Polynomial Functors Constrained by Regular Expressions

Dan Piponi[1] and Brent A. Yorgey[2]

[1] Google, dpiponi@gmail.com
[2] Williams College, byorgey@gmail.com

Abstract. We show that every regular language, via some DFA which accepts it, gives rise to a homomorphism from the semiring of polynomial functors to the semiring of $n \times n$ matrices over polynomial functors. Given some polynomial functor and a regular language, this homomorphism can be used to automatically derive a functor whose values have the same shape as those of the original functor, but whose sequences of leaf types correspond to strings in the language.
The primary interest of this result lies in the fact that certain regular languages correspond to previously studied derivative-like operations on polynomial functors, which have proven useful in program construction. For example, the regular language a^*ha^* yields the *derivative* of a polynomial functor, and b^*ha^* its *dissection*. Using our framework, we are able to unify and lend new perspective on this previous work. For example, it turns out that dissection of polynomial functors corresponds to taking *divided differences* of real or complex functions, and, guided by this parallel, we show how to generalize binary dissection to n-ary dissection.

Keywords: polynomial, functors, regular expressions, differentiation, dissection

1 Introduction

Consider the standard polymorphic singly-linked list type, which can be defined in Haskell [8] as:

$$\begin{aligned} &\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \\ &\qquad\qquad \mid\ \mathit{Cons}\ a\ (\mathit{List}\ a) \end{aligned}$$

This type is *homogeneous*, meaning that each element in the list has the same type as every other element.

Suppose, however, that we wanted lists with a different constraint on the types of its elements. For example, we might want lists whose elements alternate between two types a and b, beginning with a and ending with b.

One way to encode such an alternating list is with a pair of mutually recursive types, as follows:

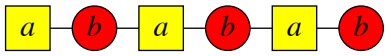

Fig. 1. A list with alternating types

$$\begin{aligned}&\textbf{data}\ \mathit{List}_1\ a\ b = \mathit{Nil}_1\\ &\qquad\qquad\mid\ \mathit{Cons}_1\ a\ (\mathit{List}_2\ a\ b)\\ &\textbf{data}\ \mathit{List}_2\ a\ b = \mathit{Cons}_2\ b\ (\mathit{List}_1\ a\ b)\end{aligned}$$

The required type is $\mathit{List}_1\ a\ b$: a value of type $\mathit{List}_1\ a\ b$ must be either empty (Nil_1) or contain a value of type a, followed by a value of type b, followed recursively by another $\mathit{List}_1\ a\ b$.

In fact, we can think of $\mathit{List}_1\ a\ b$ as containing values whose *shape* corresponds to the original *List* type (that is, there is a natural embedding from $\mathit{List}_1\ a\ a$ into $\mathit{List}\ a$, i.e. an injective polymorphic function $\forall a.\mathit{List}_1\ a\ a \to \mathit{List}\ a$), but whose sequence of element types corresponds to the *regular expression* $(ab)^*$, that is, any number of repetitions of the sequence ab.

We can easily generalize this idea to regular expressions other than $(ab)^*$ (though constructing the corresponding types may be complicated). We can also generalize to algebraic data types other than *List*, by considering the sequence of element types encountered by a canonical inorder traversal of each data structure [10]. That is, in general, given some algebraic data type and a regular expression, we consider the problem of constructing a corresponding algebraic data type "of the same shape" but with sequences of element types matching the regular expression.

For example, consider the following type *Tree* of nonempty binary trees with data stored in the leaves:

$$\begin{aligned}&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a\\ &\qquad\qquad\mid\ \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\end{aligned}$$

Consider again the problem of writing down a type whose values have the same shape as values of type $\mathit{Tree}\ a$, but where the data elements alternate between two types a and b, beginning with a leftmost element of type a and ending with a rightmost element of type b. An example can be seen in Fig. 2.

Suppose $\mathit{Tree}_{12}\ a\ b$ is such a type. Values of type $\mathit{Tree}_{12}\ a\ b$ cannot consist solely of a leaf node: there must be at least two elements, one of type a and one of type b. Hence a value of type $\mathit{Tree}_{12}\ a\ b$ must be a fork consisting of two subtrees. There are two ways this could happen. The left subtree could start with a and end with b, in which case the right subtree must also start with a and end with b. Or the left subtree could start with a and end with a, in which case the right subtree must start with b and end with b. So we are led to define

$$\begin{aligned}&\textbf{data}\ \mathit{Tree}_{12}\ a\ b = \mathit{Fork}_{12}\ (\mathit{Tree}_{12}\ a\ b)\ (\mathit{Tree}_{12}\ a\ b)\\ &\qquad\qquad\mid\ \mathit{Fork}'_{12}\ (\mathit{Tree}_{11}\ a\ b)\ (\mathit{Tree}_{22}\ a\ b)\end{aligned}$$

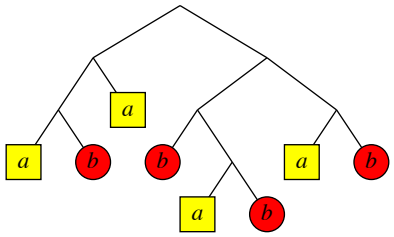

Fig. 2. A tree with alternating leaf types

where $Tree_{11}\ a\ b$ represents alternating trees with left and rightmost elements both of type a, and similarly for $Tree_{22}$.

Of course, we are now left with the task of defining $Tree_{11}$ and $Tree_{22}$, but we can carry out similar reasoning: for example a $Tree_{11}$ value can either be a single leaf of type a, or a branch with a $Tree_{12}$ and $Tree_{11}$, or a $Tree_{11}$ and $Tree_{21}$. All told, we obtain

$$\begin{aligned}
&\textbf{data}\ Tree_{11}\ a\ b = Leaf_{11}\ a \\
&\qquad\qquad\quad \mid Fork_{11}\ (Tree_{12}\ a\ b)\ (Tree_{11}\ a\ b) \\
&\qquad\qquad\quad \mid Fork'_{11}\ (Tree_{11}\ a\ b)\ (Tree_{21}\ a\ b)
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ Tree_{22}\ a\ b = Leaf_{22}\ b \\
&\qquad\qquad\quad \mid Fork_{22}\ (Tree_{22}\ a\ b)\ (Tree_{12}\ a\ b) \\
&\qquad\qquad\quad \mid Fork'_{22}\ (Tree_{21}\ a\ b)\ (Tree_{22}\ a\ b)
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ Tree_{21}\ a\ b = Fork_{21}\ (Tree_{22}\ a\ b)\ (Tree_{11}\ a\ b) \\
&\qquad\qquad\quad \mid Fork'_{21}\ (Tree_{21}\ a\ b)\ (Tree_{21}\ a\ b)
\end{aligned}$$

Any tree of type $Tree_{12}\ a\ b$ is now constrained to have alternating leaf node types. For example, here are two values of type $Tree_{12}\ Int\ Char$:

$$\begin{aligned}
&ex_1, ex_2 :: Tree_{12}\ Int\ Char \\
&ex_1 = Fork'_{12}\ (Leaf_{11}\ 1)\ (Leaf_{22}\ \texttt{'a'}) \\
&ex_2 = Fork'_{12}\ (Fork_{11}\ ex_1\ (Leaf_{11}\ 2))\ (Leaf_{22}\ \texttt{'b'})
\end{aligned}$$

ex_2 can also be seen in pictorial form in Fig. 3.

