Adding Structure to Monoids

thus hopefully ending Haskell's string type confusion

Mario Blažević
Stilo International plc
blamario@yahoo.com

Abstract

This paper presents the rationale and design of *monoid-subclasses*. This Haskell library consists of a collection of type classes that generalize the interface of several common data types, most importantly those used to represent strings. We demonstrate that the mathematical theory behind *monoid-subclasses* can bring substantial practical benefits to the Haskell library ecosystem by generalizing *attoparsec*, one of the most popular Haskell parsing libraries.

Categories and Subject Descriptors D.2.13 [*Reusable Software*]: Reusable libraries,Reuse models

General Terms Algorithms, Performance, Design

Keywords Monoids; Cancellative; Generic Programming

1. Introduction

Due to the accidents of history, a newcomer to Haskell wishing to try some basic string or file manipulations immediately confronts a bewildering choice of data types to represent a string:

- String has been a part of Haskell from the beginning, and is the only string type to be baked into the language specification. For reasons unknown[1], but which can be assumed to trace back to LISP, the same specification refuses to apply any of the available type-abstraction facilities and declares String to be synonymous with a linked list of characters [Char]. The consequences of this decision on the performance of the String data type trigger a sequence of fixes.[2]
- ByteString[7] has a history nearly as long. Its predecessor PackedString was already shipping with Glasgow Haskell Compiler in 1996. In 2003 the first widely used Haskell application Darcs needed more performance for its text file manipulations, so it adapted PackedString into FastPackedString. The data type was finally extracted into a standalone library in 2005. At the same time it was renamed to ByteString in order to clarify that it represents sequences of bytes rather than of characters.
  A ByteString value is represented as a pointer into an array and thus requires much less memory than String. In addition, many ByteString operations are potentially subject to de-forestation optimization, thus avoiding the construction of the intermediate data structures.
- Lazy ByteString is an alternative implementation of the aforementioned strict ByteString with different performance characteristics, dating from 2006. The two types share the same home library, the same user interface, and much of the implementation code. They are not compatible, however, merely convertible with no loss of information.
- Text[12] was implemented in 2009. Its low-level representation largely follows ByteString, the most important difference being that it stores Unicode characters instead of bytes. This means that it can be used as a replacement for String, usually providing much better performance.
- Lazy Text is related to the strict Text described above in much the same way lazy ByteString is related to strict ByteString. Again, the two types are convertible but not compatible.

The last data type in this list in particular is superior in virtually every respect to the original String. Not only does it perform better, but it can be treated as an opaque data type. Its low-level representation and the operations' implementation could be improved or completely replaced in future with little need to change the client code.[3]

The remainder of the paper will use the capitalized String to refer to the original **type** String = [Char]. All other uses of the term "string type" will refer to any data type used to represent strings in general.

1.1 The string compatibility problem

The string type situation in Haskell is not a problem only for beginners. Even experienced Haskell developers must confront this choice every time they start a new library or application. The problem is not simply in choosing the optimal string type for the problem at hand, it is to choose the best type for interfacing with other Haskell libraries that the new code will end up using, or being used by. There are two basic answers to the interfacing problem:

1.1.1 Conversion

When developing an application or a framework, one straightforward answer that can be given is: pick one central string type,

[1] There is no mention of any decision-making on String in [13] at all.

[2] Theoretically speaking, a Haskell list need not be implemented as a linked list. In practice, GHC does an admirable job of optimizing lists and String in particular, but they still cost 20-40 bytes per character [12] and their concatenation complexity is still O(n).

[3] The low-level details are available through the Data.Text.Internal and Data.Text.Lazy.Internal modules, whose use is warned against.

Haskell '13, September 23–24, 2013, Boston, MA, USA.

ACM 978-1-4503-2383-3/13/09...$15.00.
http://dx.doi.org/10.1145/2503778.2503785

preferably a well-performing one, and stick with it. If any libraries used by the application expect a different string type, convert to and from the central representation as necessary.

This central type solution can work for an occasional library as well. In some cases there is a clear performance winner among the string types, whose advantage overrides the potential cost of conversion. For most libraries and their authors, however, compatibility with other libraries is more important than the performance gains they could wring from the choice of any particular string type.

1.1.2 Abstraction

Different libraries have incorporated different solutions to this compatibility problem. Some libraries go so far as to duplicate (or worse) much of their code in order to accommodate different string types.[4]

Another approach is to declare a type class and to move all the type-specific code into its instances. In fact, all the candidate string types are already instances of the standard type classes Eq, Ord, IsString, and Monoid. This much is already sufficient to generically implement any code that needs only create string values, compare them, or concatenate them together.

For libraries that need to examine a given string and take it apart, these standard type classes are unfortunately not sufficient. They must declare additional type classes with methods they need. For example, *Parsec* (since version 3) declares

$$\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{Stream}\ s\ m\ t \mid s \to t\ \textbf{where}$$
$$uncons :: s \to m\ (\mathsf{Maybe}\ (t, s))$$

with instances for lists, strict and lazy ByteString, and strict and lazy Text.

There are also several library packages[5] that define classes named StringLike or ListLike, either for internal use only or for common use by several other packages. The classes come with instances for different string types. This may be the clearest indication of the magnitude of the problem, and also presents the most closely related work to the one presented here. We will later compare these approaches to ours.

1.2 Overview of the rest of the paper

The remainder of the paper will present one possible way to stop the current fragmentation. In the spirit of Haskell, the solution uses type classes to abstract over all of the existing string data types. Similarly to the Monoid class that they extend, however, the new classes' theoretical foundations allow them to be instantiated by many additional types unrelated to strings. We hope that this combination of practicality and generality makes the presented solution attractive to developers of string-processing libraries in Haskell.

The following section will present the design of the *monoid-subclasses* library, beginning with its mathematical foundations and ending with some practical refinements. Then we discuss an early application of the library, analyze the results, and finish with a comparison to the related work and a look at the possible future development.

2. Design

2.1 Mathematical background

Introductory textbooks on abstract algebra tend to dwell on semigroups and monoids, and than expand to groups. While monoids are a part of the Haskell base libraries, and many wish that semigroups were as well, groups are relegated to the rarely-used libraries specific to algebra. The main reason is that there are few data types that can implement the inverse operation that groups require.

