Project Report for Generic Programming

Generic Automata

Vali Georgescul, Rui S Barbosa, Julian Verdurmen

Universiteit Utrecht

3rd July 2009

Abstract

We explain two generalizations of classical automata, both with a coalgebraic flavour: F-automata and Tree automata. The former abstracts the shape of the transition function whilst the latter generalizes the input language from sequences of symbols to arbitrary tree-like structures of symbols.

We show how to implement the two extensions in HASKELL using techniques and libraries from type-generic programming and discuss their suitability.

Along the way, we also address some problems related to generic programming that we encountered during this project.

1 Introduction

Automata theory deals with systems composed by a (possibly infinite) set of states and a transition function specifying how to change to new states, possibly depending on input values and/or producing some sort of output. There are several different classes of automata used for different purposes, from recognizing languages to representing processes.

Every time such a class is introduced (or implemented in some programming language), a lot of operations are specifically defined for that class. This usually includes a run function, product, bisimilarity/minimization, composition, etc. These specific definitions are normally quite similar in flavour from case to case, suggesting that there is a common ground.

The idea behind this project is to abstract common patterns to several classes of transition systems. We aim to develop a HASKELL library using type-generic programming techniques which implements these operations once and for all for several different kinds of automata. In our opinion, such a library has some advantages, both theoretical and practical:

- by abstracting, we get a clearer understanding of the nature of such operations
- for the same reason, the definitions are probably simpler and focused on the essential
- a generic library can be used consistently in many instantiations
- it is more likely that it can be used when we are dealing with a specific kind of automata
- it is also easier to maintain and extend

Starting from the classical notion of a complete deterministic automata, we shall consider two generalizations separately:

- abstract the shape of the transition to an arbitrary F-coalgebra (Section 2). This includes classical deterministic (Id), partial ($Maybe$) and non-deterministic (powerset) automata as well as other automata enriched with some extra structure.

- abstract the shape of the input language (Section 3). Classical automata recognize sequences of symbols (list-like languages). This extension deals with tree automata, whose recognized words have an arbitrary tree structure.

It will become clear that both extensions share a similar coalgebraic flavour. Thus, in the end, we discuss what can be done to integrate the two implementations (Section 4).

A main goal of this project was to experiment applying generic programming techniques to build a library.

- Throughout the presentation, we try to give our account on how to achieve this;
- We summarize our opinion on the suitability of the existing GP libraries and the problems encountered during this project (Section 5.1)
- While implementing the library, we also came across some GP problems whose scope is not restricted to this automata library. Particularly, we present an extension to the *Regular* library which allows the definition of container types (Section 2.3.1), we show a way to define a powerset functor in HASKELL and include it in our universe (Section 2.3.3) and we discuss the possibility of defining generic monad instances for some classes of functors (Section 2.6).

2 Abstracting the Shape: F-Automata

2.1 Motivating Examples

2.1.1 Classical Automata

We shall start by considering some classical examples of automata. In this presentation, we shall mainly consider input (reactive) automata, where the transitions are determined by symbols given as input (actions from the environment). However, the approach can also be extended to output (active) automata or IO automata (Section 2.5.3)

This kind of automata have a set of states S (this set is not necessarily finite in general) and a set A of input symbols (or actions), which we require to be finite. The most important ingredient is a transition function. This function defines the way a transition system reacts to each input action, by changing its state. Let us look at its shape for some well-known classes of automata:

(complete) deterministic	(CDA)	$A \to S \to S$
partial deterministic	(PDA)	$A \to S \to \mathbf{1} + S$
non-deterministic	(NDA)	$A \to S \to \mathcal{P}_{fin}\ S$

In the first case, the automata will be always able to move to a new state whatever the input symbol and the current state are. The second class allows partiality of the transition function, thus the automata may not be able to move to a new state in some cases. Note that, in HASKELL $\mathbf{1} + S$ would probably be written as $Maybe\ S$. In non deterministic automata, the result of the transition function might be an arbitrary (finite) set of states, meaning that the automata might continue through any of those states (or all in parallel, depending on how you look at it).

2.1.2 List-shaped automata

We now consider a not so classical example. Suppose we want to express the possibility of multiple next states (as in NDA) but where these possible transitions have some order (the informal meaning might be some kind of preference). We shall also allow repeated end states. This ordered multiset structure is clearly captured by a list. The transition function would be as follows:

$$\text{list shaped} \quad \text{(LSA)} \quad A \to S \to S^\star$$

2.2 The General Pattern: a Coalgebra

Observing the four transition functions presented so far, we can definitely notice a common similarity: each possible input symbol determines a transition from each state to some structured collection of states. We will abstract from the actual structure of this collection and represent it as a functor F. A transition function is then a F-coalgebra on the set of states for each input symbol.

$$F\text{-automata} \quad A \to S \to F\ S$$

In the remainder of this section, we will describe some common operations which can be defined generically for any functor in a certain universe and also explain how we can write these functions generically in HASKELL.

2.3 Universe of Functors

Before proceeding to the implementation of generic automata operations, we need to consider how to represent them in a generic way. As we said, the shape of these automata is determined by a functor F. Hence, we need to define a universe in which we can represent all the functors of interest.