While this works, the procedure was somewhat *ad hoc*. We reasoned about the properties of the pieces that result when a string matching $(ab)^*$ is split into two substrings, and used this to find corresponding types for the subtrees. One might wonder why we ended up with four mutually recursive types—is there any simpler solution? And how well does this sort of reasoning extend to more complicated structures or regular expressions? Our goal will be to derive a more principled way to do this analysis for any regular language and any suitable (*polynomial*) data type.

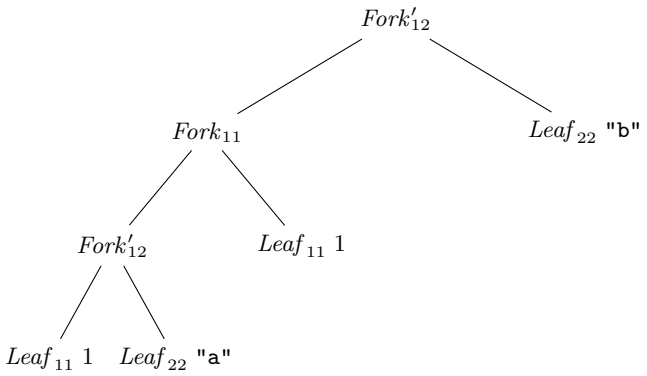

Fig. 3. A tree with alternating leaf types

For certain languages, this problem has already been explored in the literature, though without being phrased in terms of regular languages. For example, consider the regular language a^*ha^*. It matches sequences of as with precisely one occurrence of h somewhere in the middle. Data structures whose inorder sequence of element types matches a^*ha^* have all elements of type a, except for one which has type h. This corresponds to a zipper type [5] with elements of type h at the 'focus'; if we substitute the unit type for h, we get the *derivative* of the original type [1] (Fig. 4). Likewise, the regular language b^*ha^* corresponds to *dissection types* [9].

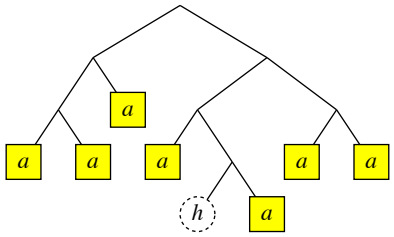

Fig. 4. A tree corresponding to the regular language a^*ha^*

Zippers, derivatives and dissections are usually described using Leibniz rules and their generalizations. We'll show how these rules can be placed in a more general framework applying to any regular language.

In the remainder of the paper, we first review some standard results about regular languages and DFAs (Sect. 2). We describe our framework informally (Sect. 3) and give some examples of its application (Sect. 4) and describe an alternative encoding which can be more convenient in practice (Sect. 5). We conclude with a discussion of derivatives (Sect. 6) and divided differences (Sect. 7).

2 Regular Expressions and DFAs

We begin with a review of the basic theory of regular languages and deterministic finite automata in Sections 2.1–2.3. Readers already familiar with this theory may safely skip these sections. In Section 2.4 we introduce some preliminary material on star semirings which, though not novel, may not be as familiar to readers.

2.1 Regular Expressions

A *regular expression* [6] over an alphabet Σ is a term of the following grammar:

$$R ::= \bullet \mid \varepsilon \mid a \in \Sigma \mid R + R \mid RR \mid R^* \tag{1}$$

When writing regular expressions, we allow parentheses for disambiguation, and adopt the common convention that Kleene star (R^*) has higher precedence than concatenation (RR), which has higher precedence than alternation ($R+R$).

Semantically, we can interpret each regular expression R as a set of strings $[\![R]\!] \subseteq \Sigma^*$, where Σ^* denotes the set of all finite sequences built from elements of Σ. In particular,

- $[\![\bullet]\!] = \varnothing$ denotes the empty set.
- $[\![\varepsilon]\!] = \{\varepsilon\}$ denotes the singleton set containing the empty string.
- $[\![a]\!] = \{a\}$ denotes the singleton set containing the length-1 sequence a.
- $[\![R_1 + R_2]\!] = [\![R_1]\!] \cup [\![R_2]\!]$.
- $[\![R_1 R_2]\!] = [\![R_1]\!]\,[\![R_2]\!]$, where $L_1 L_2$ denotes pairwise concatenation of sets,

$$L_1 L_2 = \{s_1 s_2 \mid s_1 \in L_1, s_2 \in L_2\}.$$

- $[\![R^*]\!] = [\![R]\!]^*$, where L^* denotes the least fixed point solution of

$$L^* = \{\varepsilon\} \cup LL^*.$$

  Note that such a least fixed point must exist by the Knaster-Tarski theorem [18], since the mapping $\varphi(S) = \{\varepsilon\} \cup LS$ is monotone, that is, if $S \subseteq T$ then $\varphi(S) \subseteq \varphi(T)$.

Finally, a *regular language* over the alphabet Σ is a set $L \subseteq \Sigma^*$ which is the interpretation $L = [\![R]\!]$ of some regular expression R.

2.2 DFAs

A *deterministic finite automaton* (DFA) is a quintuple $(Q, \Sigma, \delta, q_0, \mathcal{A})$ consisting of

- a nonempty set of states Q,
- a set of input symbols Σ,
- a *transition function* $\delta : Q \times \Sigma \to Q$,

– a distinguished *start state* $q_0 \in Q$, and
– a set $\mathcal{A} \subseteq Q$ of *accept states*. (One often sees F used to represent the set of accept or "final" states, but this would conflict with our use of F to represent functors later.)

We can "run" a DFA on an input string by feeding it symbols from the string one by one. When encountering the symbol s in state q, the DFA changes to state $\delta(q, s)$. If a DFA beginning in its start state q_0 ends in state q' after being fed a string in this way, we say the DFA *accepts* the string if $q' \in \mathcal{A}$, and *rejects* the string otherwise. Thus, a DFA D can be seen as defining a subset $L_D \subseteq \Sigma^*$ of the set of all possible strings, namely, those strings which it accepts.

We can draw a DFA as a directed multigraph where each graph edge is labeled by a symbol from Σ. Each state is a vertex, and an edge is drawn from q_1 to q_2 and labeled with symbol s whenever $\delta(q_1, s) = q_2$. In addition, we indicate accept states with a double circle, and always label the start state as 1. We can think of the state of the DFA as "walking" through the graph each time it receives an input. Fig. 5 shows an example.