Most importantly for our purposes, no string-like data type can support the inverse operation.

Is the abstract algebra theory then useless beyond monoids? Not quite. There are some interesting algebraic structures that are more powerful than semigroups and monoids, yet more general than full groups.

2.1.1 Commutative semigroup and monoid

The most commonly recognized subclass of semigroups are *commutative semigroups*, whose binary operation $\diamond$ has the commutativity property:

$$a \diamond b = b \diamond a$$

A commutative semigroup which is also a monoid is called a commutative monoid. The corresponding Haskell class declaration would be

$$\textbf{class}\ \mathsf{Monoid}\ m \Rightarrow \mathsf{CommutativeMonoid}\ m$$

The CommutativeMonoid class has no methods of its own. Its only purpose is to constrain the type parameters of other type classes and functions.

2.1.2 Cancellative semigroup

A semigroup S with the binary operation $\diamond$ is called *left cancellative*[6] if, for any $a, b, c \in S$, $a \diamond b = a \diamond c$ implies $b = c$; one can cancel out the common factor on the left. Conversely, in a *right cancellative semigroup* $a \diamond c = b \diamond c$ always implies $a = b$. A semigroup that is both left- and right-cancellative is simply called a *cancellative semigroup*. This property of a semigroup is orthogonal to the existence of the neutral element, so we can also speak of a left- and right-cancellative monoid, or just a cancellative monoid.

Every group is a cancellative monoid: to cancel out the common factor, we can simply apply its inverse to both sides of the equation. Not every cancellative monoid is a group, however. A case of particular interest to us is that of strings or sequences of any kind, which indeed are cancellative but (as already noted) have no inverse.

The guarantee that the cancellation produces a unique result means that we can introduce a new binary operation for it. In fact, the Text type already supports this operation under the names $stripPrefix$ and $stripSuffix$. We shall adopt these names for the left- and right-cancellative monoid, respectively. For the monoids which are both commutative and cancellative, we need to introduce a new operation which we shall denote as $\not\diamond$, or </> in ASCII notation. The Haskell representation of the three cancellative monoid classes would then be:

$$\textbf{class}\ \mathsf{Monoid}\ m \Rightarrow \mathsf{LeftCancellativeMonoid}\ m$$
$$\quad \textbf{where}\ stripPrefix :: m \to m \to \mathsf{Maybe}\ m$$
$$\textbf{class}\ \mathsf{Monoid}\ m \Rightarrow \mathsf{RightCancellativeMonoid}\ m$$
$$\quad \textbf{where}\ stripSuffix :: m \to m \to \mathsf{Maybe}\ m$$
$$\textbf{class}\ (\mathsf{CommutativeMonoid}\ m,$$
$$\qquad \mathsf{LeftCancellativeMonoid}\ m,$$
$$\qquad \mathsf{RightCancellativeMonoid}\ m) \Rightarrow$$
$$\quad \mathsf{CancellativeMonoid}\ m\ \textbf{where}$$
$$\quad (\not\diamond) :: m \to m \to \mathsf{Maybe}\ m$$

[4] A cursory search reveals attoparsec, blaze-markup, csv-conduit, double-conversion, netspec, polyparse, process-extras, and reform-blaze.

[5] GroundHog, ListLike, network-fancy, StringLike, tagsoup

The three class methods satisfy the following laws:

$$\begin{aligned} stripPrefix\ a\ (a \diamond b) &= \mathsf{Just}\ b && (1)\\ stripSuffix\ b\ (a \diamond b) &= \mathsf{Just}\ a && (2)\\ (a \diamond b) \not\diamond b &= \mathsf{Just}\ a && (3)\\ stripPrefix\ a\ c = \mathsf{Just}\ b &\Rightarrow a \diamond b = c && (4)\\ stripSuffix\ b\ c = \mathsf{Just}\ a &\Rightarrow a \diamond b = c && (5)\\ a \not\diamond b = \mathsf{Just}\ c &\Rightarrow b \diamond c = c \diamond b = a && (6) \end{aligned}$$

The first three laws above state the left and right cancellativity of the monoids. The remaining laws further restrict the operations' semantics to return Nothing whenever the cancellation is impossible.

In LeftCancellativeMonoid and RightCancellativeMonoid we now have the first two theoretically well-grounded classes that all string types can implement in addition to Eq, Ord, IsString, and Monoid. Not only that, but other sequence-preserving containers like $[a]$, Seq a and Vector a are their instances as well.

The commutative CancellativeMonoid class has no bearing on string data types, but it is provided for completeness. Among all data types in the *containers* and *base* library, the only non-trivial instance of this class is Sum n.

2.1.3 Reductive semigroup

One might expect all other container monoids such as Set a to be cancellative as well, but that is not the case. The reason for this is that the monoid operation on Set, namely the set union, is non-injective because it discards all duplicate elements. That makes it impossible to reconstruct its original operands. The natural cancellation operation for the set union would be the set difference, but it does not satisfy the law (3):[6]

$$(\{a, b\} \cup \{a\}) \setminus \{a\} = \{b\} \neq \{a, b\}$$

The Product monoid has a similar problem: every multiplication is cancellative with the exception of multiplication by zero. The cancellation cannot satisfy the law (3) because $0/0$ is not defined.

We can still accommodate the numerous data types with non-injective monoid operation like Product, Set, and Map. These monoids may not be cancellative, but they are *reductive monoids*. The formal difference between a cancellative and reductive semigroup is merely in the placement of a qualifier:

$$\begin{aligned} (\forall a, b, c \in S)(a \diamond b = a \diamond c \Rightarrow b = c) &\quad \text{left cancellative}\\ (\forall b, c \in S)((\forall a \in S)(a \diamond b = a \diamond c) \Rightarrow b = c) &\quad \text{left reductive} \end{aligned} \tag{7}$$

A left-cancellative semigroup S requires the section $(a\diamond) :: S \to S$ to be injective for every $a \in S$ so it has a unique inverse. In a reductive non-cancellative semigroup the operation of cancellation does not guarantee a unique result. To reflect this fact, we could tweak the cancellation so it returns all possible results:

$\mathbf{class}$ Monoid $m \Rightarrow$ ReductiveMonoid m $\mathbf{where}$
  $(\not\diamond) :: m \to m \to [m]$

Turning back to the Set example, we would have

$$(\{a, b\} \diamond \{a\}) \not\diamond \{a\} = \{a, b\} \not\diamond \{a\} = \{\{b\}, \{a, b\}\}$$

In practice, however, having all possible cancellation results would probably not be very useful. For the purpose of cancellation, having one result is as good as having many to choose one from. Cases like $0 \not\diamond 0$ in the Product monoid can have an infinite number of solutions. For these reasons and others, the ReductiveMonoid class will keep the same type signature for the $\not\diamond$ operation as CancellativeMonoid. The only difference is that the former class will obey only a subset of the laws respected by the latter.