2.3.1 Regular Datatypes

The basic universe of functors could be constructed from sums, products, units and constants. As we also want to express recursive data types, we shall consider a fix operation of pattern functors (type functors) of regular data types. This constitutes the universe defined in the Regular library.

However, this library has a drawback: it has no notion of type parameters. In the library, we can represent types defined as the smallest solution to the equation $T \cong GT$ for some regular functor G (called the pattern functor of type T), that is, $T = \mu G$. Of course, this type T can itself be of the form $T'\ a$, thus including a type parameter. This means that the type $[a]$ (lists of a) can be given the following pattern functor

$$\textbf{type instance}\ PF\ [a] = \mathbf{1} + K\ a \times Id$$

The problem, however, lies on the fact that the representation has no knowledge of where are the occurrences of a in the type structure: they are represented as any other constant type. This knowledge is crucial if one wants, for example, to define a generic map function (mapping the a's) or a $crush$ over all those a's.

This motivated us to consider a simple variation on this library that solves this problem. We will restrict our attention to types with only one parameter, as these are sufficient for the purpose of this presentation. The rôle of the pattern functor G (described above) will now be played by a pattern bifunctor B. A bifunctor is a functor with two arguments: it satisfies the functor laws when fixing any of the arguments. Our type T will be defined by an equation of the form $TA \cong B(A, TA)$, or equivalently, $TA = \mu_X B(A, X)$. So, the first argument of the pattern bifunctor will represent the type parameter whilst the second will be used to point to recursive occurrences.

We start by providing the data types with which we can construct pattern bi-functors in our universe. The following definitions are quite straightforward and similar to their counterparts in the Regular library.

$$\begin{aligned}
&\textbf{data}\ \mathbf{1} && a\ r = \mathbf{1}\\
&\textbf{data}\ K\ k && a\ r = K\ k\\
&\textbf{data}\ (f + g) && a\ r = L\ (f\ a\ r) \mid R\ (g\ a\ r)\\
&\textbf{data}\ (f \times g) && a\ r = f\ a\ r \times g\ a\ r
\end{aligned}$$

Now, instead of indicating only the recursive occurrences (Id in Regular), we have to consider pointing to each of the arguments. We chose names resembling their intended usage in pattern functors.

$$\mathbf{data}\ Rec\ a\ r = Rec\ r$$
$$\mathbf{data}\ Par\ a\ r = Par\ a$$

We shall define the following class for bi-functors, whose method is analogous to $fmap$ (the action of the functor on arrows):

$$\begin{aligned}&\mathbf{class}\ BiFunctor\ (f :: * \to * \to *)\ \mathbf{where}\\ &\quad bimap :: (a \to b) \to (c \to d) \to f\ a\ c \to f\ b\ d\end{aligned}$$

and give instances for each construction defined above

$$\begin{aligned}&\mathbf{instance}\ BiFunctor\ \mathbf{1}\ \mathbf{where}\\ &\quad bimap\ \_\ \_\ \mathbf{1} = \mathbf{1}\\ &\mathbf{instance}\ BiFunctor\ (K\ k)\ \mathbf{where}\\ &\quad bimap\ \_\ \_\ (K\ k) = K\ k\\ &\mathbf{instance}\ (BiFunctor\ f, BiFunctor\ g) \Rightarrow BiFunctor\ (f + g)\ \mathbf{where}\\ &\quad bimap\ f\ g\ (L\ x) = L\ (bimap\ f\ g\ x)\\ &\quad bimap\ f\ g\ (R\ y) = R\ (bimap\ f\ g\ y)\\ &\mathbf{instance}\ (BiFunctor\ f, BiFunctor\ g) \Rightarrow BiFunctor\ (f \times g)\ \mathbf{where}\\ &\quad bimap\ f\ g\ (x \times y) = bimap\ f\ g\ x \times bimap\ f\ g\ y\\ &\mathbf{instance}\ BiFunctor\ Rec\ \mathbf{where}\\ &\quad bimap\ f\ g\ (Rec\ r) = Rec\ (g\ r)\\ &\mathbf{instance}\ BiFunctor\ Par\ \mathbf{where}\\ &\quad bimap\ f\ g\ (Par\ a) = Par\ (f\ a)\end{aligned}$$

To complete our generic representation, we need a way to convert an arbitrary (yet suitable) data type to this universe. This is achieved by the following ParRegular class.

$$\begin{aligned}&\mathbf{class}\ BiFunctor\ (PBF\ t) \Rightarrow ParRegular\ t\ \mathbf{where}\\ &\quad \mathbf{type}\ PBF\ t :: * \to * \to *\\ &\quad from :: t\ a \to (PBF\ t)\ a\ (t\ a)\\ &\quad to :: (PBF\ t)\ a\ (t\ a) \to t\ a\end{aligned}$$

As an example, the type of lists would now have the following pattern functor

$$\begin{aligned}&\mathbf{instance}\ ParRegular\ [\,]\ \mathbf{where}\\ &\quad \mathbf{type}\ PBF\ [\,] = \mathbf{1} + Par \times Rec\end{aligned}$$

So that we can use such a parametrized regular type as a functor, we need the following instance

$$\begin{aligned}&\mathbf{instance}\ (Regular\ f) \Rightarrow Functor\ f\ \mathbf{where}\\ &\quad fmap\ f = to \cdot bimap\ f\ (fmap\ f) \cdot from\end{aligned}$$

Remark. The reader might have noticed that the EMGM or LIGD approaches also provide a way to refer to type parameters, though not to recursive occurrences. In fact, these would have sufficed for some of the implemented operations we present. However, some side ideas (for example, section 2.6) as well as tree automata need to work with pattern functors. Thus, the approach seems to be more appropriate due to its flexibility. It also seems more natural to work on a library dealing directly with functors.

