ c &= 0 \\ \mathcal{D}\ (p + q)\ c &= \mathcal{D}\ p\ c + \mathcal{D}\ q\ c \\ \mathcal{D}\ (p * q)\ c &= at_\epsilon\ p \cdot \mathcal{D}\ q\ c + \mathcal{D}\ p\ c * q \\ \mathcal{D}\ (p^*)\ c &= (at_\epsilon\ p)^* \cdot \mathcal{D}\ p\ c * p^* \\ \mathcal{D}\ (s \cdot p)\ c &= s \cdot \mathcal{D}\ p\ c \end{aligned}$$

$$\begin{aligned} \mathcal{D}\ (\ \ [\ ]\ \ \ \mapsto b) &= \lambda\, c \rightarrow 0 \\ \mathcal{D}\ (c' : cs' \mapsto b) &= c' \mapsto cs' \mapsto b \end{aligned}$$

Although $\mathcal{D}\ p$ is defined as a *function* from leading symbols, it could instead be another representation with function-like semantics, such as as $h\ b$ for an appropriate functor h. To relate h to the choice of alphabet c, introduce a type family:

type family $Key\ (h :: Type \rightarrow Type) :: Type$

type instance $Key\ ((\rightarrow)\ \ a) = a$
type instance $Key\ (Map\ a) = a$

Generalizing in this way (with functions as a special case) enables convenient memoization, which has been found to be quite useful in practice for derivative-based parsing [42]. A few generalizations to the equations in Lemma 11 suffice to generalize from $c \rightarrow ([c] \rightarrow b)$ to $h\ ([c] \rightarrow b)$ [3, Appendix A]. We must assume that $Key\ h = c$ and that h is an "additive functor", i.e., $\forall b.\ Additive\ b \Rightarrow Additive\ (h\ b)$ with (!) for h being an additive monoid homomorphism.

Theorem 12 (Proved in [3, Appendix A])**.** The following properties hold (in the generalized setting of a functor h with $Key\ h = c$):

$$\begin{aligned} &0 = 0 \blacktriangleleft 0 \\ &1 = 1 \blacktriangleleft 0 \\ &(a \blacktriangleleft dp) + (b \blacktriangleleft dq) = a + b \blacktriangleleft dp + dq \\ &(a \blacktriangleleft dp) * q = a \cdot q + (0 \blacktriangleleft fmap\ (*\ q)\ dp) \\ &(a \blacktriangleleft dp)^* = q\ \textbf{where}\ q = a^* \cdot (1 \blacktriangleleft fmap\ (*\ q)\ dp) \end{aligned}$$

[2] Mathematically, the $(\cdot)$ equation says that at_ϵ is a left b-semiring homomorphism as well, since every semiring is a (left and right) semimodule over itself. Likewise, the $(\mapsto)$ equation might be written as "$null\ w \mapsto b$" or even "$at_\epsilon\ w \mapsto b$". Unfortunately, these prettier formulations would lead to ambiguity during Haskell type inference.

$$s \cdot (a \triangleleft dp) = s * a \triangleleft \mathit{fmap}\ (s\ \cdot)\ dp$$

$$w \mapsto b = \mathit{foldr}\ (\lambda\ c\ t \to 0 \triangleleft c \mapsto t)\ (b \triangleleft 0)\ w$$

7 REGULAR EXPRESSIONS

Lemmas 10 and 11 generalize and were inspired by a technique of Brzozowski [8] for recognizing regular languages. Figure 3 generalizes regular expressions in the same way that $a \to b$ generalizes $\mathcal{P}\ a$, to yield a value of type b (a star semiring). The constructor *Value* b generalizes 0 and 1 to yield a semiring value.

Theorem 13. Given the definitions in Figure 3, (!) is a homomorphism with respect to each instantiated class.

The implementation in Figure 3 generalizes the regular expression matching algorithm of Brzozowski [8], adding customizable memoization, depending on choice of the indexable functor h. Note that the definition of $e\ !\ w$ is exactly $at_\epsilon\ (\mathcal{D}^*\ e\ w)$ generalized to indexable h, performing syntactic differentiation with respect to successive characters in w and applying at_ϵ to the final resulting regular expression.

For efficiency, and sometimes even termination (with recursively defined languages), we will need to add some optimizations to the *Additive* and *Semiring* instances for *RegExp* in Figure 3:

```
p + q | isZero p  = q            p * q | isZero p  = 0
      | isZero q  = p                  | isOne p   = q
      | otherwise = p :+ q             | otherwise = p :* q
```

For $p * q$, we might also check whether q is 0 or 1, but doing so itself leads to non-termination in right-recursive grammars.

As an alternative to repeated syntactic differentiation, we can reinterpret the original (syntactic) regular expression in another semiring as follows:

```
regexp :: (StarSemiring x, HasSingle [Key h] b x, Semiring b) ⇒ RegExp h b → x
regexp (Char c)  = single [c]
regexp (Value b) = value b
regexp (u :+ v)  = regexp u + regexp v
regexp (u :* v)  = regexp u * regexp v
regexp (Star u)  = (regexp u)*
```

Next, we will see a choice of f that eliminates the syntactic overhead.