Every cancellative semigroup is also reductive, so the natural way to declare the two classes in Haskell is to make ReductiveMonoid a superclass of CancellativeMonoid. When we add the non-commutative classes to the mix, we end up with six classes:

$\mathbf{class}$ Monoid $m \Rightarrow$ LeftReductiveMonoid m $\mathbf{where}$
  $stripPrefix :: m \to m \to$ Maybe m
$\mathbf{class}$ Monoid $m \Rightarrow$ RightReductiveMonoid m $\mathbf{where}$
  $stripSuffix :: m \to m \to$ Maybe m
$\mathbf{class}$ (CommutativeMonoid m, LeftReductiveMonoid m,
    RightReductiveMonoid m)
  $\Rightarrow$ ReductiveMonoid m $\mathbf{where}$
  $(\not\diamond) :: m \to m \to$ Maybe m
$\mathbf{class}$ LeftReductiveMonoid m
  $\Rightarrow$ LeftCancellativeMonoid m
$\mathbf{class}$ RightReductiveMonoid m
  $\Rightarrow$ RightCancellativeMonoid m
$\mathbf{class}$ (LeftCancellativeMonoid m,
    RightCancellativeMonoid m, ReductiveMonoid m)
  $\Rightarrow$ CancellativeMonoid m

The cancellative classes do not add any new methods, they merely require the inherited methods to respect the full set of cancellation laws listed above. The reductive classes require only the laws (4), (5), and (6) to be followed. In principle the law (7) and its equivalents should be respected as well, but unfortunately they cannot be expressed in Haskell.

2.1.4 GCD-semigroup

A semigroup $(S, \diamond)$ is called a *GCD-semigroup*[9] if for its any two elements $a, b \in S$ it also contains their greatest common divisor $gcd(a, b)$ satisfying the following properties:

$$\begin{gathered} (\exists a_1 \in S)\ a = a_1 \diamond gcd(a, b)\\ (\exists b_1 \in S)\ b = b_1 \diamond gcd(a, b)\\ a = a_2 \diamond c \wedge b = b_2 \diamond c \Rightarrow (\exists c_1 \in S)\ gcd(a, b) = c_1 \diamond c \end{gathered}$$

Since the gcd operation subsumes the test of divisibility, every GCD-semigroup is necessarily reductive. The Haskell type class GCDMonoid should therefore be declared a subclass of ReductiveMonoid. As we had before, there will be a separate class for commutative monoids and two classes for non-commutative ones:

$\mathbf{class}$ (ReductiveMonoid m, LeftGCDMonoid m,
    RightGCDMonoid m)
  $\Rightarrow$ GCDMonoid m $\mathbf{where}$
  $gcd :: m \to m \to m$

$\mathbf{class}$ LeftReductiveMonoid m
  $\Rightarrow$ LeftGCDMonoid m $\mathbf{where}$
  $commonPrefix :: m \to m \to m$
  $stripCommonPrefix :: m \to m \to (m, m, m)$

$\mathbf{class}$ RightReductiveMonoid m
  $\Rightarrow$ RightGCDMonoid m $\mathbf{where}$
  $commonSuffix :: m \to m \to m$
  $stripCommonSuffix :: m \to m \to (m, m, m)$

Informally, the function $commonPrefix$ returns the greatest prefix common to its two arguments, while $commonSuffix$ returns

[6] A multiset data type would not have this problem, since its union operation is injective.

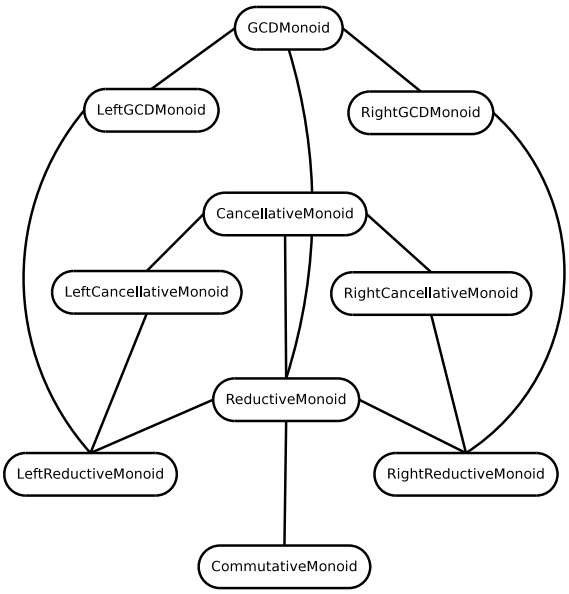

Figure 1. Reductive, Cancellative, and GCD classes

their greatest common suffix. For the commutative GCDMonoid instances the two methods are synonymous with the method gcd.

The classes must satisfy the appropriate laws, of which we present only those that apply to the LeftGCDMonoid class; the others are equivalent:

$$\begin{aligned} stripCommonPrefix\ a\ b = (p, a', b') \\ \Downarrow \\ commonPrefix\ a\ b = p \\ \wedge\ a = p \diamond a' \\ \wedge\ b = p \diamond b' \end{aligned}$$

$$\begin{aligned} stripPrefix\ p\ a = \mathsf{Just}\ a' \\ \wedge\ stripPrefix\ p\ b = \mathsf{Just}\ b' \\ \Downarrow \\ stripCommonPrefix\ a\ b = (p, a', b') \end{aligned}$$

In case the instance of LeftReductiveMonoid is also an instance of LeftCancellativeMonoid, it must additionally satisfy the following law:

$$commonPrefix\ (p \diamond a)\ (p \diamond b) = p \diamond (commonPrefix\ a\ b)$$

All the classes presented so far are related, and their relationship can be visualized as the diagram in Figure 1.