Also, a existent way of dealing with type parameters in generic programming is the language extension PolyP. We chose not to use it as it is possible to achieve the same using a library instead.

2.3.2 (Finite) Covariant Exponentials

A useful extension to this universe are covariant exponentials $-^K$. At first, we might be tempted to encode those in HASKELL as follows

$$\mathbf{data}\ (f \wedge k)\ a\ r = Exp\ (k \to f\ a\ r)$$

However, for most definitions, we will have to range through all the domain of this function, which means it has to be finite. As we cannot enforce K to be finite (without restricting it to a certain class of types, which is not too bad yet perhaps less natural to use), we will consider the following definition.

$$\mathbf{data}\ (f \wedge k)\ a\ r = Exp\ \{ unExp :: k \mapsto f\ a\ r \}$$

We now need to define the type $(\mapsto)$ for finite functions, which will be basically a map (from Data.Map). The purpose of including exponentials in this presentation is also to present some abbreviations to deal with these functions in a natural way. This will be useful throughout the presentation: note that there is a finite function between the set of input symbols and transitions (cf. the representation in Section 2.4 and the discussion in Section 2.5.3).

$$\begin{aligned}
&\mathbf{type}\ (\mapsto) = Map\\
&\mathbf{infixl}\ 9\ \backslash\$\\
&\mathbf{infixr}\ 9\ \backslash.\\
&(\backslash\$) \quad :: (Ord\ a) \Rightarrow (a \mapsto b) \to a \to b\\
&f\ \backslash\$\ x \quad = guardFromJust\ \texttt{"finite function not defined"}\ (M.lookup\ x\ f)\\
&\quad \mathbf{where}\ guardFromJust\ _ \quad (Just\ x) = x\\
&\qquad\qquad guardFromJust\ err\ Nothing = error\ err\\
&(\backslash.) \quad :: (Ord\ a) \Rightarrow (b \to c) \to (a \mapsto b) \to (a \mapsto c)\\
&g\ \backslash.\ f \quad = M.map\ g\ f\\
&domain :: (Ord\ a) \Rightarrow (a \mapsto b) \to [a]\\
&domain = M.keys\\
&range \quad :: (Ord\ a) \Rightarrow (a \mapsto b) \to [b]\\
&range \quad = M.elems
\end{aligned}$$

The $BiFunctor$ instance can now be given.

$$\begin{aligned}
&\mathbf{instance}\ BiFunctor\ f \Rightarrow BiFunctor\ (f \wedge k)\ \mathbf{where}\\
&\quad bimap\ f\ g\ (Exp\ m) = Exp\ (bimap\ f\ g\ \backslash.\ m)
\end{aligned}$$

2.3.3 The Powerset Functor

One other thing we would like to have is a way to represent the powerset functor. There are two difficulties here. First, sets cannot be declared as functors because they need the stored types to be in class Ord (or at least Eq, which is conceptually the correct notion). Second, they are not regular datatypes, so they do not have a pattern functor. In order to use power sets when defining pattern functors (to express sets, lists of sets or rose trees with unordered children), we need the following datatype

$$\mathbf{data}\ \mathcal{P}_{fin}\ f\ a\ r = PS\ (Set\ (f\ a\ r))$$

and an instance $BiFunctor\ f \Rightarrow BiFunctor\ (\mathcal{P}_{fin}\ f)$. To use them outside a fixed point type (to express sets of lists for example), we refer to Section 2.3.4.

As we already noticed, the desired instance is impossible to give due to the equality instances requirement (we will ignore that there is a Ord requirement for Data.Set, because conceptually

we could do it only with Eq). The workaround might seem a bit awkward. We need to consider new $Functor$ and $BiFunctor$ classes with this Eq requirement.

$$\begin{aligned}
&\textbf{class } EFunctor\ f \textbf{ where}\\
&efmap\ \ :: (Eq\ a, Eq\ b) \Rightarrow (a \to b) \to f\ a \to f\ b\\
&\textbf{class } EBiFunctor\ f \textbf{ where}\\
&ebimap :: (Eq\ a, Eq\ b, Eq\ c, Eq\ d) \Rightarrow (a \to b) \to (c \to d) \to f\ a\ c \to f\ b\ d
\end{aligned}$$

Clearly, any $Functor$ is also a $EFunctor$.