8 TRIES

Section 4 provided an implementation of language recognition and its generalization to the monoid semiring ($a \to b$ for monoid a and semiring b), packaged as instances of a few common algebraic abstractions (*Additive*, *Semiring* etc). While simple and correct, these implementations are quite inefficient, primarily due to naive backtracking and redundant comparison. Section 6 explored the nature of functions on lists, identifying a decomposition principle and its relationship to the vocabulary of semirings and related algebraic abstractions. Applying this principle to a generalized form of regular expressions led to Brzozowski's algorithm, generalized from sets to functions in Section 7, providing an alternative to naive backtracking but still involving repeated syntactic manipulation as each candidate string is matched. Nevertheless, with some syntactic optimizations and memoization, recognition speed with this technique can be fairly good [1, 42].

```
data RegExp h b = Char (Key h)
                | Value b
                | RegExp h b :+ RegExp h b
                | RegExp h b :* RegExp h b
                | Star (RegExp h b)
  deriving Functor

instance Additive b ⇒ Additive (RegExp h b) where
  0 = Value 0
  (+) = (:+)

instance Semiring b ⇒ Semiring (RegExp h b) where
  (·) b = fmap (b ∗)

instance Semiring b ⇒ Semiring (RegExp h b) where
  1 = Value 1
  (∗) = (:*)

instance Semiring b ⇒ StarSemiring (RegExp h b) where
  e* = Star e

type FR h b = (HasSingle (Key h) (RegExp h b) (h (RegExp h b))
              , Additive (h (RegExp h b)), Functor h, IsZero b, IsOne b)

instance (FR h b, StarSemiring b, c ∼ Key h, Eq c) ⇒ Indexable [c] b (RegExp h b) where
  e ! w = at_ϵ (foldl ((!) ∘ 𝒟) e w)

instance (FR h b, StarSemiring b, c ∼ Key h, Eq c) ⇒ HasSingle [c] b (RegExp h b) where
  w ↦ b = b · product (map Char w)

at_ϵ :: StarSemiring b ⇒ RegExp h b → b
at_ϵ (Char _)  = 0
at_ϵ (Value b) = b
at_ϵ (p :+ q)  = at_ϵ p + at_ϵ q
at_ϵ (p :* q)  = at_ϵ p ∗ at_ϵ q
at_ϵ (Star p)  = (at_ϵ p)*

𝒟 :: (FR h b, StarSemiring b) ⇒ RegExp h b → h (RegExp h b)
𝒟 (Char c)  = single c
𝒟 (Value _) = 0
𝒟 (p :+ q)  = 𝒟 p + 𝒟 q
𝒟 (p :* q)  = fmap (at_ϵ p ·) (𝒟 q) + fmap (∗ q) (𝒟 p)
𝒟 (Star p)  = fmap (λ d → (at_ϵ p)* · d ∗ Star p) (𝒟 p)
```

Fig. 3. Semiring-generalized regular expressions denoting $[c] \to b$

As an alternative to regular expression differentiation, note that the problem of redundant comparison is solved elegantly by the classic trie ("prefix tree") data structure introduced by Thue in 1912 [32, Section 6.3]. This data structure was later generalized to arbitrary (regular) algebraic data types [10] and then from sets to functions [22]. Restricting our attention to functions of *lists* ("strings" over some alphabet), we can formulate a simple trie data type along the lines of ($\triangleleft$) from Section 6, with an entry for ε and a sub-trie for each possible character:

$$\mathbf{data}\ \mathit{LTrie}\ c\ b = b \mathrel{:\triangleleft} c \to \mathit{LTrie}\ c\ b \quad \text{-- first guess}$$

While this definition would work, we can get much better efficiency if we memoize the functions of c, e.g., as a generalized trie or a finite map. Rather than commit to a particular representation for subtrie collections, let's replace the type parameter c with a functor h whose associated key type is c. The functor-parametrized list trie is also known as the "cofree comonad" [23, 31, 46, 55–57].

$$\mathbf{data}\ \mathit{Cofree}\ h\ b = b \mathrel{:\triangleleft} h\ (\mathit{Cofree}\ h\ b)$$

The similarity between $\mathit{Cofree}\ h\ b$ and the function decomposition from Section 6 (motivating the constructor name "$:\triangleleft$") makes for easy instance calculation. As with $\mathcal{P}\ a$ and $\mathit{Map}\ a\ b$, we can define a trie counterpart to the free monoid semiring $[c] \to b$.

Theorem 14 (Proved in [3, Appendix A]). Given the definitions in Figure 4, ($!$) is a homomorphism with respect to each instantiated class.

Although the ($\triangleleft$) decomposition in Section 6 was inspired by wanting to understand the essence of regular expression derivatives, the application to tries is in retrospect more straightforward, since the representation directly mirrors the decomposition. Applying the ($\triangleleft$) decomposition to tries also appears to be more streamlined than the application to regular expressions. During matching, the next character in the candidate string is used to directly index to the relevant derivative (sub-trie), efficiently bypassing all other paths. As one might hope, ($!$) is a comonad homomorphism from $\mathit{Cofree}\ h$ to $(\to)\ (\mathit{Key}\ h)$ [3, Appendix A].

9 PERFORMANCE

While the implementation has had no performance tuning and only rudimentary benchmarking, we can at least get a sanity check on performance and functionality. Figure 5 shows the source code for a collection of examples, all polymorphic in the choice of semiring. The *atoz* language contains single letters from 'a' to 'z'. The examples *anbn* and *dyck* are two classic, non-regular, context-free languages: $\{\, a^n\ b^n \mid n \in \mathbb{N} \,\}$ and the Dyck language of balanced brackets.