2.1.5 Factorial semigroup

A non-commutative semigroup S is called *factorial* if every element $a \in S$ has a unique factorization $a = a_1 \diamond a_2 \diamond \ldots \diamond a_n$ into a sequence of *atomic (i.e., prime)* elements a_i that cannot be factorized themselves. The same term can be applied to a commutative semigroup, except in that case we require the factorizations to be unique only up to reordering. If the semigroup has a unit element (*i.e.*, if it is a monoid), we only consider the factorizations that do not include it. The unit element is necessarily prime itself.

This Haskell type class can be used to model the concept of the factorial monoid:

$$\begin{aligned} &\textbf{class}\ \mathsf{Monoid}\ m \Rightarrow \mathsf{FactorialMonoid}\ m\ \textbf{where} \\ &\quad factors :: m \to [m] \\ &\quad factors = unfoldr\ splitPrimePrefix \\ &\quad splitPrimePrefix :: m \to \mathsf{Maybe}\ (m, m) \end{aligned}$$

provided it satisfies the following laws:

$$\begin{aligned} mconcat \circ factors &= id \\ factors\ \varepsilon &= [] \\ all\ (\lambda p \to factors\ p = [p]) &\circ factors \end{aligned}$$

The first law states that the factorization is proper, so if we concatenate all the factors we get back the original monoid. The second law clarifies the proper factorization of the unit value ε, or `mempty`. Finally, the third law demands every factor to be prime.

It's worth noting one law that is explicitly not required from the FactorialMonoid class instances:

$$factors\ (a \diamond b) = factors\ a \diamond factors\ b$$

If it held, this law would make factorizations a global property. It would also disallow the FactorialMonoid instances of many data types like that of Product, again due to multiplication by zero.

Most monoidal data structures in use are factorial. All string types can be trivially made instances of the FactorialMonoid class, and so can Sum, Product, Set, Map, Vector, and others. For example:

- The $factors$ of a String or Text value are its one-character substrings:

  $$factors\ \texttt{"abc"} \equiv [\texttt{"a"}, \texttt{"b"}, \texttt{"c"}]$$

- A Product is factorized into the list of its prime factors:

  $$\begin{aligned} &factors\ (\mathsf{Product}\ 20) \\ &\equiv [\mathsf{Product}\ 2, \mathsf{Product}\ 2, \mathsf{Product}\ 5] \end{aligned}$$

- The $factors$ of a Set are its singleton subsets:

  $$\begin{aligned} &factors\ (fromList\ [1,2,3]) \\ &\equiv [singleton\ 1, singleton\ 2, singleton\ 3] \end{aligned}$$

- In general, any monoidal collection type that is both Foldable and Applicative can be made an instance of FactorialMonoid:

  $$factors = foldMap\ (\lambda s \to [pure\ s])$$

Given the $factors$ function, we can generalize many of the higher-order functions that operate on lists. In fact, some of them could be said to generalize even upon their already-generalized counterparts from the Foldable and Traversable classes:

$$\begin{aligned} &foldl :: \mathsf{FactorialMonoid}\ m \Rightarrow \\ &\qquad (a \to m \to a) \to a \to m \to a \\ &foldr :: \mathsf{FactorialMonoid}\ m \Rightarrow \\ &\qquad (m \to a \to a) \to a \to m \to a \\ &length :: \mathsf{FactorialMonoid}\ m \Rightarrow m \to \mathsf{Int} \\ &foldMap :: (\mathsf{FactorialMonoid}\ m, \mathsf{Monoid}\ n) \Rightarrow \\ &\qquad (m \to n) \to m \to n \\ &take :: \mathsf{FactorialMonoid}\ m \Rightarrow \mathsf{Int} \to m \to m \\ &takeWhile :: \mathsf{FactorialMonoid}\ m \Rightarrow \\ &\qquad (m \to \mathsf{Bool}) \to m \to m \\ &reverse :: \mathsf{FactorialMonoid}\ m \Rightarrow m \to m \end{aligned}$$

Wherever a Foldable or Traversable class method in its signature has an element a of the structure $f\ a$, the corresponding FactorialMonoid method has a prime factor of the monoid m.

2.2 Practical considerations

The two remaining classes that come with *monoid-subclasses* are not as well theoretically founded as those presented so far. They were motivated purely by practical concerns. That does not mean they are any *more* practical, only that I failed to find any existing algebraic concepts to base them on.

2.2.1 MonoidNull

The simplest possible inspecting extension to the Monoid class would be

$$\begin{aligned}&\textbf{class}\ \mathsf{Monoid}\ m \Rightarrow \mathsf{MonoidNull}\ m\ \textbf{where}\\ &\quad \mathit{null} :: \mathsf{Monoid}\ m \Rightarrow m \to \mathsf{Bool}\end{aligned}$$

There is really not much to say about it. The class comes with a single law:

$$\mathit{null}\ a \Leftrightarrow a = \varepsilon$$

We can easily tell for any factorial monoid whether it is the unit by checking if its *factors* are an empty list. It follows that MonoidNull should be a superclass of FactorialMonoid.

2.2.2 Textual Monoid

Class TextualMonoid brings together all functionality required of a string type. Its declaration is

$$\begin{aligned}&\textbf{class}\ (\mathsf{IsString}\ t, \mathsf{LeftReductiveMonoid}\ t,\\ &\qquad \mathsf{LeftGCDMonoid}\ t, \mathsf{FactorialMonoid}\ t)\\ &\quad \Rightarrow \mathsf{TextualMonoid}\ t\ \textbf{where}\\ &\quad \mathit{splitCharacterPrefix} :: t \to \mathsf{Maybe}\ (\mathsf{Char}, t)\end{aligned}$$

The new $\mathit{splitCharacterPrefix}$ method is closely related to $\mathit{splitPrimePrefix}$ from the FactorialMonoid class, the difference being that the former attempts to coerce the prime prefix of the monoid into a character. If the monoid starts with a character the method must always extract it:

$$\mathit{splitCharacterPrefix}\ (\mathit{fromString}\ [c] \diamond t) = \mathsf{Just}\ (c, t)$$