$$\begin{aligned}
&\textbf{instance } (Functor\ f) \Rightarrow EFunctor\ f \textbf{ where}\\
&\quad efmap\ \ = fmap\\
&\textbf{instance } (BiFunctor\ f) \Rightarrow EBiFunctor\ f \textbf{ where}\\
&\quad ebimap = bimap
\end{aligned}$$

Now, we can give the bifunctor instance for a $\mathcal{P}_{fin}$. The additional constraint $Eq2$ in this case is needed to lift equality to the functor.

$$\begin{aligned}
&\textbf{instance } (BiFunctor\ f, Eq2\ f) \Rightarrow EBiFunctor\ (\mathcal{P}_{fin}\ f) \textbf{ where}\\
&\quad ebimap\ f\ g\ (PS\ xs) = PS\ (Set.map\ (bimap\ f\ g)\ xs)\\
&\textbf{class } Eq2\ (f :: * \to * \to *) \textbf{ where}\\
&\quad eq2 :: (Eq\ a, Eq\ r) \Rightarrow f\ a\ r \to f\ a\ r \to Bool\\
&\textbf{instance } (Eq2\ f, Eq\ a, Eq\ r) \Rightarrow Eq\ (f\ a\ r) \textbf{ where}\\
&\quad (\equiv) = eq2
\end{aligned}$$

For simplicity, from now on we will continue to use the normal $Functor$, $BiFunctor$ and $ParRegular$ classes, but bearing in mind that this can be changed in order to accomodate power sets.

2.3.4 Composition of Functors

The $ParRegular$ class only allows regular datatypes along with the possibility of using power sets to construct their pattern functor. This allows for example to represent the type of rose trees with unordered and not repeated children:

$$\begin{aligned}
&\textbf{data } SetRose\ a = Node\ a\ (Set\ SetRose)\\
&\textbf{instance } ParRegular\ SetRose \textbf{ where}\\
&\quad \textbf{type } PBF\ SetRose = Par \times \mathcal{P}_{fin}\ Rec
\end{aligned}$$

Of course, this also includes the possibility of representing non-recursive data types, such as $Maybe$ or Set

$$\begin{aligned}
&\textbf{data } Maybe\ x = Nothing \mid Just\ x\\
&\textbf{instance } ParRegular\ Maybe \textbf{ where}\\
&\quad \textbf{type } PBF\ Maybe = \mathbf{1} + Par\\
&\textbf{instance } ParRegual\ Set \textbf{ where}\\
&\quad \textbf{type } PBF\ Set = \mathcal{P}_{fin}\ Par
\end{aligned}$$

The problem comes when mixing (composing) both kinds of types. For example, the types $ML\ a = Maybe\ [a]$ or $SL\ a = Set\ [a]$ cannot be given directly an instance of class $ParRegular$. The way to express them in our universe is as composition of two (extended regular, $ParRegular$) functors. This composition types are not only useful to express basic transition shapes but will also appear more generally when defining the product of automata and other operations (Section 2.5.3).

The definition of composition and the $Functor$ instance are pretty straightforward. Note that wrapping composition is necessary, to avoid partial application of type synonyms.

```
data (f · g) x = Comp {unComp :: f (g x)}
instance (Functor f, Functor g) ⇒ Functor (f · g) where
  fmap f = Comp · fmap (fmap f) · unComp
```

2.3.5 Generic Functions

Our full universe is then constituted by regular parametrized types (using powersets if we consider the *EBiFunctor* class) and their composition. We already defined a generic function for this universe: *fmap*. Note that we needed the definitions of *bimap* for every constructor of pattern functors, a way to lift these definitions to a *Functor* instance for any *ParRegular* and a instance for composition of two types which are already of a *Functor* instance. Other generic functions are defined in the same way. As an example, we go through the definition of *crush*, a function which collects every occurrence of the type parameter and combines them with a monoid. (As we discuss in the end of this section, this function differs slightly from the usual *crush*, or more appropriately *crushr*, that can be found in most GP libraries). We start by considering the following crusher structure: a monoid and a way to inject values of a type a in its the underlying set.

```
data Crusher a r = Crusher {plus :: r → r → r
                           , zero :: r
                           , inj  :: a → r
                           }
```

We consider a class *BCrush* defining how to use this structure at the pattern bifunctor level

```
class BCrush f where
  bcrush :: Crusher a r → f a r → r
```

and give the corresponding instances

```
instance BCrush 1 where
  bcrush m _       = zero m
instance BCrush (K k) where
  bcrush m _       = zero m
instance (BCrush f, BCrush g) ⇒ BCrush (f + g) where
  bcrush m (L x)   = bcrush m x
  bcrush m (R y)   = bcrush m y
instance (BCrush f, BCrush g) ⇒ BCrush (f × g) where
  bcrush m (x × y) = (plus m) (bcrush m x) (bcrush m y)
instance (Ord k, BCrush f) ⇒ BCrush (f ∧ k) where
  bcrush m (Exp g) = foldr (λx r → (plus m) (bcrush m x) r) (zero m) (range g)
instance BCrush f ⇒ BCrush (P_fin f) where
  bcrush m (PS x)  = S.fold (λx r → (plus m) (bcrush m x) r) (zero m) x
instance BCrush Par where
  bcrush m (Par x) = (inj m) x
instance BCrush Rec where
  bcrush _ (Rec x) = x
```