Figure 6 gives some execution times for these examples measured with the *criterion* library [44], compiled with GHC 8.6.3, and running on a late 2013 MacBook Pro. (Note milliseconds vs microseconds—"ms" vs "μs".) Each example is interpreted in four semirings: $\mathit{RegExp}\ ((\to)\ \mathit{Char})\ \mathbb{N}$, $\mathit{RegExp}\ (\mathit{Map}\ \mathit{Char})\ \mathbb{N}$, $\mathit{Cofree}\ ((\to)\ \mathit{Char}\ \mathbb{N})$, and $\mathit{Cofree}\ (\mathit{Map}\ \mathit{Char})\ \mathbb{N}$. Each interpretation of each language is given a matching input string of length 100; and matches are counted, thanks to use of the $\mathbb{N}$ semiring. (The $a^* * a^*$ example matches in 101 ways, while the others match uniquely.) As the figure shows, memoization (via *Map*) is only moderately helpful (and occasionally harmful) for *RegExp*. *Cofree*, on the other hand, performs terribly without memoization and (in these examples) 2K to 230K times faster with memoization. Here, memoized *Cofree* performs between 8.5 and 485 times faster than memoized *RegExp* and between 11.5 and 1075 times faster than nonmemoized *RegExp*. The two recursively defined examples fail to terminate with *RegExp Map*, perhaps because the implementation (Section 7) lacks one more crucial tricks [42]. Other *RegExp* improvements [1, 42] might narrow the gap further, and careful study and optimization of the *Cofree* implementation (Figure 4) might widen it.

```
infix 1 :◁
data Cofree h b = b :◁ h (Cofree h b) deriving Functor

instance Indexable (Cofree h b) h ⇒ Indexable b (Cofree h) where
   type instance Key (Cofree h) = [Key h]
   (!) (b :◁ dp) = b ◁ (!) ∘ (!) dp   -- (b :◁ dp) ! w = case w of {[ ] → b; c : cs → dp ! c ! cs}

instance (Additive (h (Cofree h b)), Additive b) ⇒ Additive (Cofree h b) where
   0 = 0 :◁ 0
   (a :◁ dp) + (b :◁ dq) = a + b :◁ dp + dq

instance (Functor h, Semiring b) ⇒ LeftSemimodule b (Cofree h b) where
   (·) s = fmap (s ∗)

instance (Functor h, Additive (h (Cofree h b)), Semiring b, IsZero b) ⇒
            Semiring (Cofree h b) where
   1 = 1 :◁ 0
   (a :◁ dp) ∗ q = a · q + (0 :◁ fmap (∗ q) dp)

instance (Functor h, Additive (h (Cofree h b)), StarSemiring b, IsZero b) ⇒
            StarSemiring (Cofree h b) where
   (a :◁ dp)* = q where q = a* · (1 :◁ fmap (∗ q) dp)

instance (HasSingle (Cofree h b) h, Additive (h (Cofree h b)), Additive b) ⇒
            HasSingle b (Cofree h) where
   w ↦ b = foldr (λ c t → 0 :◁ c ↦ t) (b :◁ 0) w

instance (Additive (h (Cofree h b)), IsZero (h (Cofree h b)), IsZero b) ⇒
            IsZero (Cofree h b) where
   isZero (a :◁ dp) = isZero a ∧ isZero dp

instance (Functor h, Additive (h (Cofree h b)), IsZero b, IsZero (h (Cofree h b)), IsOne b) ⇒
            IsOne (Cofree h b) where
   isOne (a :◁ dp) = isOne a ∧ isZero dp
```

Fig. 4. List tries denoting $[c] \to b$

10 CONVOLUTION

Consider again the definition of multiplication in the monoid semiring, on $f, g :: a \to b$ from Figure 1.

$$f * g = \sum_{u,v} u \diamond v \mapsto f\ u * g\ v$$

As in Section 4, specializing the *codomain* to *Bool* and using the set/predicate isomorphism from Section 3, we can translate this definition from predicates to "languages" (sets of values in some monoid):

$$f * g = \{\, u \diamond v \mid u \in f \wedge v \in g \,\}$$

$$a = \mathit{single}\ \texttt{"a"}$$
$$b = \mathit{single}\ \texttt{"b"}$$

$$\mathit{atoz} = \mathit{sum}\ [\mathit{single}\ [c] \mid c \leftarrow [\texttt{'a'}\,..\,\texttt{'z'}]]$$

$$\mathit{fishy} = \mathit{atoz}^* * \mathit{single}\ \texttt{"fish"} * \mathit{atoz}^*$$

$$\mathit{anbn} = 1 + a * \mathit{anbn} * b$$

$$\mathit{dyck} = (\mathit{single}\ \texttt{"["} * \mathit{dyck} * \mathit{single}\ \texttt{"]"})^*$$

Fig. 5. Examples

Example	$\mathit{RegExp}_{\rightarrow}$	$\mathit{RegExp}_{\mathit{Map}}$	$\mathit{Cofree}_{\rightarrow}$	$\mathit{Cofree}_{\mathit{Map}}$
a^*	30.56 μs	22.45 μs	5.258 ms	2.624 μs
atoz^*	690.4 μs	690.9 μs	10.89 ms	3.574 μs
$a^* * a^*$	2.818 ms	1.274 ms	601.6 ms	2.619 μs
$a^* * b^*$	52.26 μs	36.59 μs	14.40 ms	2.789 μs
$a^* * b * a^*$	56.53 μs	49.21 μs	14.58 ms	2.798 μs
fishy	1.276 ms	2.528 ms	29.73 ms	4.233 μs
anbn	1.293 ms	∞	12.12 ms	2.770 μs
dyck	254.9 μs	∞	24.77 ms	3.062 μs

Fig. 6. Running times for examples in Figure 5

which is the definition of the concatenation of two languages from Section 4. Likewise, by specializing the *domain* of the functions to sequences (from general monoids), we got efficient matching of semiring-generalized "languages", as in Sections 6 and 8, which translated to regular expressions (Section 7), generalizing work of Brzozowski [8].

Let's now consider specializing the functions' domains to *integers* rather than sequences, recalling that integers (and numeric types in general) form a monoid under addition.

$$\begin{aligned}
f * g &= \sum_{u,v} u + v \mapsto f\ u * g\ v && \text{-- Figure 1 with } (\diamond) = (+)\\
&= \lambda\, w \rightarrow \sum_{\substack{u,v\\ u+v=w}} f\ u * g\ v && \text{-- equivalent definition}\\
&= \lambda\, w \rightarrow \sum_{\substack{u,v\\ v=w-u}} f\ u * g\ v && \text{-- solve } u + v = w \text{ for } v\\
&= \lambda\, w \rightarrow \sum_{u} f\ u * g\ (w - u) && \text{-- substitute } w - u \text{ for } v
\end{aligned}$$

This last form is the standard definition of one-dimensional, discrete *convolution* [53, Chapter 6].[3] Therefore, just as monoid semiring multiplication generalizes language concatenation (via the predicate/set isomorphism), it also generalizes the usual notion of discrete convolution. Moreover, if the domain is a continuous type such as $\mathbb{R}$ or $\mathbb{C}$, we can reinterpret summation as integration,

[3] Note that this reasoning applies to *any* group (monoid with inverses)

resulting in *continuous* convolution. Additionally, for multi-dimensional (discrete or continuous) convolution, we can simply use tuples of scalar indices for w and u, defining tuple addition and subtraction componentwise. Alternatively, curry, convolve, and uncurry, exploiting the fact that *curry* is a semiring homomorphism (Theorem 2).