Another law of TextualMonoid restricts each character to make a single prime factor:

$$\begin{aligned}\mathit{splitCharacterPrefix}\ m &= \mathsf{Just}\ (c, t)\\ &\Downarrow\\ (\exists p)(\mathit{splitCharacterPrefix}\ p &= \mathsf{Just}\ (c, \varepsilon)\\ \wedge\ \mathit{splitPrimePrefix}\ m &= \mathsf{Just}\ (p, t))\end{aligned}$$

It is worth noting that not every prime factor needs to correspond to a character:

$$\begin{aligned}\mathit{splitPrimePrefix}\ m &= \mathsf{Just}\ (p, t)\\ \not\Rightarrow\ \mathit{splitCharacterPrefix}\ m &= \mathsf{Just}\ (c, t)\end{aligned}$$

The reason we don't enforce this is to accommodate data types which contain characters interleaved with other things. One obvious use for these non-character factors would be the invalid encodings that don't correspond to any character, but they could also represent markup or other structural parts of the data type.

To summarize, TextualMonoid together with its super-classes provides the following functionality:

- creation of values from lists of characters (and with GHC's OverloadedStrings extension, directly from string literals as well) inherited from the IsString class,
- Eq and Ord comparisons,
- matching and stripping of prefixes, inherited from LeftReductiveMonoid,
- finding the longest common prefix inherited from LeftGCDMonoid,
- splitting a value into a sequence of atomic values with at most one character each, and
- extracting the character a value starts with.

Notably missing from the superclass constraints are RightReductiveMonoid and RightGCDMonoid. They were left out because the existing string types do not provide efficient instances of these classes, and because the suffix operations are less common than the prefix operations in practice.

Even though all of the existing string types are instances of LeftCancellativeMonoid, the TextualMonoid class requires only the LeftReductiveMonoid superclass in order to allow more possible instances. Full cancellativity is often not required in practice, and the constraint can be added explicitly when necessary.

2.2.3 Performance requirements

Method signatures and law guarantees are all necessary, but not always sufficient. The client of a type class quite often in practice wants to know not only that a particular instance conforms with the type class interface, but that it does so efficiently. The only support we can offer here is to informally declare various worst-case complexity bounds for the class methods. At the moment the only such informal guarantees in *monoid-subclasses* are that

- the complexity of the null method instance should be constant, while
- the complexity of the method instances $\mathit{stripPrefix}\ m\ n$ and $\mathit{stripSuffix}\ m\ n$ should be no worse than $O(\#m \cdot \log \#n)$.

The latter requirement eliminates the possibility of a RightReductiveMonoid instance for lists – a guarantee on one side is a constraint on the other.

2.2.4 Additional class methods

All of the class presentations above concentrated on the fundamental class methods, because they suffice as the basis for defining all functions of interest. In the case of FactorialMonoid, for example, functions like length, take, $\mathit{takeWhile}$, $\mathit{reverse}$, and others could be implemented using the method $\mathit{factors}$ alone.

While this minimalistic approach is elegant, the performance of such functions would be far worse than that of their specialized counterparts. These functions have therefore been made into additional class methods, and most instances of FactorialMonoid map them directly onto their specialized versions. As an example, the actual definition of the FactorialMonoid instance for strict Text type is given in Figure 2.

All additional methods have been given a default implementation based on the fundamental methods, so new instances of classes can be easily defined.

2.2.5 Language standard compatibility

All type class declarations laid out above are fully compatible with the Haskell 98 and Haskell 2010 standards. Most class instance definitions are compatible with these standards as well. The only exception is the **instance** TextualMonoid String, which requires the FlexibleInstances language extension provided by GHC. The extension is not controversial, and even it could be made unnecessary by introducing an IsChar class. We have judged the present form of the instance declaration less intrusive.

```
instance FactorialMonoid Text.Text where
  factors = Text.chunksOf 1
  primePrefix = Text.take 1
  primeSuffix x = if Text.null x
                  then Text.empty
                  else Text.singleton (Text.last x)
  splitPrimePrefix =
    fmap (first Text.singleton c) ∘ Text.uncons
  splitPrimeSuffix x =
    if Text.null x
    then Nothing
    else Just (Text.init x, Text.singleton (Text.last x))
  foldl f = Text.foldl f'
    where f' a char = f a (Text.singleton char)
  foldl' f = Text.foldl' f'
    where f' a char = f a (Text.singleton char)
  foldr f = Text.foldr f'
    where f' char a = f (Text.singleton char) a
  length = Text.length
  span f = Text.span (f ∘ Text.singleton)
  break f = Text.break (f ∘ Text.singleton)
  dropWhile f = Text.dropWhile (f ∘ Text.singleton)
  takeWhile f = Text.takeWhile (f ∘ Text.singleton)
  split f = Text.split f'
    where f' = f ∘ Text.singleton
  splitAt = Text.splitAt
  drop = Text.drop
  take = Text.take
  reverse = Text.reverse
```

Figure 2. Full instance of the FactorialMonoid class for Text

3. Parser Combinator Libraries

Most of the classes presented in the previous section were originally a part of the *incremental-parser*[2] library, before they gained a library package of their own. The aforementioned parser package still depends on these classes and uses them to implement its generic parsing combinators.

The *incremental-parser* package is not readily comparable with other parsing combinator libraries, so we won't discuss it here. We have instead turned to the more popular *attoparsec*[17] library and extended it with the generic versions of its existing combinators. Another benefit of this approach is that something quite similar has been done before with *nanoparsec*, and that gives us another point of comparison.

3.1 Attoparsec

The *attoparsec* parser combinator library is a quick and simple alternative to *Parsec*[16]. One simplification from *Parsec 3* that is of particular interest here is that *attoparsec* is monomorphic in its input type. The only input type it originally supported was strict ByteString, but later versions have added new monomorphic implementations of the same operators that accept only Text inputs instead of ByteString. This deficiency makes *attoparsec* a prime candidate for generalization.