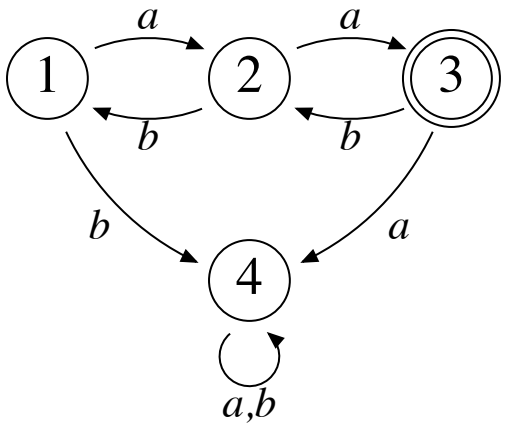

Fig. 5. An example DFA

It is convenient to allow the transition function δ to be partial. Operationally, encountering a state q and input s for which $\delta(q, s)$ is undefined corresponds to the DFA *rejecting* its input. This often simplifies matters, since we may omit "sink states" from which there is no path to any accepting state, making δ undefined whenever it would have otherwise yielded such a sink state. For example, the DFA from Fig. 5 may be simplified to the one shown in Fig. 6, by dropping state 4.

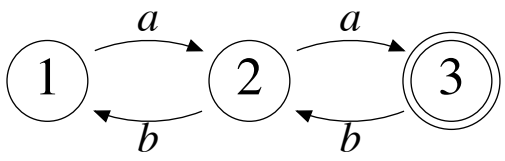

Fig. 6. Example DFA, simplified

As is standard, we may define $\delta^* : Q \times \Sigma^* \rightharpoonup Q$ as an iterated version of δ:

$$\delta^*(q, \varepsilon) = q \tag{2}$$

$$\delta^*(q, s\omega) = \delta^*(\delta(q, s), \omega) \tag{3}$$

If $\delta^*(q_0, \omega) = q_1$, then we say that the string ω "takes" or "drives" the DFA from state q_0 to state q_1. More generally, given a string ω, we can partially apply δ^* to obtain a "driving function" $\chi : Q \rightharpoonup Q$ which encodes how the string ω drives the DFA: if the DFA starts in state q then after processing ω it will either halt with an error or end in state $\chi(q)$.

2.3 Kleene's Theorem

Connecting the previous two sections is *Kleene's Theorem*, which says that the theory of regular expressions and the theory of DFAs are really "about the same thing". In particular, the set of strings accepted by a DFA is always a regular language, and conversely, for every regular language there exists a DFA which accepts it. Moreover, the proof of the theorem is constructive: given a regular expression, we may algorithmically construct a corresponding DFA, and vice versa. For example, the regular expression b^*ha^* corresponds to the DFA shown in Fig. 7. It is not hard to verify that strings taking the DFA from state 1 to state 2 (the accept state) are precisely those matching the regular expression b^*ha^*.

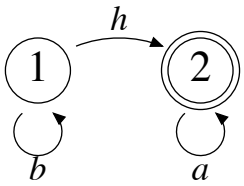

Fig. 7. A DFA for b^*ha^*

The precise details of these constructions are not important for the purposes of this paper; interested readers should consult a reference such as Sipser [15].

We note in passing that one can also associate *nondeterministic* finite automata (NFAs) to regular expressions, and the remainder of the story of this paper could probably be retold using NFAs. However, it is not clear whether we would gain any benefit from making this generalization, so we will stick with the simpler notion of DFAs.

2.4 Semirings

A *semiring* is a set R equipped with two binary operations, $+$ and $\cdot$, and two distinguished elements, $0, 1 \in R$, such that

- $(+, 0)$ is a commutative monoid (that is, 0 is an identity for $+$, and $+$ is commutative and associative),
- $(\cdot, 1)$ is a monoid,
- $\cdot$ distributes over $+$ from both the left and the right, that is, $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$, and
- 0 is an annihilator for $\cdot$, that is $r \cdot 0 = 0 \cdot r = 0$ for all $r \in R$.

Examples of semirings include:

- $(\mathit{Bool}, \vee, \mathit{False}, \wedge, \mathit{True})$, boolean values under disjunction and conjunction;
- $(\mathbb{N}, +, 0, \cdot, 1)$, the natural numbers under addition and multiplication;
- $(\mathbb{R}^+ \cup \{-\infty\}, \max, -\infty, +, 0)$, the nonnegative real numbers (adjoined with $-\infty$) under maximum and addition;
- the set of regular languages forms a semiring under the operations of union and pairwise concatenation, with $0 = \varnothing$ and $1 = \{\varepsilon\}$.

A *star semiring* or *closed semiring* [7] has an additional operation, $(-)^*$, satisfying the axiom

$$r^* = 1 + r \cdot r^* = 1 + r^* \cdot r\,, \tag{4}$$

for all $r \in R$. Intuitively, $r^* = 1 + r + r^2 + r^3 + \ldots$ (although such infinite sums do not necessarily make sense in all semirings). The semiring of regular languages is closed, via Kleene star.[3]

If R is a semiring, then the set of $n \times n$ matrices with elements in R is also a semiring, where matrix addition and multiplication are defined in the usual manner in terms of addition and multiplication in R. If R is a star semiring, then a star operator can also be defined for matrices; for details see Lehmann [7] and Dolan [2].

Finally, a *semiring homomorphism* is a mapping from the elements of one semiring to another that preserves the semiring structure, that is, that sends 0 to 0, 1 to 1, and preserves addition and multiplication.

[3] In fact, regular languages (and several of the other examples above) form *Kleene algebras*, which essentially add to a star semiring the requirement that $+$ is idempotent ($a + a = a$). However, for our purposes we do not need the extra restriction.

3 DFAs and Matrices of Functors

Viewing regular expressions through the lens of DFAs gives us exactly the tools we need to generalize our *ad hoc* analysis from the introduction.

3.1 A More Principled Derivation

Consider again the task of encoding a type with the same shape as

$$\begin{aligned}&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a\\ &\qquad\qquad\ \ |\ \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\end{aligned}$$

whose sequence of element types matches the regular expression $(ab)^*$, as in the introduction. This time, however, we will think about it from the point of view of the corresponding DFA, shown in Fig. 8.

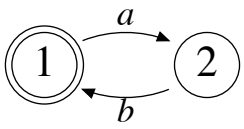

Fig. 8. A DFA for $(ab)^*$

The key is to consider not just the data type we are ultimately interested in—in this case, those trees which take the DFA from state 1 to itself—but an entire family of related types. In particular, let $T_{ij}\ a\ b$ denote the type of binary trees whose element type sequences take the DFA from state i to state j. Since the DFA has two states, there are four such types:

- $T_{11}\ a\ b$ — this is the type of trees we are primarily interested in constructing, whose leaf sequences match $(ab)^*$.
- $T_{12}\ a\ b$ — trees of this type have leaf sequences which take the DFA from state 1 to state 2; that is, they match the regular expression $a(ba)^*$ (or, equivalently, $(ab)^*a$).
- $T_{21}\ a\ b$ — trees matching $b(ab)^*$.
- $T_{22}\ a\ b$ — trees matching $(ba)^*$.