Note that what we have just defined is an algebra of the functor $f\ a$ (the bifunctor f with the first argument fixed). The way to lift this *crush* function to a *ParRegular* data type is by a catamorphism of this algebra. So, first we need the *fold* function:

```
type Algebra f x = f x → x
fold :: (ParRegular t) ⇒ Algebra (PBF t a) r → t a → r
fold alg = alg · bimap id (fold alg) · from
```

Now, we define a *Crush* class (for functors) and give the instances for both *ParRegular* types (using *fold*) and composition of crushable functors.

```
class Crush f where
  crush :: Crusher a b → f a → b
instance (ParRegular t, BCrush (PBF t)) ⇒ Crush t where
  crush m = fold (bcrush m)
instance (Crush f, Crush g) ⇒ Crush (f · g) where
crush m = crush m {inj = crush m} · unComp
```

We think that the universe presented here is not completely satisfactory. Somehow the power is a bit restricted outside of pattern functors and having these two levels does not look completely natural. Note also that it is not be possible (to our knowledge) to define the *crushr* function in this universe: the composition instance would present a problem. For these reasons, it might be a good idea to reconsider this universe we have sketched.

2.4 Representing F-Automata

2.4.1 Datatypes

The coalgebraic pattern of the transition function is expressed very simply in *Haskell*. The following type represents a transition system:

```
type TS F s = s → F s
```

A labelled transition system (where the transition is determined by an input symbol) is represented as follows:

```
type LTS F a s = a ↦ TS F s
```

Note that the set of symbols is required to be finite.

In the remainder of this section, we show to specify certain classes of automata and give the definition of two concrete automata. The purpose is to illustrate the intended use of the library and to equip us with some examples to better understand the generic definition of the operations.

2.4.2 Encoding Specific Classes of Automata

The following type synonyms represent the classes of automata discussed in Section 2.1.

```
type CDA a s = LTS Id      a s
type PDA a s = LTS Maybe a s
type NDA a s = LTS Set     a s
type LSA  a s = LTS []     a s
```

where

```
newtype Id x = Id {unId :: x}
```

To be able to use the generic operations, we would still need to declare the four functors above (*Id*, *Maybe*, *Set* and []) as part of our universe. We already showed how to do that for lists, so we use *Maybe* as an example.

```
instance ParRegular Maybe where
  type PBF Maybe = 1 + Par
  from Nothing     = L 1
  from (Just x)    = R (Par x)
  to   (L 1)       = Nothing
  to   (R (Par x)) = Just x
```

Figure 1: Example Partial Deterministic Automaton

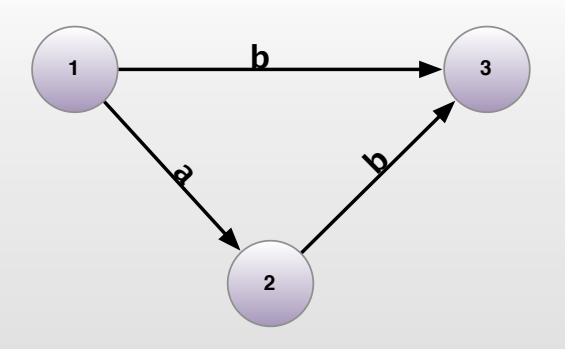
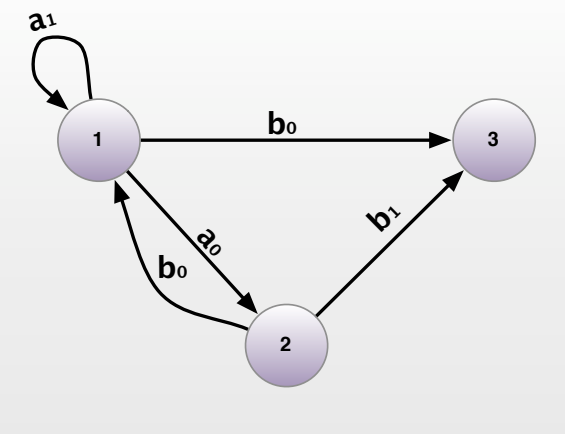


Figure 2: Example List-Shaped Automaton



2.4.3 Representing Specific Automata

Below is the encoding of the two automata in Figures 1 and 2. The former is a partial deterministic automata (from each state, there is at most one outgoing transition with the same label) while the latter is a list-shaped automata (in diagram 2, the order of the edges from a certain state by a certain symbol is indicated by the subscripts in those symbols).

```
pda1 :: PDA Char Int
pda1 = M.fromList [('a', funcFromList [(1, Just 2   )
                                      ,(2, Nothing)
                                      ,(3, Nothing)
                                      ])
                  ,('b', funcFromList [(1, Just 3   )
                                      ,(2, Just 3   )
                                      ,(3, Nothing)
                                      ])
                  ]
lsa1 :: LFA Char Int
lsa1 = M.fromList  [('a', funcFromList [(1, [2,1]    )
                                       ,(2, []       )
                                       ,(3, []       )
                                       ])
                   ,('b', funcFromList [(1, [3]      )
```

```
                                  , (2, [1, 3]    )
                                  , (3, [ ]       )
                                  ])
                ]
```

2.5 Operations

We shall now consider automata related operations which can be generically defined. We present two of them in more detail (run and bisimilarity) and briefly discuss some other.