3.2 Nanoparsec and ListLike

This idea has been followed before. The *nanoparsec*[19] library is a polymorphic re-implementation of *attoparsec* that accepts any input instantiating the ListLike class, imported from the library package of the same name. This class has two type parameters, which makes it incompatible with the Haskell 2010 language standard. The two parameters represent the full list-like type and the type of the items it contains:

```
class (FoldableLL full item, Monoid full) ⇒
      ListLike full item | full → item where
  singleton :: item → full
  head :: full → item
  tail :: full → full
  null :: full → Bool
```

3.3 Monoid-attoparsec

The *monoid-attoparsec* library [3] is another generalization of *attoparsec* based on the monoid subclasses introduced in the previous section. The parsing combinators it exports are copied without any modification from *attoparsec*. Its primitive parser functions have been re-implemented, and they have several differences with *nanoparsec* worth noting:

- Where a primitive parser function in *attoparsec* has the hard-coded token type in its signature, the corresponding function in *nanoparsec* has the collection item type parameter, and in *monoid-attoparsec* the collection type. This collection is always a singleton (a.k.a. a prime monoid) in such cases. Take the example of the parser function *satisfy*:

```
  -- attoparsec ByteString
satisfy :: (Word8 → Bool) → Parser Word8
  -- attoparsec Text
satisfy :: (Char → Bool) → Parser Char
  -- nanoparsec
satisfy :: ListLike δ ϵ ⇒ (ϵ → Bool) → Parser δ ϵ
  -- monoid-attoparsec
satisfy :: FactorialMonoid t ⇒ (t → Bool) → Parser t t
```

- Since the most common use of parser libraries is on text, an additional allowance has been made for the TextualMonoid inputs. For each *attoparsec* function that works with input tokens, there are two functions in *monoid-attoparsec*: one that expects a generic FactorialMonoid input as explained above, and another specialized to TextualMonoid and Char. To take the example of *satisfy* again, the specialized version is

```
satisfyChar :: TextualMonoid t ⇒
               (Char → Bool) → Parser t Char
```

- While most primitive parser functions of *monoid-attoparsec* work with FactorialMonoid inputs, that is no rule. The function *endOfInput*, for example, needs its input class only to instantiate MonoidNull. Function *string* requires the LeftGCDMonoid class as well, and several functions as explained above work only with the TextualMonoid inputs.
- Both *nanoparsec* and *monoid-attoparsec* accept as inputs all string types, as well as several additional data structures such as Seq which are not typically used for parsing. These data types instantiate both ListLike and the monoid subclasses. There are also several data types that are not ListLike in any sense, but are instances of LeftGCDMonoid and FactorialMonoid and therefore qualify as inputs to *monoid-attoparsec*. The following parser, if it can be called that, takes a Product Integer input and checks if the input is a Hamming number, *i.e.*, if it has the form $2^i \cdot 3^j \cdot 5^k$:

```
hamming :: Parser (Product Integer) ()
hamming = skipWhile pred *> endOfInput
  where pred (Product n) = elem n [2, 3, 5]
```

A slightly more practical parser with a non-standard input type would be:

```
tokens :: (Ord t, TextualMonoid t) ⇒ Parser (t, Set t) [t]
tokens = token `sepBy` takeWhile1 space
  where token = do (tok, _) ← takeTill space
                   string (ε, Set.singleton tok)
                   return tok
        space (t, _) = t ≡ " "
```

This parser's input is a pair of a TextualMonoid and a set of tokens of the same type. It consumes from the first component of the pair as many space-separated tokens as it can find in the set. No token is allowed more than once, because the *string* invocation above consumes it from the set.

4. Results

In order to check if the *monoid-attoparsec* library can be used in practice with a satisfactory performance, we are going to take an existing parser relying on *attoparsec* and to modify it so it uses *monoid-attoparsec* instead. The parser to modify will be *attoparsec-csv*[5], a relatively simple CSV parser that takes Text input and returns [[Text]] output: a list of CSV records, where each record is a list of fields.

4.1 CSV parser rewrite

We are applying only the minimal modifications required to switch the underlying parser combinator library. Since *attoparsec-csv* uses various character-parsing functions for Text inputs, we can generalize the input type only up to the TextualMonoid class.

- the library imports are changed;
- the type signatures of the parser functions must be generalized; for example, Parser [T.Text] becomes TextualMonoid $t \Rightarrow$ Parser t [t];
- T.*append* and T.*concat* function calls specific to Text are replaced by more general *mappend* and *mconcat*;
- all *takeWhile* calls are replaced by *takeCharsWhile*; and finally,
- *return* `'"'` is replaced with a more general *return* `"\""` that can return any IsString instance.

The original source code of *attoparsec-csv* is available at [5], the generalized version is listed in Appendix A.

4.2 Benchmarks

We compare the performance of the original *attoparsec-csv* library against our modified version using the sample CSV input with the *OpenFlights Airports Database*[7].

Parser combinator library	Input type	Time
attoparsec	Text	86.9 ms
nanoparsec	Text	> 1 min
nanoparsec	CharString	83.3 ms
monoid-attoparsec	ByteStringUTF8	93.5 ms
monoid-attoparsec	Text	103.7 ms
monoid-attoparsec	Lazy.Text	178.4 ms
monoid-attoparsec	Seq Char	197.2 ms
monoid-attoparsec	String	286.8 ms
monoid-attoparsec	Vector Char	545.3 ms

[7] From http://openflights.org/data.html

The timing was performed using the Criterion benchmarking library[18] and GHC 7.6.2. The entire input file of 636KiB in UTF-8 encoding was loaded in memory and converted to the desired input type before the measurement.

In most cases the entire input was parsed at once, using the function *parseCSV*. However, three input types took an exceedingly long time to parse with this method. The cause of the slowdown turned out to be the monoid append operation $\diamond$ that *attoparsec* invokes on the input after a local failure, where the right-hand argument of the operator tends to be empty. For these three data types, Lazy.Text, String, and Vector Char, the $\diamond$ operation requires time proportional to the length of its left-hand argument. This behaviour is a part of the core definitions of *attoparsec* and as such was inherited by *nanoparsec* and *monoid-attoparsec*.

The adopted workaround was to apply the attoparsec CSV parser incrementally to small chunks of the input, using the function *parseChunkedCSV*. The optimal chunk length was empirically determined to be approximately 24 characters in case of String and Vector Char, and 256 characters in case of Lazy.Text. The times shown in the table were measured using these chunk lengths.

The *nanoparsec* measurements were performed using a modified version of *attoparsec-csv* that imports *nanoparsec* instead of *attoparsec*, together with other necessary modifications. The source code of this modified module is provided in Appendix B.