What if we use functions from $\mathbb{N}$ rather than from $\mathbb{Z}$? Because $\mathbb{N} \simeq [()]$ (essentially, Peano numbers), we can directly use the definitions in Section 6 for domain $[c]$, specialized to $c = ()$. As a suitable indexable functor, we can simply use the identity functor:

```
newtype Identity b = Identity b deriving
   (Functor, Additive, IsZero, IsOne, LeftSemimodule s, Semiring)

instance Indexable () b (Identity b) where Identity a ! () = a
instance HasSingle () b (Identity b) where () ↦ b = Identity b
```

The type *Cofree Identity* is isomorphic to *streams* (infinite-only lists). Inlining and simplification during compilation might eliminate all of the run-time overhead of introducing the identity functor.

Just as *Cofree Identity* gives (necessarily infinite) streams, *Cofree Maybe* gives (possibly finite) *nonempty lists* [38, 56]. As with finite maps, we can interpret absence (*Nothing*) as 0 [3]. Alternatively, define instances directly for lists, specified by a denotation of $[b]$ as $\mathbb{N} \to b$. The instances resemble those in Figure 4, but have an extra case for the empty list and no *fmap*:

```
instance Additive b ⇒ Indexable ℕ b [b] where
   [] ! _ = 0
   (b : _ ) ! 0 = b
   (_ : bs) ! n = bs ! (n − 1)

instance Additive b ⇒ Additive [b] where
   0 = []
   [] + bs = bs
   as + [] = as
   (a : as) + (b : bs) = a + b : as + bs

instance (Semiring b, IsZero b, IsOne b) ⇒ Semiring [b] where
   1 = 1 : 0
   []       ∗ _ = []     -- 0 ∗ q = 0
   (a : dp) ∗ q = a · q + (0 : dp ∗ q)
```

This last definition is reminiscent of long multiplication, which is convolution in disguise.

11 BEYOND CONVOLUTION

Many uses of discrete convolution (including convolutional neural networks [33, Chapter 9]) involve functions having finite support, i.e., nonzero on only a finite subset of their domains. In many cases, these domain subsets may be defined by finite *intervals*. For instance, such a 2D operation would be given by intervals in each dimension, together specifying lower left and upper right corners of a 2D interval (rectangle) outside of which the functions are guaranteed to be zero. The two input intervals needn't have the same size, and the result's interval of support is typically larger than both inputs, with size equaling the sum of the sizes in each dimension (minus one for the discrete case). Since the result's support size is entirely predictable and based only on the arguments' sizes, it is appealing to track sizes statically via types. For instance, a 1D convolution might have the following type:

$$(*) :: Semiring\ s \Rightarrow Array_{m+1}\ s \to Array_{n+1}\ s \to Array_{m+n+1}\ s$$

Unfortunately, this signature is incompatible with semiring multiplication, in which arguments and result all have the same type.

From the perspective of functions, an array of size n is a memoized function from Fin_n, a type representing the finite set $\{ 0, \ldots, n - 1 \}$. We can still define convolution in the customary sense in terms of index addition:

$$f * g = \sum_{u,v} u + v \mapsto f\ u * g\ v$$

where now

$$(+) :: Fin_{m+1} \to Fin_{n+1} \to Fin_{m+n+1}$$

Indices can no longer form a monoid under addition, however, due to the nonuniformity of types.

The inability to support convolution on statically sized arrays (or other memoized forms of functions over finite domains) as semiring multiplication came from the expectation that indices/arguments combine via a monoid. Fortunately, this expectation can be dropped by generalizing from monoidal combination to an *arbitrary* binary operation $h :: a \to b \to c$. For now, let's call this more general operation "$lift_2\ h$".

$$\begin{aligned} &lift_2 :: Semiring\ s \Rightarrow (a \to b \to c) \to (a \to s) \to (b \to s) \to (c \to s) \\ &lift_2\ h\ f\ g = \lambda\, w \to \sum_{u,v} h\ u\ v \mapsto f\ u * g\ v \end{aligned}$$

We can similarly lift functions of *any* arity:

$$\begin{aligned} &lift_n :: Semiring\ s \Rightarrow (a_1 \to \ldots \to a_n \to b) \to (a_1 \to s) \to \ldots \to (a_n \to s) \to (b \to s) \\ &lift_n\ h\ f_1 \ldots f_n = \sum_{u_1,\ldots,u_n} h\ u_1\ \cdots\ u_n \mapsto f_1\ u_1 * \cdots * f_n\ u_n \end{aligned}$$

Here we are summing over the set-valued *preimage* of w under h. Now consider two specific cases of $lift_n$:

$$\begin{aligned} &lift_1 :: Semiring\ s \Rightarrow (a \to b) \to (a \to s) \to (b \to s) \\ &lift_1\ h\ f = \sum_{u} h\ u \mapsto f\ u \end{aligned}$$

$$\begin{aligned} &lift_0 :: Semiring\ s \Rightarrow b \to (b \to s) \\ &lift_0\ b = b \mapsto 1 \\ &\qquad\ \ = single\ b \end{aligned}$$

The signatures of $lift_2$, $lift_1$, and $lift_0$ *almost* generalize to those of $liftA_2$, *fmap*, and *pure* from the *Functor* and *Applicative* type classes [39, 61]. In type systems like Haskell's, however, $a \to s$ is the functor $(a \to)$ applied to s, while we would need it to be $(\to s)$ applied to a. To fix this problem, define a type wrapper that swaps domain and codomain type parameters:

$$\textbf{newtype}\ s \leftarrow a = F\ (a \to s)$$

The use of $s \leftarrow a$ as an alternative to $a \to s$ allows us to give instances for both and to stay within Haskell's type system (and ability to infer types via first-order unification).