What does a tree of type T_{11} look like? It cannot be a leaf, because a single leaf takes the DFA from state 1 to 2 or vice versa. It must be a pair of trees, which together take the DFA from state 1 to state 1. There are two ways for that to happen: both trees could themselves begin and end in state 1; or the first tree could take the DFA from state 1 to state 2, and the second from state 2 to state 1. We can carry out a similar analysis for the other three types. In fact, we have already carried out this exact analysis in the introduction, but it is now a

bit less ad hoc. In particular, we can now see that we end up with four mutually recursive types precisely because the DFA for $(ab)^*$ has two states, and we need one type for each ordered pair of states.

In general, given a DFA with states Q and alphabet $\Sigma = \{a_1, \ldots, a_n\}$, we get a mutually recursive family of types

$$T_{ij}\ a_1\ \ldots\ a_n$$

indexed by a pair of states from Q and by one type argument for each alphabet symbol. We are ultimately interested in types of the form $T_{q_0 k}$ where $k \in \mathcal{A}$, that is, types which are indexed by the start state and some accept state of the DFA.

Though shifting our point of view to DFAs has given us a better framework for determining which types we must define, we still had to reason on a case-by-case basis to determine the definitions of these types. The next two sections show how we can concisely and elegantly formalize this process in terms of *matrices*.

3.2 Polynomial Functors

We now abstract away from the particular details of Haskell data types and work in terms of a simple language of *polynomial functors*. We inductively define the universe **Fun** of polynomial functors as follows, simultaneously giving both syntax and semantics.

- $K_A \in \mathbf{Fun}$ denotes the constant functor $K_A\ a = A$, which ignores its argument and yields A.
- $X \in \mathbf{Fun}$ denotes the identity functor $X\ a = a$.
- Given $F, G \in \mathbf{Fun}$, we can form their sum, $F + G \in \mathbf{Fun}$, with $(F + G)\ a = F\ a + G\ a$.
- We can also form products of functors, $(F \times G)\ a = F\ a \times G\ a$. We often abbreviate $F \times G$ as FG.
- Finally, we allow functors to be defined by mutually recursive systems of equations

$$\begin{cases} F_1 = \Phi_1(F_1, \ldots, F_n) \\ \vdots \\ F_n = \Phi_n(F_1, \ldots, F_n), \end{cases}$$

  where each Φ_k is a polynomial functor expression with free variables in $\{F_1, \ldots, F_n\}$, and interpret them using a standard least fixed point semantics. For example, the single recursive equation $L = 1 + X \times L$ denotes the standard type of (finite) polymorphic lists. As another example, the pair of mutually recursive equations

$$\begin{aligned} E &= K_{Unit} + X \times O \\ O &= X \times E \end{aligned}$$

  defines the types of even- and odd-length polymorphic lists. Here, *Unit* denotes the unit type with a single inhabitant.

It is worth pointing out that functors form a semiring (up to isomorphism) under $+$ and $\times$, where $1 = K_{Unit}$ and $0 = K_{Void}$ (*Void* denotes the type with no inhabitants). We therefore will simply write 0 and 1 in place of K_{Unit} and K_{Void}. In fact, functors also form a star semiring, with the polymorphic list type playing the role of the star operator, that is, $F^* = 1 + F \times F^*$.

The above language also generalizes naturally from unary to n-ary functors. We write $\mathbf{Fun}_n$ for the universe of n-ary polynomial functors, so $\mathbf{Fun} = \mathbf{Fun}_1$.

- $K_A\ a_1\ \ldots\ a_n = A$.
- The identity functor X generalizes to the family of projections X_m, where

$$X_m\ a_1\ \ldots\ a_n = a_m.$$

  That is, X_m is the functor which yields its mth argument, and may be regarded as an n-ary functor for any $n \geqslant m$. More generally, the arguments to a functor can be labeled by the elements of some alphabet Σ, instead of being numbered positionally, and we write $\mathbf{Fun}_\Sigma$ for the universe of such functors. In that case, for $a \in \Sigma$ we write X_a for the projection which picks out the argument labeled by a.
- $(F + G)\ a_1\ \ldots\ a_n = (F\ a_1\ \ldots\ a_n) + (G\ a_1\ \ldots\ a_n)$.
- $(F \times G)\ a_1\ \ldots\ a_n = (F\ a_1\ \ldots\ a_n) \times (G\ a_1\ \ldots\ a_n)$.
- Recursion also generalizes straightforwardly.

Of course, n-ary functors also form a semiring for any n.

As an example, the Haskell type

data *S a b* = *Apple a* | *Banana b* | *Fork* (*S a b*) (*S a b*)

corresponds to the bifunctor (that is, 2-ary functor) $S = X_a + X_b + S \times S$; we may also abbreviate $S \times S$ as S^2.

By induction over functor descriptions, we may define $\mathcal{S} : \mathbf{Fun}_\Sigma \to \mathcal{P}(\Sigma^*)$ which gives the sequences of leaf types that can occur in the values of a given functor. Thinking of values of a given functor as trees, $\mathcal{S}(-)$ corresponds to an inorder traversal. That is:

$$\begin{aligned}
\mathcal{S}(0) &= \varnothing \\
\mathcal{S}(K_A) &= \{\varepsilon\} \quad (A \neq \mathit{Void}) \\
\mathcal{S}(X_a) &= \{a\} \\
\mathcal{S}(F + G) &= \mathcal{S}(F) \cup \mathcal{S}(G) \\
\mathcal{S}(F \times G) &= \mathcal{S}(F)\mathcal{S}(G)
\end{aligned}$$

Finally, given a system $F_m = \Phi_m(F_1, \ldots, F_n)$ we simply set

$$\mathcal{S}(F_m) = \mathcal{S}(\Phi_m(F_1, \ldots, F_n))$$

for each m, and take the least fixed point (ordering sets by inclusion). For example, given the list functor $L = 1 + XL$, we obtain

$$\mathcal{S}(L) = \{\varepsilon\} \cup \{1\sigma \mid \sigma \in \mathcal{S}(L)\}$$

whose least fixed point is the infinite set $\{\varepsilon, 1, 11, 111, \ldots\}$ as expected.

3.3 Matrices of Functors

Now suppose we have a unary functor F and some DFA $D = (Q, \Sigma, \delta, q_0, \mathcal{A})$. Let $F_{ij} \in \mathbf{Fun}_\Sigma$ denote the type with the same shape as F but whose sequences of leaf types take D from state i to state j. We are ultimately interested in constructing

$$\sum_{k \in \mathcal{A}} T_{q_0 k},$$

the sum of all types T_{ij} whose leaf sequences start in state q_0 and tahe the DFA to some accept state. Note that F_{ij} has arity Σ, that is, there is a leaf type corresponding to each alphabet symbol of D. We can deduce F_{ij} compositionally, by recursion on the syntax of functor expressions.