2.5.1 Run

The first generic function we consider is run. This function gets a list of input symbols (a word) and runs the automata starting starting from one state. A interesting question is what should be the return type for a generic automata. We define three versions of this function, each reflecting one of the following possible approaches:

- forget the structure, reducing to sets
- keep the structure along the whole run
- flatten the structure built along successive steps, using the monad operations (when defined)

Below are (partial) signatures for these functions.

$$
\begin{array}{ll}
runForgetful & :: (..., Functor\ F, Crush\ F) \Rightarrow LTS\ a\ F\ s \rightarrow [a] \rightarrow s \rightarrow Set\ s \\
runPreserving & :: (..., Functor\ F) \Rightarrow LTS\ a\ F\ s \rightarrow [a] \rightarrow s \rightarrow Star\ F\ s \\
runInMonad & :: (..., Functor\ F, Monad\ F) \Rightarrow LTS\ a\ F\ s \rightarrow [a] \rightarrow s \rightarrow F\ s
\end{array}
$$

where

$$\mathbf{data}\ Star\ F\ x = End\ x \mid Step\ (F\ (Star\ F\ x))$$

corresponds to a finite power of the functor F, allowing us to keep the structure of successive transitions.

To better understand the intuition behind these approaches, let us consider the case of list-shaped automata with a transition function $\delta : S \rightarrow S^\star$ If we read two symbols, starting from state s_0, we first go to a list $\delta(s_0)$ and then apply the transition function once again to each of the elements of $\delta(s_0)$. If we keep all the structure of the run, we would get a list of lists of states and, if yet another symbol was read, we would have a list of lists of lists of states. That is the approach of $runPreserving$, where as result we get an inhabitant of a type isomorphic to an arbitrary number of nested lists. The $runForgetful$ approach is absolutely opposite. In this case, we care only about possible final states (and not about the structure which might be associated). Thus, the idea is to collect all the elements of the nested lists in a set. The last approach is somehow in between, although it can only be defined for functors which have a monadic structure, which is the case of lists. In this case, the nested lists would be flattened to one list: note that this flattening (in this case, $concat$) is the multiplication operation ($join$) of the list monad.

The definitions for all the approaches share the following common pattern.

$$
\begin{array}{l}
runPatt :: (..., Functor\ f) \Rightarrow (s \rightarrow res, f\ res \rightarrow res) \rightarrow LTS\ f\ a\ s \rightarrow [a] \rightarrow s \rightarrow res \\
runPatt\ (\eta, \mu)\ \delta\ [\,] \quad\quad = \eta \\
runPatt\ (\eta, \mu)\ \delta\ (a : as) = \mu \cdot fmap\ (runInMonad\ \delta\ as) \cdot (\delta\ \backslash\$\ a)
\end{array}
$$

Then we need only instantiate the η and μ operations. For sets, we have

$$
\begin{aligned}
&runForgetful = runPatt\ (S.singleton, setjoin \cdot toSet)\\
&\quad \textbf{where}\ toSet :: (..., Crush\ f) \Rightarrow f\ a \to Set\ a\\
&\qquad\quad toSet = crush\ Crusher\ \{plus = S.union, zero = S.empty, inj = S.singleton\}\\
&\qquad\quad setjoin :: (Ord\ a) \Rightarrow Set\ (Set\ a) \to Set\ a\\
&\qquad\quad setjoin = S.fold\ S.union\ S.empty
\end{aligned}
$$

Note that we use the generic combinator $crush$ (Section 2.3.5) to reduce a value of an arbitrary (container) datatype to the set of all the contained values.

The second approach is quite simple: as we want to preserve all the structure, the η and μ functions should be id. We need however to pass tag the values so that we are able to typecheck the definition. We do this using the constructors of datatype $Star$.

$$runPreserving = runPatt\ (End, Step)$$

The monadic version is also obvious (the η and μ are the usual η and μ from a monad):

$$runInMonad = runPatt\ (return, join)$$

Considering the automaton of figure 2, we can see the results for reading the word "ab".

```
> runForgetful lsa1 "ab" 1
fromList [1,3]
> runPreserving lsa1 "ab" 1
Step [Step [End 1,End 3],Step [End 3]]
> runInMonad lsa1 "ab" 1
[1,3,3]
```

2.5.2 Bisimilarity

As another example, we will define a function which determines whether two states are bisimilar. Being bisimilar means that the behaviour of the transition system starting from each of these states is indistinguishable. A useful application of this knowledge is minimization: two bisimilar states can be reduced to only one state.