With this change, we can replace the specialized $lift_n$ operations with standard ones. An enhanced version of the *Functor* and *Applicative* classes (similar to those by Kidney [28]) appear in Figure 7, along with instances for functions and finite maps. Other representations would need similar

```
class Functor f where
  type Ok f a :: Constraint
  type Ok f a = ()
  fmap :: (Ok f a, Ok f b) ⇒ (a → b) → f a → f b

class Functor f ⇒ Applicative f where
  pure :: Ok f a ⇒ a → f a
  liftA₂ :: (Ok f a, Ok f b, Ok f c) ⇒ (a → b → c) → f a → f b → f c

infixl 1 ≫=
class Applicative f ⇒ Monad f where
  (≫=) :: (Ok f a, Ok f b) ⇒ f a → (a → f b) → f b

instance Semiring b ⇒ Functor ((←) b) where
  type Ok ((←) b) a = Eq a
  fmap h (F f) = ∑_{u} h u ↦ f u

instance Semiring b ⇒ Applicative ((←) b) where
  pure a = single a
  liftA₂ h (F f) (F g) = ∑_{u,v} h u v ↦ f u ∗ g v

newtype Map′ b a = M (Map a b)

instance IsZero b ⇒ Functor (Map′ b) where
  type Ok (Map′ b) a = Ord a
  fmap h (M p) = ∑_{a∈M.keys p} h a ↦ p ! a

instance IsZero b ⇒ Applicative (Map′ b) where
  pure a = single a
  liftA₂ h (M p) (M q) = ∑_{a∈M.keys p, b∈M.keys q} h a b ↦ (p ! a) ∗ (q ! b)
```

Fig. 7. *Functor* and *Applicative* classes and some instances

reversal of type arguments. [4,5] Higher-arity liftings can be defined via these three. For $b \leftarrow a$, these definitions are not really executable code, since they involve potentially infinite summations, but they serve as specifications for other representations such as finite maps, regular expressions, and tries.

Theorem 15. For each instance defined in Figure 7, $1 = \mathit{pure}\ \varepsilon$, and $(\ast) = \mathit{liftA}_2\ (\diamond)$.

[4] The enhancement is the associated constraint [5] *Ok*, limiting the types that the class methods must support. The line "**type** *Ok f a* = ()" means that the constraint on *a* defaults to (), which holds vacuously for all *a*.

[5] Originally, *Applicative* had a (⊛) method from which one can easily define liftA_2. Since the base library version 4.10, liftA_2 was added as a method (along with a default definition of (⊛)) to allow for more efficient implementation [16, Section 3.2.2].

```
instance Semiring b ⇒ Semiring (a → b) where
  1 = pure 1          -- i.e., 1 = λ a → 1
  (∗) = liftA2 (∗)    -- i.e., f ∗ g = λ a → f a ∗ g a

newtype b ← a = F (a → b) deriving (Additive, HasSingle b, LeftSemimodule b, Indexable a b)

instance (Semiring b, Monoid a) ⇒ Semiring (b ← a) where
  1 = pure ε
  (∗) = liftA2 (⋄)
```

Fig. 8. The $a \rightarrow b$ and $b \leftarrow a$ semirings

Proof. Immediate from the instance definitions. □

Given the type distinction between $a \rightarrow b$ and $b \leftarrow a$, let's now reconsider the *Semiring* instance for functions in Figure 1. There is an alternative choice that is in some ways more compelling, as shown in Figure 8, along with a the old $a \rightarrow b$ instance reexpressed and reassigned to $b \leftarrow a$. Just as the *Additive* and *Semiring* instances for $Bool \leftarrow a$ give us four important languages operations (union, concatenation and their identities), now the *Semiring* $(a \rightarrow Bool)$ gives us two more: the *intersection* of languages and its identity (the set of all "strings"). These two semirings share several instances in common, expressed in Figure 8 via GHC-Haskell's `GeneralizedNewtypeDeriving` language extension (present since GHC 6.8.1 and later made safe by Breitner et al. [7]). All six of these operations are also useful in their generalized form (i.e., for $a \rightarrow b$ and $b \leftarrow a$ for semirings b). As with *Additive*, this *Semiring* $(a \rightarrow b)$ instance implies that curried functions (of any number and type of arguments and with semiring result type) are semirings, with *curry* and *uncurry* being semiring homomorphisms. (The proof is very similar to that of Theorem 1.)

The $a \rightarrow b$ and $b \leftarrow a$ semirings have another deep relationship:

Theorem 16. *The Fourier transform is a semiring and left semimodule homomorphism from $b \leftarrow a$ to $a \rightarrow b$.*

This theorem is more often expressed by saying that (a) the Fourier transform is linear (i.e., an additive-monoid and left-semimodule homomorphism), and (b) the Fourier transform of a convolution (i.e., $(\ast)$ on $b \leftarrow a$) of two functions is the pointwise product (i.e., $(\ast)$ on $a \rightarrow b$) of the Fourier transforms of the two functions. The latter property is known as "the convolution theorem" [6, Chapter 6].

12 THE FREE SEMIMODULE MONAD

Where there's an applicative, there's often a compatible monad. For $b \leftarrow a$, the monad is known as the "free semimodule monad" (or sometimes the "free *vector space* monad") [15, 30, 48]. The semimodule's dimension is the cardinality of a. Basis vectors have the form $single\ u = u \mapsto 1$ for $u :: a$ (mapping u to 1 and every other value to 0 as in Figure 1).