- The constant functor K_A creates structures containing no elements, *i.e.* which do not cause the DFA to transition at all. So the only way a K_A-structure can take the DFA from state i to state j is if $i = j$:

$$(K_A)_{ij} = \begin{cases} K_A & i = j \\ 0 & i \neq j \end{cases} \tag{5}$$

  As a special case, the functor $1 = K_{Unit}$ yields

$$1_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}. \tag{6}$$

- A value with shape $F + G$ is either a value with shape F or a value with shape G; so the set of $F + G$ shapes taking the DFA from state i to state j is the disjoint sum of the corresponding F and G shapes:

$$(F + G)_{ij} = F_{ij} + G_{ij}. \tag{7}$$

- Products are more interesting. An FG-structure consists of an F-structure paired with a G-structure, whose leaf types drive the DFA in sequence. Hence, in order to take the DFA from state i to state j overall, the F-structure must take the DFA from state i to some state k, and then the G-structure must take it from k to j. This works for any state k, so $(FG)_{ij}$ is the sum over all such possibilities. Thus,

$$(FG)_{ij} = \sum_{k \in Q} F_{ik} G_{kj}. \tag{8}$$

- Finally, for a recursive system of functors

$$\overline{F_m} = \overline{\Phi_m(F_1, \ldots, F_n)},$$

  we may mutually define

$$(F_m)_{ij} = \left(\Phi_m(F_1, \ldots, F_n)\right)_{ij},$$

  interpreted via the same least fixed point semantics.

The above rules for 1, sums, and products might look familiar: in fact, they are just the definitions of the identity matrix, matrix addition, and matrix product. That is, given some functor F and DFA D, we can arrange all the F_{ij} in a matrix, $[F]_D$, whose (i,j)th entry is F_{ij}. (We also write simply $[F]$ when D can be inferred.) Then we can rephrase (6)–(8) above as

- $[1]_D = I_{|\Sigma|}$, that is, the $|\Sigma| \times |\Sigma|$ identity matrix, with ones along the main diagonal and zeros everywhere else;
- $[F + G]_D = [F]_D + [G]_D$; and
- $[FG]_D = [F]_D [G]_D$.

So far, given a DFA D, we have the makings of a homomorphism from the semiring of arity-1 functors to the semiring of $|Q| \times |Q|$ matrices of arity-Σ functors. However, there is still some unfinished business, namely, the interpretation of $[X]_D$. This gets at the heart of the matter, and to understand it, we must take a slight detour.

3.4 Transition Matrices

Given a simple directed graph G with n nodes, its *adjacency matrix* is an $n \times n$ matrix M_G with a 1 in the i, j position if there is an edge from node i to node j, and a 0 otherwise. It is a standard observation that the mth power of M_G encodes information about length-m paths in G; specifically, the i, j entry of M_G^m is the number of distinct paths of length m from i to j. This is because a path from i to j of length m is the concatenation of a length-$(m-1)$ path from i to some k followed by an edge from k to j, so the total number of length-m paths is the sum of such paths over all possible k; this is exactly what is computed by the matrix multiplication $M_G^{m-1} M = M_G^m$.

However, as observed independently by O'Connor [11] and Dolan [2], and as is standard weighted automata theory [3], this can be generalized by parameterizing the construction over an arbitrary semiring. In particular, we may suppose that the edges of G are labeled by elements of some semiring R, and form the adjacency matrix M_G as before, but using the labels on edges, and $0 \in R$ for missing edges. The mth power of M_G still encodes information about length-m paths, but the interpretation depends on the specific choice of R and on the edge labeling. Choosing the semiring $(\mathbb{N}, +, \cdot)$ with all edges labeled by 1 gives us a count of distinct paths, as before. If we choose $(\mathit{Bool}, \vee, \wedge)$ and label each edge with *True*, the i, j entry of M_G^m tells us whether there exists any path of length m from i to j. Choosing $(\mathbb{R}, \min, +)$ and labeling edges with costs yields the minimum cost of length-m paths; choosing $(\mathcal{P}(\Sigma^*), \cup, \cdot)$ (that is, languages over some alphabet Σ under union and pairwise concatenation) and labeling edges with elements from Σ yields sets of words corresponding to length-m paths.

Moreover, if R is a star semiring, then M_G^* encodes information about paths of *any* length (recall that, intuitively, $M_G^* = I + M_G + M_G^2 + M_G^3 + \dots$). Choosing $R = (\mathbb{R}, \min, +)$ and computing M_G^* thus solves the all-pairs shortest paths problem; $(\mathit{Bool}, \vee, \wedge)$ tells us whether any paths exist between each pair of nodes; and

so on. Note that $(\mathbb{N}, +, \cdot)$ is not closed, but we can make it so by adjoining $+\infty$; this corresponds to the observation that the number of distinct paths between a pair of nodes in a graph may be infinite if the graph contains any cycles.

Of course, DFAs can also be thought of as graphs. Suppose we have a DFA D, a semiring R, and a function $\Sigma \to R$ assigning an element of R to each alphabet symbol. In this context, we call the adjacency matrix for D a *transition matrix*.[4] The graph of a DFA may not be simple, that is, there may be multiple edges in a DFA between a given pair of nodes, each corresponding to a different alphabet symbol. We can handle this by summing in R. That is, the transition matrix M_D is the $|Q| \times |Q|$ matrix over R whose component at i, j is the sum, over all edges from i to j, of the R-values corresponding to their labels.

For example, consider the DFA in Fig. 6, and the semiring $(\mathbb{N}, +, \cdot)$. If we send each edge label (i.e. alphabet symbol) to 1, we obtain the transition matrix

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

The mth power of this matrix tells us how many strings of length m take the DFA from one given state to another. If we instead send each edge label to the singleton language containing only that symbol as a length-1 string, as a member of the semiring of regular languages, we obtain the transition matrix

$$\begin{bmatrix} \varnothing & \{a\} & \varnothing \\ \{b\} & \varnothing & \{a\} \\ \varnothing & \{b\} & \varnothing \end{bmatrix}.$$

The star of this matrix yields the complete set of strings that drives the DFA between each pair of states.

We can now see how to interpret $[X]_D$: it is the transition matrix for D, taken over the semiring of arity-Σ functors, where each transition a is replaced by the functor X_a. That is, in general, each entry of $[X]_D$ will consist of a (possibly empty) sum of functors

$$\sum_{\substack{a \in \Sigma \\ \delta(i,a)=j}} X_a.$$

By definition, these will drive the DFA in the proper way; moreover, sums of X_a are the only functors with the same shape as X.