We will now formalize this intuition and try to characterize it in a generic way. Note that bisimilarity has to be an equivalence relation on the set of states and, furthermore, a congruence with respect to the transition function (the transition function is what determines the behaviour, so this condition expresses indistinguishability). Let us consider some classes of automata as examples. The usual notion of bisimilarity corresponds to NDA, with a transition function $\delta : A \to S \to \mathcal{P}\ S$. In such a transition system, the bisimilarity relation is the largest relation $\sim$ satisfying the folowing condition:

$$
\begin{aligned}
p \sim q \quad \text{iff} \quad \forall a \in A.\ & (\forall p' \in \delta(a,p).\ \exists q' \in \delta(a,q).\ p' \sim q')\\
& \qquad \wedge\\
& (\forall q' \in \delta(a,q).\ \exists p' \in \delta(a,p).\ p' \sim q')
\end{aligned}
$$

In sum, the resulting sets of states (transiting from p and from q) taken modulo the bisimilarity relation have to be equal under bisimilarity.

Restricting this to the case of CDA ($\delta : A \to S \to S$), we can say that $p \sim q$ if and only if $\forall a \in A.\ \delta(a,p) \sim \delta(a,q)$. (Again, we are talking about the largest relation satisfying this condition).

Let us now consider an example with some more structure: list-shaped automata (Section 2.1.2). In this case, the same definition does not completely correspond to our intuition (actually, it does if all we care about is the behaviour as given by $runForgetful$). We have to take into account

the extra structure of the resulting states. Thus, two states are only bisimilar if, for each symbol read, they move to lists of states of the same size and where the elements in the same position are themselves bisimilar. This is exactly the pattern captured by the HASKELL function $zipWith$. This time, we have the condition

$$p \sim q \text{ iff } \forall a \in A.\ \bigwedge zipWith(\sim)(\delta\ a\ p)(\delta\ a\ q)$$

In general, this will be quite similar. Thus, we will need a generic $zipWith$-like function.

$$fZipWith :: (..., Zip\ f) \Rightarrow (a \to b \to c) \to f\ a \to f\ b \to f\ c$$

Actually, we will need this function to be safe (not throw errors on unmatching structures), thus returning a $Maybe$

$$fZipWith :: (..., Zip\ f) \Rightarrow (a \to b \to c) \to f\ a \to f\ b \to Maybe\ (f\ c)$$

Moreover, in the definition of bisimlation, we will keep track of a stack with all the recursive calls we are in. This stack is a list of current assumptions. Each time we visit a pair of nodes, we recursively try to check bisimilarity on the matching structures obtained after transitions. When doing this, we assume that our original pair is bisimilar. We keep track of this stack of "states in the current path" in a $Reader$ monad. So we need yet another variation of zipWith:

$$fSafeZipWithM :: (..., Zip\ f, Monad\ m) \Rightarrow (a \to b \to m\ c) \to f\ a \to f\ b \to m\ (Maybe\ (f\ c))$$

This function is defined in our generic universe as explained in Section 2.3.5.

Using this, we can write the actual $bisimilar$ function as explained above.

```
bisimilar :: (Eq s, Ord a, Zip f) ⇒ LTS f a s → s → s → Bool
bisimilar δ p q = runReader (bisim δ p q) []


bisim :: (Eq s, Ord a, Zip f) ⇒ LTS f a s → s → s → Reader [(s, s)] Bool
bisim δ p q = do stack ← ask
                 if p ≡ q ∨ (p, q) ∈ stack ∨ (q, p) ∈ stack
                   then return True
                   else liftM and $ mapM (bisimBy δ p q) (alphabet δ)
bisimBy :: (Eq s, Ord a, Zip f) ⇒ LTS f a s → s → s → a → Reader [(s, s)] Bool
bisimBy δ p q a = let p' = (δ \$ a) p
                      q' = (δ \$ a) q
                  in  local ((p, q):) $
                        liftM (maybe False and) (fSafeZipWithM (bisim δ) p' q')
```

2.5.3 Other Operations

In the same spirit, there are a lot more automata operations which can be defined generically on the functor F.

Bisimilarity Relation / Minimization The function above determines whether two states are bisimilar. Also useful is a function that calculates the whole bisimilarity relation. This works by refining the equivalence classes starting from the identity relation (the smallest possible relation satisfying the bisimulation conditions). The knowledge of this relation can be used to minimize an automaton, by quotienting its set of states.

Product This is also a very common automata operation. It is used, for example, to construct an automaton that recognizes the intersection or union of the languages ot two other automata. The set of states of the product is the Cartesian product of the sets of states of the original

automata. The transition function has to somehow express that both automata are run at the same time.

Let us consider two automata, with transition functions

$$\delta_1 : S_1 \to F_1\ S_1 \quad \text{and} \quad \delta_2 : S_2 \to F_2\ S_2$$

The product automaton will have a transition function with shape

$$S_1 \times S_2 \to G(S_1 \times S_2)$$

We will try to explain what this G can be. Starting from a state in $S_1 \times S_2$, we can apply the function $\delta_1 \times \delta_2$ (apply the corresponding transition function to each component) and go to $F_1\ S_1 \times F_2\ S_2$. From here, we use the following natural transformations (known as left and right strengths of a functor F):

$$\tau_l^F : F \times K \to F \cdot (Id \times K) \qquad \tau_r^F : K \times F \to F \cdot (K \times Id)$$