The monad instances for $(\leftarrow)\ b$ and $Map'\ b$ are defined as follows:[6]

```
instance Semiring s ⇒ Monad ((←) s) where
  (>>=) :: (s ← a) → (a → (s ← b))) → (s ← b)
```

[6] The *return* method does not appear here, since it is equivalent to *pure* from *Applicative*.

$$F\ f \gg\!\!= h = \sum_a f\ a \cdot h\ a$$

$$\textbf{instance}\ (\mathit{Semiring}\ b, \mathit{IsZero}\ b) \Rightarrow \mathit{Monad}\ (\mathit{Map}'\ b)\ \textbf{where}$$
$$\quad M\ m \gg\!\!= h = \sum_{a \in M.\mathit{keys}\ m} m\ !\ a \cdot h\ a$$

Theorem 17 (Proved in [3, Appendix A]). The definitions of *fmap* and liftA_2 on $(\leftarrow)\ b$ in Figure 7 satisfy the following standard equations for monads:

$$\mathit{fmap}\ h\ p = p \gg\!\!= \mathit{pure} \circ h$$

$$\begin{aligned}\mathit{liftA}_2\ h\ p\ q &= p \gg\!\!= \lambda\, u \to \mathit{fmap}\ (h\ u)\ q \\ &= p \gg\!\!= \lambda\, u \to q \gg\!\!= \lambda\, v \to \mathit{pure}\ (h\ u\ v)\end{aligned}$$

13 OTHER APPLICATIONS

13.1 Polynomials

As is well known, univariate polynomials form a semiring and can be multiplied by convolving their coefficients. Perhaps less known is that this trick extends naturally to power series and to multivariate polynomials.

Looking more closely, univariate polynomials (and even power series) can be represented by a collection of coefficients indexed by exponents, or conversely as a collection of exponents weighted by coefficients. For a polynomial in a variable x, an association $i \mapsto c$ of coefficient c with exponent i represents the monomial (polynomial term) $c * x^i$. One can use a variety of representations for these indexed collections. We'll consider efficient representations below, but let's begin as $b \leftarrow \mathbb{N}$ along with a denotation as a (polynomial) function of type $b \to b$:

$$\mathit{poly}_1 :: \mathit{Semiring}\ b \Rightarrow (b \leftarrow \mathbb{N}) \to (b \to b)$$
$$\mathit{poly}_1\ (F\ f) = \lambda\, x \to \sum_i f\ i * x^i$$

Polynomial multiplication via convolution follows from the following property:

Theorem 18 (Proved in [3, Appendix A]). The function poly_1 is a semiring homomorphism when multiplication on b commutes.

Pragmatically, Theorem 18 says that the $b \leftarrow \mathbb{N}$ semiring (in which $(*)$ is convolution) correctly implements arithmetic on univariate polynomials. More usefully, we can adopt other representations of $b \leftarrow \mathbb{N}$, such as $\mathit{Map}\ \mathbb{N}\ b$. For viewing results, wrap these representations in a new type, and provide a *Show* instance:

$$\textbf{newtype}\ \mathit{Poly}_1\ z = \mathit{Poly}_1\ z\ \textbf{deriving}\ (\mathit{Additive}, \mathit{Semiring}, \mathit{Indexable}\ n\ b, \mathit{HasSingle}\ n\ b)$$

$$\textbf{instance}\ (\ldots) \Rightarrow \mathit{Show}\ (\mathit{Poly}_1\ z)\ \textbf{where}\ \ldots$$

Try it out (with prompts indicated by "$\lambda\rangle$ "):

$$\lambda\rangle\ \textbf{let}\ p = \mathit{single}\ 1 + \mathit{value}\ 3 :: \mathit{Poly}_1\ (\mathit{Map}\ \mathbb{N}\ \mathbb{Z})$$
$$\lambda\rangle\ p$$
$$x + 3$$

$$\lambda\rangle\ p^3$$
$$x^3 + 9x^2 + 27x + 27$$

$\lambda\rangle\ p^7$
$2187 + 5103x + 5103x^2 + 2835x^3 + 945x^4 + 189x^5 + 21x^6 + x^7$

$\lambda\rangle\ \mathit{poly}_1\ (p^5)\ 17 = (\mathit{poly}_1\ p\ 17)^5$
True

We can also use [ ] in place of *Map* ℕ. The example above yields identical results. Since lists are potentially infinite (unlike finite maps), however, this simple change enables power series [3, 40, 41].

What about multivariate polynomials, i.e., polynomial functions over higher-dimensional domains? Generalize to n dimensions:

$$\begin{aligned}&\mathit{poly} :: (b \leftarrow \mathbb{N}^n) \to (b^n \to b)\\ &\mathit{poly}\ (F\ f)\ (x :: b^n) = \sum_{p::\mathbb{N}^n} f\ p * x\hat{}p\end{aligned}$$

$$\begin{aligned}&\textbf{infixr}\ 8\ \hat{}\\ &(\hat{}) :: b^n \to \mathbb{N}^n \to b\\ &x\hat{}p = \prod_{i<n} x_i^{p_i}\end{aligned}$$

For instance, for $n = 3$, $(x, y, z)\hat{}(i, j, k) = x^i * y^j * z^k$. Generalizing further, let n be any type, and interpret b^n and $\mathbb{N}^n$ as $n \to b$ and $n \to \mathbb{N}$:

$$\begin{aligned}&\mathit{poly} :: (b \leftarrow (n \to \mathbb{N})) \to ((n \to b) \to b)\\ &\mathit{poly}\ (F\ f)\ (x :: n \to b) = \sum_{p::n\to\mathbb{N}} f\ p * x\hat{}p\end{aligned}$$

$$\begin{aligned}&\textbf{infixr}\ 8\\ &(\hat{}) :: (n \to b) \to (n \to \mathbb{N}) \to b\\ &x\hat{}p = \prod_{i} (x\ i)^{(p\ i)}\end{aligned}$$

Theorem 19 (Proved in [3, Appendix A]). *The generalized poly function is a semiring homomorphism when multiplication on b commutes.*

Theorem 19 says that the $b \leftarrow (n \to \mathbb{N})$ semiring (in which $(*)$ is higher-dimensional convolution) correctly implements arithmetic on multivariate polynomials. We can instead use *Map* $(f\ \mathbb{N})\ b$ to denote $b \leftarrow (n \to \mathbb{N})$, where f is indexable with *Key* $f = n$. One convenient choice is to let $n = \mathit{String}$ for variable names, and $f = \mathit{Map\ String}$.[7] As with Poly_1, wrap this representation in a new type, and add a *Show* instance:

$$\begin{aligned}&\textbf{newtype}\ \mathit{Poly}_M\ b = \mathit{Poly}_M\ (\mathit{Map}\ (\mathit{Map\ String}\ \mathbb{N})\ b)\\ &\quad \textbf{deriving}\ (\mathit{Additive}, \mathit{Semiring}, \mathit{Indexable}\ n, \mathit{HasSingle}\ n, \mathit{Functor})\end{aligned}$$

$$\textbf{instance}\ (\ldots) \Rightarrow \mathit{Show}\ (\mathit{Poly}_M\ b)\ \textbf{where} \ldots$$

$$\begin{aligned}&\mathit{var} :: \mathit{Semiring}\ b \Rightarrow \mathit{String} \to \mathit{Poly}_M\ b\\ &\mathit{var} = \mathit{single} \circ \mathit{single}\end{aligned}$$