4 Examples

To make things more concrete, we can revisit some familiar examples using our new framework. As a first example, consider the regular expression $(aa)^*$,

[4] Textbooks on automata often define the *transition matrix* for a DFA as the $|Q| \times |\Sigma|$ matrix with its q, s entry equal to $\delta(q, s)$. This is just a particular representation of the function δ, and quite uninteresting, so we co-opt the term *transition matrix* to refer to something more worthwhile.

corresponding to the DFA shown in Fig. 9, along with the standard polymorphic list type, $L = 1 + XL$. The matrix for L can be written

$$[L] = \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}.$$

The punchline is that we can take the recursive equation for L and simply apply

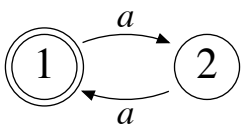

Fig. 9. A DFA for $(aa)^*$

the homomorphism to both sides, resulting in the matrix equation

$$[L] = [1 + XL] = [1] + [X]\,[L]\,,$$

where $[X]$ is the transition matrix for D, namely

$$[X] = \begin{bmatrix} 0 & X_a \\ X_a & 0 \end{bmatrix}.$$

Expanding out this matrix equation and performing the indicated matrix operations yields

$$\begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & X_a \\ X_a & 0 \end{bmatrix} \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{bmatrix}$$
$$= \begin{bmatrix} 1 + X_a L_{21} & X_a L_{22} \\ X_a L_{11} & 1 + X_a L_{12} \end{bmatrix}.$$

We can see that L_{11} and L_{22} are isomorphic, as are L_{12} and L_{21}; this is because the DFA D has a nontrivial automorphism (ignoring start and accept states). Thinking about the meaning of paths through the DFA, we see that L_{11} is the type of lists with even length, and L_{21}, lists with odd length. More familiarly:

```
data EvenList a = EvenNil | EvenCons a (OddList a)
data OddList a  = OddCons a (EvenList a)
```

As another example, consider again the recursive tree type given by $T = X + T^2$, along with the two-state DFA for $(ab)^*$ shown in Fig. 8. Applying the homomorphism, we obtain

$$[T] = \left[X + T^2\right] = [X] + [T]^2\,,$$

where

$$[X] = \begin{bmatrix} 0 & X_a \\ X_b & 0 \end{bmatrix}.$$

This yields

$$\begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} = \begin{bmatrix} 0 & X_a \\ X_b & 0 \end{bmatrix} + \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix}^2$$
$$= \begin{bmatrix} T_{11}^2 + T_{12}\,T_{21} & X_a + T_{11}\,T_{12} + T_{12}\,T_{22} \\ X_b + T_{21}\,T_{11} + T_{22}\,T_{21} & T_{21}\,T_{12} + T_{22}^2 \end{bmatrix}.$$

Equating the left- and right-hand sides elementwise yields precisely the definitions for T_{ij} we derived in Section 1.

As a final example, consider the type $T = X + T^2$ again, but this time constrained by the regular expression b^*ha^*, with transition matrix $\begin{bmatrix} X_b & X_h \\ 0 & X_a \end{bmatrix}$. Applying the homomorphism yields

$$\begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} = \begin{bmatrix} X_b & X_h \\ 0 & X_a \end{bmatrix} + \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix}^2$$
$$= \begin{bmatrix} X_b + T_{11}^2 + T_{12}\,T_{21} & X_h + T_{11}\,T_{12} + T_{12}\,T_{22} \\ T_{21}\,T_{11} + T_{22}\,T_{21} & X_a + T_{21}\,T_{12} + T_{22}^2 \end{bmatrix}.$$

Here something strange happens: looking at the DFA, it is plain that there are no paths from state 2 to state 1, and we therefore expect the corresponding type T_{21} to be empty. However, it does not look empty at first sight: we have $T_{21} = T_{21}\,T_{11} + T_{22}\,T_{21}$. In fact, it *is* empty, since we are interpreting recursively defined functors via a least fixed point semantics, and it is not hard to see that 0 is in fact a fixed point of the above equation for T_{21}. In practice, we can perform a reachability analysis for a DFA beforehand (e.g. by taking the star of its transition matrix under $(Bool, \vee, \wedge)$) to see which states are reachable from which other states; if there is no path from i to j then we know $T_{ij} = 0$, which can simplify calculations. For example, substituting $T_{21} = 0$ into the above equation and simplifying yields

$$\begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} = \begin{bmatrix} X_b + T_{11}^2 & X_h + T_{11}\,T_{12} + T_{12}\,T_{22} \\ 0 & X_a + T_{22}^2 \end{bmatrix}.$$

5 An Alternative Representation

One way to look at the examples shown so far is that we have essentially had to *duplicate* the initial functor F, resulting in several slightly different copies, each with a slightly different set of "constructors", in order to keep track of which constructors are allowed at which points. Such encodings would be extremely trying to work with in practice, requiring much tedious case analysis. In a language with a sufficiently expressive type system, however, we do not need to duplicate

anything, but can instead make use of types to dictate which constructors are allowed in which situations.

What information, exactly, do we need to keep around at the level of types? It is not enough to just index by a pair of DFA states; the problem is that each constructor may correspond to *multiple* possible pairs of states. In fact, what we need is to index by an entire *driving function*. Given some functor T, the idea is to produce just a *single* n-ary functor T_χ indexed by a (total!) driving function $\chi : Q \to Q$. A value of type T_χ is a structure with a shape allowed by T, whose sequence of leaf types, taken together, drives the DFA in the way encoded by χ. The desired type can then be selected as the sum of all types indexed by driving functions taking the start state to some accepting state.

For details of this encoding, see Yorgey [19]. Encoding driving functions and their composition requires only natural numbers and lists, so they can be encoded in any language which allows encoding these at the level of types.

The above approach requires indexing by *total* driving functions. As pointed out by an anonymous reviewer, one can also index by *relations* which can encode partial driving functions. For example, considering again the DFA for b^*ha^* shown in Fig. 7, and the tree type $T = X + T^2$, we have the following Haskell code. *States* encodes the states of the DFA, and *Trans* encodes a relation on states, with each constructor corresponding to an edge in the DFA. The original *Tree a* type is transformed into *Tree′*, where the *Leaf* constructor is parameterized by a transition, and the *Fork* constructor encodes a sum via existential quantification of k. *Tree′* could also be parameterized over an arbitrary relation of the appropriate kind, which allows constructing *Tree* variants constrained by any DFA over an alphabet of size 3.

```
{-# LANGUAGE DataKinds, GADTs, KindSignatures, PolyKinds #-}
data States = S1 | S2

data Trans b h a :: State → State → ∗ where
  B :: b → Trans b h a S1 S1
  H :: h → Trans b h a S1 S2
  A :: a → Trans b h a S2 S2

data Tree′ :: ∗ → ∗ → ∗ → State → State → ∗ where
  Leaf :: Trans b h a i j → Tree′ r b h a i j
  Fork :: Tree′ r b h a i k → Tree′ r b h a k j → Tree′ r b h a i j
```

6 Derivatives, Again

Now that we have seen the general framework, let's return to the specific application of computing *derivatives* of data types. In order to compute a derivative, we need the DFA for the regular expression a^*ha^*, shown in Fig. 10. The corresponding transition matrix is

$$[X] = \begin{bmatrix} X_a & X_h \\ 0 & X_a \end{bmatrix}.$$

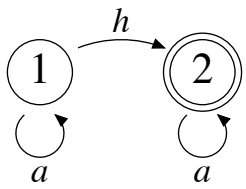

Fig. 10. A DFA for derivatives

Suppose we start with a functor defined as a product:

$$F = GH$$

Expanding via the homomorphism to matrices of bifunctors, we obtain

$$\begin{bmatrix} F_{11} & F_{12} \\ 0 & F_{22} \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} \\ 0 & G_{22} \end{bmatrix} \begin{bmatrix} H_{11} & H_{12} \\ 0 & H_{22} \end{bmatrix}$$