The terrible performance of *nanoparsec* with the Text input is partly due to the $\diamond$ operation noted above, but even more to its regular use of the *length* method of ListLike. This method call is inherited from *attoparsec* as a shortcut within the input prefix check. The shortcut pays off with the ByteString inputs, but the complexity of *length* in the Text instance of ListLike is $O(n)$, which translates to $O(n^2)$ for the entire parse. The *monoid-attoparsec* adaptation of the input prefix check eschews the use of the *length* method in favour of *stripCommonPrefix* which has a more predictable performance.

In the interest of fairness, a performance measurement of *nanoparsec* with the CharString input type has also been included for comparison. This data type is a **newtype** wrapper around ByteString and provides a ListLike instance with the $O(1)$ complexity for its *length* method, so it does not suffer from the same problem. Note however that we had to provide an invalid instance of IsString CharString which can accept only a small subset of the Unicode character set. This is the reason *monoid-subclasses* does not provide such a simple wrapper around ByteString to masquerade as a TextualMonoid. The only ByteString wrapper that the package currently provides is ByteStringUTF8, capable of representing any Unicode character.

4.3 Conclusions

We can see from the result table that the best performing string types are the strict Text and ByteStringUTF8. The lazy variant of Text has proven much worse in this case, but mostly because of the heavy use of the $\diamond$ operation by *attoparsec* as explained above.

If we compare the performance of strict Text input under *monoid-attoparsec* against *attoparsec* which is specialized and highly optimized for this type, the cost of the generalization is around 20%. This performance cost is not trivial, but it must be weighed against the added ability to choose the optimal data type for the task at hand. It should also be noted that *monoid-attoparsec* is only a conservative extension of *attoparsec*, leaving its internals completely unmodified. More intrusive optimizations are certainly possible.

5. Related and future work

The motivation for *monoid-subclasses* originally appeared in the context of streaming Haskell coroutines in producer-consumer relationships[4]. A stream exchanged between such coroutines naturally forms a monoid, and the consumer often needs to examine the structure of the received stream chunks. The *iteratee* library[14, 15] thus depends on ListLike for the very same reason. We intend to revisit this application area.

In order to cover as many data types as possible, *monoid-subclasses* concentrates on the monoidal data type alone and abstracts over the the type of the item it contains. This choice does incur some performance penalty, but we believe it is worth the added abstraction and simplicity. As a support for this position, we can only offer virtually all programming languages that have been designed and used primarily for string manipulation, such as SNOBOL[8], Icon[11], AWK[1], or Perl[20]. These languages have only a single string type and no first-class character type: when needed, a character is represented as a single-character string. This uniformity of expression has proven to be their strength.

Another benefit of ignoring the item type is that it leaves only the single type parameter for each class, which means that the class declarations do not require any extension to the standard Haskell 2010. The most prominent previous attempt to abstract over the multiple string types, the ListLike library[10], uses multi-parameter type classes instead. This may be the reason it failed to gain much traction. The classes presented by this paper are also more generic and easier to implement, which gives them a wider applicability.

The class methods provided by *monoid-subclasses* have proven to be quite sufficient for implementing efficient parsing combinator libraries, but it remains to be seen if they are capable of supporting other application domains. One area where they are currently lacking any support is I/O, though it is questionable how practical such an extension would be. Another obvious extension would be a new type class interface for searchable non-commutative monoids:

$$
\begin{aligned}
&\textbf{class}\ (\mathsf{LeftReductiveMonoid},\\
&\qquad \mathsf{RightReductiveMonoid})\\
&\quad \Rightarrow \mathsf{SearchableMonoid}\ \textbf{where}\\
&\quad \mathit{isInfix} :: t \to t \to \mathsf{Bool}\\
&\quad \mathit{stripLeftmost} :: t \to t \to \mathsf{Maybe}\ (t, t)\\
&\quad \mathit{stripRightmost} :: t \to t \to \mathsf{Maybe}\ (t, t)
\end{aligned}
$$

A more complex extension would be required to support efficient scanning and regular expression matching on monoid values.

If the Semigroup class ever assumes its rightful place as a superclass of Monoid in the base Haskell libraries, the *monoid-subclasses* package will require an implementation overhaul, and would benefit from a name change. Each of the existing classes would gain a new non-monoid superclass, but their interfaces would not need to change.

We believe that we have proven that the library has the technical merits to fix Haskell's string type problems. To actually succeed at this task, three things need to happen.

Firstly, the range of the available data types could expand. The authors of libraries that instantiate the Monoid class should consider providing the instances for its subclasses as well. This part is relatively easy for the authors, and should be helped by the fact that the *monoid-subclasses* library is lightweight and standards-compliant.

The second part of the puzzle are generic client libraries. The *monoid-attoparsec* library is a start, but we need other low-level generic libraries.

Finally, the authors of the higher-lever libraries that currently rely on *attoparsec* need to compare the benefits of generalizing their input type against the costs of the code update and the performance penalty. This call will have to be made by each author and for each library individually.

References

[1] Alfred V Aho, Brian W Kernighan, and Peter J Weinberger. Awk–a pattern scanning and processing language. *Software: Practice and Experience*, 9(4):267–279, 1979.

[2] Mario Blažević. incremental-parser: Generic parser library capable of providing partial results from partial input. Hackage. `http://hackage.haskell.org/package/incremental-parser`.

[3] Mario Blažević. monoid-attoparsec: an input-generic fork of attoparsec. Bitbucket. `https://bitbucket.org/blamario/monoid-attoparsec`.

[4] Mario Blažević. Coroutine pipelines. *The Monad. Reader Issue 19: Parallelism and Concurrency*, page 29, 2011.

[5] Robin Bate Boerop. attoparsec-csv: A parser for csv files that uses attoparsec. Hackage. `http://hackage.haskell.org/package/attoparsec-csv`.

[6] Alfred H Clifford and Gordon Bamford Preston. *The algebraic theory of semigroups, volume 1*. American Mathematical Soc., 1961.

[7] Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Practical Aspects of Declarative Languages*, pages 50–64. Springer, 2007.

[8] David J Farber, Ralph E Griswold, and Ivan P Polonsky. Snobol, a string manipulation language. *Journal of the ACM (JACM)*, 11(1):21–30, 1964.