Try it out:

$\lambda\rangle\ \textbf{let}\ p = \mathit{var}\ \texttt{"x"} + \mathit{var}\ \texttt{"y"} + \mathit{var}\ \texttt{"z"} :: \mathit{Poly}_M\ \mathbb{Z}$
$\lambda\rangle\ p$
$x + y + z$

$\lambda\rangle\ p^2$
$x^2 + 2xy + 2xz + y^2 + 2yz + z^2$

$\lambda\rangle\ p^3$
$x^3 + 3x^2y + 3xy^2 + 6xyz + 3x^2z + 3xz^2 + y^3 + 3y^2z + 3yz^2 + z^3$

[7] Unfortunately, the *Monoid* instance for the standard *Map* type defines $m \diamond m'$ so that keys present in m' *replace* those in m. This behavior is problematic for our use (and many others), so we must use a *Map* variant that wraps the standard type, changing the *Monoid* instance so that $m \diamond m'$ *combines* values (via $(\diamond)$) associated with the same keys in m and m'.

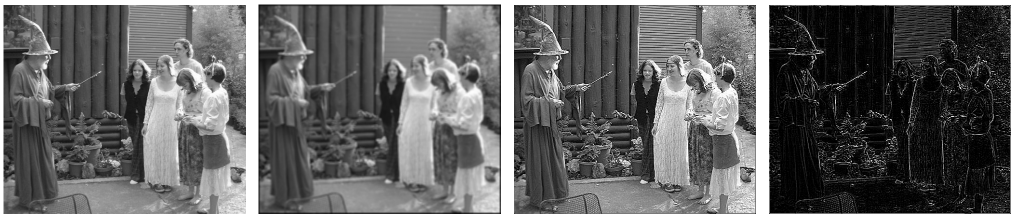

Fig. 9. Image convolution

13.2 Image Convolution

Figure 9 shows examples of image convolution with some commonly used kernels [47, 62]. The source image (left) and convolution kernels are all represented as lists of lists of floating point grayscale values. Because (semiring) multiplication on $[\,b\,]$ is defined via multiplication on b, one can nest representations arbitrarily. Other more efficient representations can work similarly.

14 RELATED WORK

This paper began with a desire to understand regular expression matching via "derivatives" by Brzozowski [8] more fundamentally and generally. Brzozowski's method spurred much follow-up investigation in recent years. Owens et al. [45] dusted off regular expression derivatives after years of neglect with a new exposition and experience report. Might and Darais [42] considerably extended expressiveness to context-free grammars as well as addressing some efficiency issues, including memoization, with further performance analysis given later [1]. Fischer et al. [13] also extended regular language membership from boolean to "weighted" by an arbitrary semiring, relating them to weighted finite automata. Piponi and Yorgey [49] investigated regular expressions and their relationship to the semiring of polynomial functors, as well as data type derivatives and dissections. Radanne and Thiemann [51] explored regular expressions extended to include intersection and complement (as did Brzozowski) with an emphasis on testing.

McIlroy [40, 41] formulated power series as a small and beautiful collection of operations on infinite coefficient streams, including not only the arithmetic operations, but also inversion and composition, as well as differentiation and integration. He also defined transcendental operations by simple recursion and integration, such as $sin = integral\ cos$ and $cos = 1 - integral\ sin$.

Dongol et al. [12] investigated convolution in a general algebraic setting that includes formal language concatenation. Kmett [31] observed that Moore machines are a special case of the cofree comonad. The connections between parsing and semirings have been explored deeply by Goodman [18, 19] and by Liu [37], building on the foundational work of Chomsky and Schützenberger [9]. Kmett [30] also explored some issues similar to those in the present paper, building on semirings and free semimodules, pointing out that the classic continuation monad can neatly represent linear functionals.

Kidney [26, 27] implemented a Haskell semiring library that helped with early implementations leading to the present paper, with a particular leaning toward convolution [29]. Several of the class instances given above, though independently encountered, also appear in that library.

REFERENCES

[1] Michael D. Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the complexity and performance of parsing with derivatives. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 224–236.

[2] Stephen Adams. 1993. Efficient sets—a balancing act. *Journal of Functional Programming* 3, 4 (1993), 553–561.