(the occurrences of 0 correspond to the observation that there are no paths in the DFA from state 2 to state 1). Let's consider each of the nonzero F_{ij} in turn. First, we have

$$F_{11} = G_{11} \times H_{11}$$

F_{11} is simply the type of structures whose leaves take the DFA from state 1 to itself and so whose leaves match the regular expression a^*; hence we have $F_{11}\ a\ h \cong F\ a$ (and similarly for G_{11}, H_{11}, F_{22}, G_{22}, and H_{22}). We also have

$$F_{12} = G_{11}H_{12} + G_{12}H_{22} \cong GH_{12} + G_{12}H.$$

This looks suspiciously like the usual Leibniz law for the derivative of a product (i.e. the "product rule" for differentiation). We also know that

$$1_{12} = 0$$

and

$$X_{12} = X_h,$$

and if $F = G+H$ then $F_{12} = G_{12}+H_{12}$. If we substitute the unit type for h, these are precisely the rules for differentiating polynomials. So F_{12} is the derivative of F.

There is another way to look at this. Write

$$[X] = \begin{bmatrix} X_a & X_h \\ 0 & X_a \end{bmatrix} = X_a I + d$$

where

$$d = \begin{bmatrix} 0 & X_h \\ 0 & 0 \end{bmatrix}$$

Note that $d^2 = 0$. Note also that

$$(X_a I)d = \begin{bmatrix} 0 & X_a X_h \\ 0 & 0 \end{bmatrix}$$

and

$$d(X_a I) = \begin{bmatrix} 0 & X_h X_a \\ 0 & 0 \end{bmatrix}.$$

Treating the product of functors as commutative is problematic in our setting, since we care about the precise sequence of leaf types. However, in this particular instance, we can specify that X_h commutes with everything, which corresponds to letting the "hole" of type h "float" to the outside—typically, when constructing a zipper structure, one does this anyway, storing the focused element separately from the rest of the structure. Under this interpretation, then, $(X_a I)$ and d commute even though matrix multiplication is not commutative in general. We then note that

$$(X_a I + d)^n = (X_a I)^n + n(X_a I)^{n-1} d,$$

making use of this special commutativity and the fact that $d^2 = 0$, annihilating all the subsequent terms. We can linearly extend this to an entire polynomial f, that is,

$$\begin{aligned} f([X]) &= f(X_a I + d) \\ &= f(X_a I) + f'(X_a I)d \\ &= \begin{bmatrix} f(X_a) & 0 \\ 0 & f(X_a) \end{bmatrix} + \begin{bmatrix} 0 & f'(X_a)X_h \\ 0 & 0 \end{bmatrix} \end{aligned}$$

The matrix d is thus playing a role similar to an "infinitesimal" in calculus, where the expression dx is manipulated informally as if $(dx)^2 = 0$. (Compare with the dual numbers described by [16].)

7 Dissection and Divided Differences

Consider again the regular expression $b^* h a^*$. Data structures with leaf sequences matching this pattern have a "hole" of type h, with values of type b to the left of the hole and values of type a to the right (Fig. 11).[5]

7.1 Dissection

Such structures have been considered by McBride [9], who refers to them as *dissections* and shows how they can be used, for example, to generically derive tail-recursive maps and folds.

[5] Typically we substitute the unit type for h, but it makes the theory work more smoothly if we represent it initially with a unique type variable.

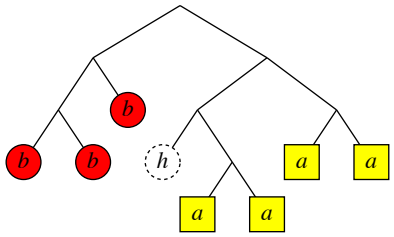

Fig. 11. A tree with leaf sequence matching b^*ha^*

Given a functor F, McBride uses $\triangle F$ to denote the bifunctor which is the dissection of F (where the unit type has been substituted for h). We have

$$\triangle X\ b\ a = 1,$$

since a dissected X consists merely of a hole,

$$\triangle 1\ b\ a = 0,$$

and

$$\triangle(F + G)\ b\ a = \triangle F\ b\ a + \triangle G\ b\ a.$$

The central construction is the Leibniz rule for dissection of a product,

$$\triangle(F \times G) = \angle F \times \triangle G + \triangle F \times \rangle G,$$

where $(\angle F)\ b\ a = F\ b$ and $(\rangle F)\ b\ a = F\ a$. That is, a dissection of an $(F \times G)$-structure consists either of an F-structure containing only elements of the first type paired with a G-dissection, or an F-dissection paired with a G-structure containing only elements of the second type.

As a simple example, consider the polynomial functor $L = 1 + XL$ of finite lists. Intuitively, the dissection of a list should consist of a list of b's, followed by a hole, and then a list of a's, that is,

$$\triangle L\ b\ a \cong L\ b \times L\ a.$$

Applying the rules above, we can derive

$$\begin{aligned}\triangle L\ b\ a &= \triangle(1 + XL)\ b\ a \\ &= (0 + \angle X \times \triangle L + \triangle X \times \rangle L)\ b\ a \\ &= b \times (\triangle L\ b\ a) + L\ a\end{aligned}$$

and thus $\triangle L\ b\ a \cong L\ b \times L\ a$ as expected.

7.2 Dissection via Matrices

The DFA recognizing b^*ha^* is illustrated in Fig. 7, and has transition matrix

$$\begin{bmatrix} X_b & X_h \\ 0 & X_a \end{bmatrix}.$$

There are clearly no leaf sequences taking this DFA from state 2 to state 1; leaf sequences matching b^* or a^* keep the DFA in state 1 or state 2, respectively; and leaf sequences matching b^*ha^* take the DFA from state 1 to state 2. That is, under the homomorphism induced by this DFA, the functor F maps to the matrix of bifunctors

$$\begin{bmatrix} \angle F & \mathbb{\Delta} F \\ 0 & \backslash F \end{bmatrix}.$$

Taking the product of two such matrices indeed yields

$$\begin{bmatrix} \angle F & \mathbb{\Delta} F \\ 0 & \backslash F \end{bmatrix} \begin{bmatrix} \angle G & \mathbb{\Delta} G \\ 0 & \backslash G \end{bmatrix} = \begin{bmatrix} \angle F \times \angle G & \angle F \times \mathbb{\Delta} G + \mathbb{\Delta} F \times \backslash G \\ 0 & \backslash F \times \backslash G \end{bmatrix},$$

as expected.

7.3 Divided Differences

Just as differentiation of types has an analytic analogue, dissection has an analogue as well, known as *divided difference*. Let $f : \mathbb{R} \to \mathbb{R}$ be a real-valued function, and let $b, a \in \mathbb{R}$. Then the *divided difference* of f at b and a, notated[6] $f_{b,a}$, is defined by

$$f_{b,a} = \frac{f_b - f_a}{b - a},$$

where for consistency of notation we write f_b for $f(b)$, and likewise for f_a. In the limit, as $a \to b$, this yields the usual derivative of f.