[9] Robert Gilmer and Tom Parker. Divisibility properties in semigroup rings. *The Michigan Mathematical Journal*, 21(1):65–86, 1974.

[10] John Goerzen. Listlike: Generic support for list-like structures. Hackage. `http://hackage.haskell.org/package/ListLike`.

[11] Ralph E Griswold and Madge T Griswold. *The Icon programming language*, volume 28. Prentice-Hall Englewood Cliffs NJ, 1983.

[12] Thomas Harper. Stream fusion on Haskell unicode strings. In *Implementation and Application of Functional Languages*, pages 125–140. Springer, 2011.

[13] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[14] Oleg Kiselyov. iteratee: Iteratee-based i/o. Hackage. `http://hackage.haskell.org/package/iteratee`.

[15] Oleg Kiselyov. Iteratees. In *Proceedings of the 11th international conference on Functional and Logic Programming*, pages 166–181. Springer-Verlag, 2012.

[16] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical report, Department of Computer Science, Utrecht University, 2001.

[17] Bryan O'Sullivan. attoparsec: Fast combinator parsing for bytestrings. Hackage. `http://hackage.haskell.org/package/attoparsec`.

[18] Bryan O'Sullivan. criterion: Robust, reliable performance measurement and analysis. Hackage. `http://hackage.haskell.org/package/criterion`.

[19] Maciej Piechotka. nanoparsec: An implementation of attoparsec-like parser around list-like. Hackage. `http://hackage.haskell.org/package/nanoparsec`.

[20] Larry Wall et al. The Perl programming language, 1994.

A. attoparsec-csv generalized using monoid-attoparsec

```
{-# Language FlexibleContexts, OverloadedStrings #-}
module Text.CSV.Monoid (parseCSV, parseChunkedCSV)
where

import Prelude hiding (splitAt)
import Control.Applicative ((<$>), (<|>), (<*>),
                            (<*), (*>), many)
import Control.Monad (void)
import Data.Monoid (mappend, mconcat)
import Data.Monoid.Textual (TextualMonoid)
import Data.Monoid.Factorial (splitAt)
import Data.Attoparsec.Monoid

lineEnd :: TextualMonoid t ⇒ Parser t ()
lineEnd =
  void (char '\n') <|> void (string "\r\n")
   <|> void (char '\r')
   <?> "end of line"

unquotedField :: TextualMonoid t ⇒ Parser t t
unquotedField =
  takeCharsWhile (∉ ",\n\r\"")
   <?> "unquoted field"

insideQuotes :: TextualMonoid t ⇒ Parser t t
insideQuotes =
  mappend <$> takeCharsWhile (≢ '"')
          <*> (mconcat
                <$> many (mappend <$> dquotes
                                  <*> insideQuotes))
   <?> "inside of double quotes"
  where
    dquotes =  string "\"\"" ≫ return "\""
               <?> "paired double quotes"

quotedField :: TextualMonoid t ⇒ Parser t t
quotedField =
  char '"' *> insideQuotes <* char '"'
   <?> "quoted field"

field :: TextualMonoid t ⇒ Parser t t
field =
  quotedField <|> unquotedField
   <?> "field"

record :: TextualMonoid t ⇒ Parser t [t]
record =
  field `sepBy1` char ','
   <?> "record"

file :: TextualMonoid t ⇒ Parser t [[t]]
file =
  (:) <$> record
      <*> manyTill (lineEnd *> record)
                   (endOfInput <|> lineEnd *> endOfInput)
   <?> "file"

parseCSV :: TextualMonoid t ⇒ t → Either String [[t]]
parseCSV = parseOnly file

parseChunkedCSV :: (Show t, TextualMonoid t) ⇒
  Int → t → Either String [[t]]
parseChunkedCSV chunkLength s =
  continue s (Partial (parse file))
  where continue s (Done _ r) = Right r
        continue s (Fail rest contexts msg) =
          Left $ show (rest, contexts, msg)
        continue s (Partial f) =
          let (prefix, suffix) = splitAt chunkLength s
          in continue suffix (f prefix)
```

B. attoparsec-csv generalized using nanoparsec

```
{-# Language FlexibleContexts, OverloadedStrings #-}
module Text.NanoCSV (parseCSV) where

import Prelude hiding (concat, elem, takeWhile)
import Control.Applicative ((<$>), (<|>), (<*>), (<*), (*>))
import Control.Monad (void)
import Data.String (IsString)
import Data.ListLike (ListLike, append, concat, cons, empty)
import Data.Nanoparsec

lineEnd :: (Eq t, IsString t, ListLike t Char) ⇒ Parser t ()
lineEnd =
  void (elem '\n') <|> void (string "\r\n")
                   <|> void (elem '\r')
   <?> "end of line"

unquotedField :: ListLike t Char ⇒ Parser t t
unquotedField =
  takeWhile (∉ ",\n\r\"")
   <?> "unquoted field"

insideQuotes :: (Eq t, IsString t, ListLike t Char) ⇒ Parser t t
insideQuotes =
  append <$> takeWhile (≢ '"')
         <*> (concat <$> many (cons <$> dquotes
                                    <*> insideQuotes))
   <?> "inside of double quotes"
  where
    dquotes =
      string "\"\"" ≫ return '"'
       <?> "paired double quotes"

quotedField :: (Eq t, IsString t, ListLike t Char) ⇒ Parser t t
quotedField =
  elem '"' *> insideQuotes <* elem '"'
   <?> "quoted field"

field :: (Eq t, IsString t, ListLike t Char) ⇒ Parser t t
field =
  quotedField <|> unquotedField
   <?> "field"

record :: (Eq t, IsString t, ListLike t Char) ⇒ Parser t [t]
record =
  field `sepBy1` elem ','
   <?> "record"

file :: (Eq t, IsString t, ListLike t Char) ⇒ Parser t [[t]]
file =
  (:) <$> record
      <*> manyTill (lineEnd *> record)
                   (endOfInput <|> lineEnd *> endOfInput)
   <?> "file"

parseCSV :: (Eq t, IsString t, ListLike t Char) ⇒
  t → Either String [[t]]
parseCSV = eitherResult ∘ (`feed` empty) ∘ parse file
```