[3] Anonymous. 2019. Generalized Convolution and Efficient Language Recognition (Extended version).
[4] Guillaume Boisseau and Jeremy Gibbons. 2018. What you needa know about Yoneda: Profunctor optics and the Yoneda lemma (functional pearl). *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 84 (July 2018), 84:1–84:27 pages.
[5] Max Bolingbroke. 2011. Constraint kinds for GHC. Blog post. http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/.
[6] Ronald N. Bracewell. 2000. *The Fourier Transform and its Applications* (3rd ed. ed.). McGraw Hill.
[7] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016), e15.
[8] Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *J. ACM* 11 (1964), 481–494.
[9] N. Chomsky and M.P. Schützenberger. 1959. The algebraic theory of context-free languages. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 26. Elsevier, 118–161.
[10] Richard H. Connelly and F. Lockwood Morris. 1995. A generalization of the trie data structure. *Mathematical Structures in Computer Science* 5, 3 (1995).
[11] Stephen Dolan. 2013. Fun with semirings: A functional pearl on the abuse of linear algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. 101–110.
[12] 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.
[13] Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A play on regular expressions: Functional pearl. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. 357–368.
[14] Kunihiko Fukushima. 1988. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks* 1, 2 (1988), 119–130.
[15] Mai Gehrke, Daniela Petrisan, and Luca Reggio. 2017. Quantifiers on languages and codensity monads. *CoRR* abs/1702.08841 (2017).
[16] GHC Team. 2017. Glasgow Haskell Compiler user's guide—Release notes for version 8.2.1. https://downloads.haskell.org/~ghc/8.2.1/docs/html/users_guide/8.2.1-notes.html.
[17] Jonathan S. Golan. 2005. Some recent applications of semiring theory. In *International Conference on Algebra in Memory of Kostia Beidar*.
[18] Joshua Goodman. 1998. *Parsing Inside-Out*. Ph.D. Dissertation. Harvard University.
[19] Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics* 25, 4 (Dec. 1999), 573–605.
[20] Charles M. Grinstead and J. Laurie Snell. 2003. *Introduction to Probability*. AMS.
[21] Jeffrey Hass. 2013. Introduction to Computer Music: Volume One. http://www.indiana.edu/~emusic/etext/toc.shtml.
[22] Ralf Hinze. 2000. Generalizing generalized tries. *Journal of Functional Programming* 10, 4 (July 2000), 327–351.
[23] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. 2013. Unifying structured recursion schemes. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. 209–220.
[24] Sneha H.L. 2017. Better insight into DSP: 10 applications of convolution in various fields. https://www.allaboutcircuits.com/technical-articles/dsp-applications-of-convolution-part-2/.
[25] John Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Inform. Process. Lett.* 22, 3 (March 1986), 141–144.
[26] Donnacha Oisín Kidney. 2016. The *semiring-num* package. http://hackage.haskell.org/package/semiring-num. Haskell library.
[27] Donnacha Oisín Kidney. 2016. Semirings. Blog post. https://doisinkidney.com/posts/2016-11-17-semirings-lhs.html.
[28] Donnacha Oisín Kidney. 2017. Constrained applicatives. Blog post. https://doisinkidney.com/posts/2017-03-08-constrained-applicatives.html.
[29] Donnacha Oisín Kidney. 2017. Convolutions and semirings. Blog post. https://doisinkidney.com/posts/2017-10-13-convolutions-and-semirings.html.
[30] Edward Kmett. 2011. Free modules and functional linear functionals. Blog post. http://comonad.com/reader/2011/free-modules-and-functional-linear-functionals/.
[31] Edward Kmett. 2015. Moore for less. Blog post. https://www.schoolofhaskell.com/user/edwardk/moore/for-less.
[32] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc.
[33] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (May 2015), 436–444.
[34] Yann LeCun, Léon Bottou, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324.
[35] Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science* 4, 1 (1977), 59–76.
[36] Daan Leijen. 2002. *Data.Map*. http://hackage.haskell.org/package/containers/docs/Data-Map.html. Haskell library.

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

[38] Sandy Maguire. 2016. Wake up and smell the cofree comonads. Blog post. https://reasonablypolymorphic.com/blog/cofree-comonads/.

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

[40] M. Douglas McIlroy. 1999. Power series, power serious. *Journal of Functional Programming* 9, 3 (May 1999), 325–337.

[41] M. Douglas McIlroy. 2001. The music of streams. *Inform. Process. Lett.* 77, 2-4 (Feb. 2001), 189–195.

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

[43] Jürg Nievergelt and Edward M. Reingold. 1973. Binary search trees of bounded balance. *SIAM Journal of Computing* 2, 1 (1973), 33–43.

[44] Bryan O'Sullivan. 2014. criterion: a Haskell microbenchmarking library. http://www.serpentine.com/criterion/.

[45] Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives reexamined. *Journal of Functional Programming* 19, 2 (2009), 173–190.

[46] Chris Penner. 2017. Radix sort, trie trees, and maps from representable functors. Blog post. https://chrispenner.ca/posts/representable-discrimination.

[47] Taylor Petrick. 2016. Convolution part three: Common kernels. https://www.taylorpetrick.com/blog/post/convolution-part3.

[48] Dan Piponi. 2007. Monads for vector spaces, probability and quantum mechanics pt. I. Blog post. http://blog.sigfpe.com/2007/02/monads-for-vector-spaces-probability.html.

[49] Dan Piponi and Brent A. Yorgey. 2015. Polynomial functors constrained by regular expressions. In *MPC (Lecture Notes in Computer Science)*, Vol. 9129. Springer, 113–136.

[50] Fatemeh Pishdadian. 2017. Filters, reverberation & convolution. http://www.cs.northwestern.edu/~pardo/courses/eecs352/lectures/MPM16-topic9-Filtering.pdf. Lecture notes.

[51] Gabriel Radanne and Peter Thiemann. 2018. Regenerate: A language generator for extended regular expressions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2018)*. 202–214.

[52] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117.

[53] Steven W. Smith. 1997. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing.

[54] Milan Straka. 2012. Adams' Trees Revisited. In *Trends in Functional Programming*. 130–145.

[55] Tarmo Uustalu and Varmo Vene. 2005. The essence of dataflow programming. In *Proceedings of the Third Asian Conference on Programming Languages and Systems (APLAS'05)*. 2–18.

[56] Tarmo Uustalu and Varmo Vene. 2008. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science (ENTCS)* 203, 5 (June 2008), 263–284.

[57] Tarmo Uustalu and Varmo Vene. 2011. The recursion scheme from the cofree recursive comonad. *Electronic Notes in Theoretical Computer Science* 229, 5 (2011), 135 – 157. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

[58] Janis Voigtländer. 2008. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). 388–403.

[59] David Wilding. 2015. *Linear Algebra Over Semirings*. Ph.D. Dissertation. University of Manchester.

[60] Brent Yorgey. 2012. Monoids: Theme and variations (functional pearl). In *Proceedings of the 2012 Haskell Symposium*. 105–116.

[61] Brent Yorgey. 2017. Typeclassopedia. https://wiki.haskell.org/Typeclassopedia. Updated from original version in *The Monad Reader*, March 2009.

[62] Ian T. Young, Jan J. Gerbrands, and Lucas J. van Vliet. 1995. Fundamentals Of Image Processing.