We now consider the type-theoretic analogue of $f_{b,a}$. We cannot directly interpret subtraction and division of functors. However, if we multiply both sides by $(b - a)$ and rearrange a bit, we can derive an equivalent relationship expressed in terms of only addition and multiplication, namely,

$$f_a + f_{b,a} \times b = a \times f_{b,a} + f_b.$$

This equation corresponds exactly to the isomorphism witnessed by McBride's function *right*,

$$right : F\ a + (\mathbb{\Delta} F\ b\ a, b) \to (a, \mathbb{\Delta} F\ b\ a) + F\ b$$

[6] Our notation is actually "backwards" with respect to the usual notation—what we write as $f_{b,a}$ is often written $f[a, b]$ or $[a, b]f$—in order to better align with the combinatorial intuition discussed later.

We can now explain why the letters b and a are "backwards". Intuitively, we can think of a dissection as a "snapshot" of a data structure in the midst of a traversal; values of type a are "unprocessed" and values of type b are "processed". The "current position" moves from left to right through the structure, turning a values into b values. This is exactly what is accomplished by *right*: given a structure full of unprocessed a values, or a dissected F with a focused b value, it moves the focus right by one step, either focusing on the first unprocessed a, or yielding a structure full of bs in the case that all the values have been processed.

7.4 Higher-Order Divided Differences

Higher-order divided differences, corresponding to higher derivatives, are defined by the recurrence

$$f_{x_n \ldots x_0} = \frac{f_{x_n \ldots x_1} - f_{x_{n-1} \ldots x_0}}{x_n - x_0}. \tag{9}$$

Alternatively, the higher-order divided differences of a function f can be arranged in a matrix, as, for example,

$$\triangle_{cba} f = \begin{bmatrix} f_c & f_{c,b} & f_{c,b,a} \\ 0 & f_b & f_{b,a} \\ 0 & 0 & f_a \end{bmatrix} \tag{10}$$

in such a way as to be a semiring homomorphism, that is, $\triangle_{cba}(f + g) = \triangle_{cba} f + \triangle_{cba} g$ and $\triangle_{cba}(fg) = \triangle_{cba} f \triangle_{cba} g$, and so on. Proving that this yields a definition equivalent to the recurrence (9) boils down to showing that if $f = gh$ then

$$f_{x_n \ldots x_0} = \sum_{j=0}^{n} g_{x_n \ldots x_j} h_{x_j \ldots x_0}. \tag{11}$$

Proving (11) is not entirely straightforward; in fact, we conjecture that the computational content of the proof, in the $n = 2$ case, essentially consists of (the interesting part of) the implementation of the isomorphism *right*.

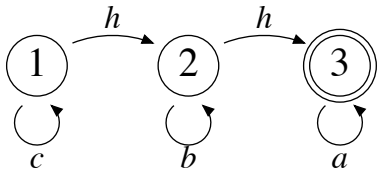

Fig. 12. A DFA for higher-order divided difference

If we now consider the DFA D in Fig. 12, we can see that (10) corresponds to the matrix $[F]_D$. More generally, a DFA consisting of a sequence of n states with

self-loops chained together by h transitions will have a transition matrix corresponding to an order-n matrix of divided differences. In general, F_{ij} will consist of $j - i$ holes interspersed among sequences of consecutive alphabet elements.

By analogy with the binary dissection case, we would expect (9) to yield an isomorphism with type

$$\triangle_{x_{n-1}\ldots x_0} F + (\triangle_{x_n \ldots x_0}, x_n) \to (x_0, \triangle_{x_n \ldots x_0} F) + \triangle_{x_n \ldots x_1} F.$$

We have not yet been able to fully make sense of this, but hope to understand it better in the future. In particular, our intuition is that this will yield a tail-recursive implementation of a structure being processed by multiple coroutines.

8 Discussion and Future Work

This paper arose out of several blog posts by both authors [12][14][13][19], although the content of this paper is neither a strict subset nor superset of the content of the blog posts. There is much remaining to be explored, in particular understanding the isomorphisms induced by higher-order divided differences, and generalizing this framework to n-ary functors and partial differentiation. It seems likely that q-derivatives can also fruitfully be seen in a similar light [17].

There are several more practical aspects to this work that remain to be explored. At a fundamental level, there would be some interesting engineering work involved in turning this into a practical library. One might also wonder to what extent it is possible to take *operations* on some polynomial functor T and automatically lift them into operations on a constrained version of T. At the very least this would require checking that the operation actually preserves the given constraints.

Some of the ideas in this paper are implicitly present in earlier work; we note in particular Duchon *et al.* [4, p. 590], who mention generating Boltzmann samplers for strings corresponding to regular expressions, also via their DFAs. It would be interesting to explore the relationship in more detail.

Acknowledgements. This work was partially supported by the National Science Foundation, under NSF 1218002, CCF-SHF Small: *Beyond Algebraic Data Types: Combinatorial Species and Mathematically-Structured Programming.*

Our sincere thanks to the anonymous reviewers, who had many helpful suggestions. Thanks also to Lukas Mai for pointing out some errors in a draft.

References

1. Michael Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. ∂ for Data: Differentiating Data Structures. *Fundam. Inform.*, 65(1-2):1–28, 2005.
2. Stephen Dolan. Fun with semirings: a functional pearl on the abuse of linear algebra. In *ACM SIGPLAN Notices*, volume 48, pages 101–110. ACM, 2013.
3. Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata.* Springer Science & Business Media, 2009.

4. Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.
5. Gérard Huet. Functional pearl: The zipper. *J. Functional Programming*, 7:7–5, 1997.
6. Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, DTIC Document, 1951.
7. Daniel J Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977.
8. Simon Marlow. Haskell 2010 Language Report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.
9. Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL*, pages 287–295, 2008.
10. Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(01):1–13, 2008.
11. Russell O'Connor. A very general method for computing shortest paths. `http://r6.ca/blog/20110808T035622Z.html`, 2011.
12. Dan Piponi. Finite Differences of Types. `http://blog.sigfpe.com/2009/09/finite-differences-of-types.html`, 2009.
13. Dan Piponi. Constraining Types with Regular Expressions. `http://blog.sigfpe.com/2010/08/constraining-types-with-regular.html`, 2010.
14. Dan Piponi. Divided Differences and the Tomography of Types. `http://blog.sigfpe.com/2010/08/divided-differences-and-tomography-of.html`, 2010.
15. Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.
16. Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode ad in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, 2008.
17. Mike Stay. Q, Jokers, and Clowns. `https://reperiendi.wordpress.com/2014/08/05/q-jokers-and-clowns/`, 2014.
18. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.
19. Brent A. Yorgey. On a problem of sigfpe. `https://byorgey.wordpress.com/2010/08/12/on-a-problem-of-sigfpe/`, 2010.