Putting this toghether, we defined a left and a right version of the product, which differ by the order in which the strenghts are applied:

$$S_1 \times S_s \xrightarrow{\delta_1 \times \delta_2} F_1\ S_1 \times F_2\ S_2 \xrightarrow{\tau_l^{F_1}} F_1\ (S_1 \times F_2\ S_2) \xrightarrow{\tau_r^{F_2}} F_1\ (F_2\ (S_1 \times S_2)) = F_1 \cdot F_2\ (S_1 \times S_2)$$

$$S_1 \times S_s \xrightarrow{\delta_1 \times \delta_2} F_1\ S_1 \times F_2\ S_2 \xrightarrow{\tau_r^{F_2}} F_2\ (F_1\ S_1 \times S_2) \xrightarrow{\tau_l^{F_1}} F_2\ (F_1\ (S_1 \times S_2)) = F_2 \cdot F_1\ (S_1 \times S_2)$$

Furthermore, if $F_1 = F_2$ and this functor is a monad, we can further apply the monad multiplication ($join$) and get a resulting automaton of the same shape as the original ones.

Note that the functorial composition that we included in our universe (Section 2.3.4) is used in the return type of the (non-monadic) product functions.

IO Automata and Composition The automata thus far considered are input automata, where the transitions are determined by a input sequence of symbols. Although we did not yet fully implemented this, it would be possible to consider input/output automata, which could be represented as:

$$\textbf{type}\ IOTS\ F\ a\ b = a \mapsto s \to F\ (s, b)$$

Special cases of input-only or output-only transition systems would be obtaining by taking b or a to be unit, respectively.

Perhaps, a good approach would be to include the input and output in the functor itself, as an exponential to a on the left or a product by b on the right. In this case, special functions would require the functor to pattern match to a certain form, depending on whether they need input, output or both. We would for example notice that the bisimilarity function we presented would not need the input condition and most likely its definition would be simplified.

Acceptance Conditions When dealing with automata recognizing finite words, we basically need to examine the result of the run determined by a certain word. Several different acceptance conditions can be implemented in this way. This may require more or less information about the runs. A typical example is to check if at least one of the reached states (which are in the set returned by $runForgetful$) is part of a special set of final states. But, having more structure, we can certainly define less common conditions by predicates on the structure of the run.

In the case of automata recognizing infinite words, we cannot define a function to determine whether a given word is recognized. It is common however to check for emptiness of the recognized language, with respect to some acceptance condition (eg: Buchi). This typically involves checking for some sort of cycles. This can be done in a way similar (actually simpler) to the $bisimilar$ function, searching the whole transition space and keeping a stack of the current path of visited states.

Interleaving We believe that some other operations common when using automata to represent processes can be given a generic definition. A good candidate seems to be the interleaving of two processes, although we have not yet implemented such a function.

2.6 Generic Monad Multiplication

Both in one of the definitions of the run function and of the product of automata, we make use of the monadic structure of the functor that characterizes the transition. The main ingredient of a monad is the multiplication (or $join$), which is a natural transformation of type $F \cdot F \to F$, corresponding to some sort of flattening operation. An interesting question is to what extent can such an operation be defined generically for a certain functor. This would lessen the burden of having to declare monad instances at least for certain classes of functors. This does not appear however to be a simple question.

We found that the more natural class of candidates is that of types which have information only in leaves, so that their pattern functor is of the form $B(A, X) = A + G(X)$ where G cannot refer to A. This is the case of leaf binary trees (where $G(X) = X^2$) and the $Maybe$ data type (where $G\ (X) = \mathbf{1}$). For this class it is quite simple to define a flattening operation. We take the information stored in the leaf and leave it in the same place but unwrapped from a leaf.

This class does not include other natural candidates such as lists or binary trees. It seems also possible to define a generic instance for types of this kind, which have a empty (unit) constructor. However, the definition is a lot more asymmetrical as it is based on some kind of concatenation operation.

3 Abstracting the Language: Tree Automata

The second generalization we consider has to do with the kind of input that drives an automaton. In the classes of automata we have been considering, transitions are determined by a finite set of symbols and a run of the automaton is determined by a sequence (a list) of those symbols. The idea behind this generalization is to lift the restriction that these symbols are provided in sequence: they can be disposed in an arbitrary tree-like structure (in accordance to an arity attributed to each symbol).

Therefore, we move from recognizing string languages to recognizing tree languages. There is also a theory relative to tree automata which generalizes that of regular languages, thus a lot of operations (such as union, intersection, etc) are suitable to be defined generically.

3.1 A First Example

To motivate this section, we will present a concrete example of a tree automaton.

Let us consider the language of integer and boolean expressions represented by the following HASKELL datatype:

```
data Expr = Zero
          | Suc Expr
          | Plus Expr Expr
          | False
          | True
          | And Expr Expr
          | Not Expr
          | Eq Expr Expr
          | IfThenElse Expr Expr Expr
```

The idea is to build a automaton that recognizes exactly the expressions which represent a valid natural number.

Table 1: Symbols of the Language of Expressions and their Arities

Symbol	Arity
0	0
$S-$	1
$-+-$	2
F	0
T	0
$-\wedge-$	2
$\neg-$	1
$-=-$	2
$if-then-else-$	3

Table 2: Transitions of the Example Tree Automaton

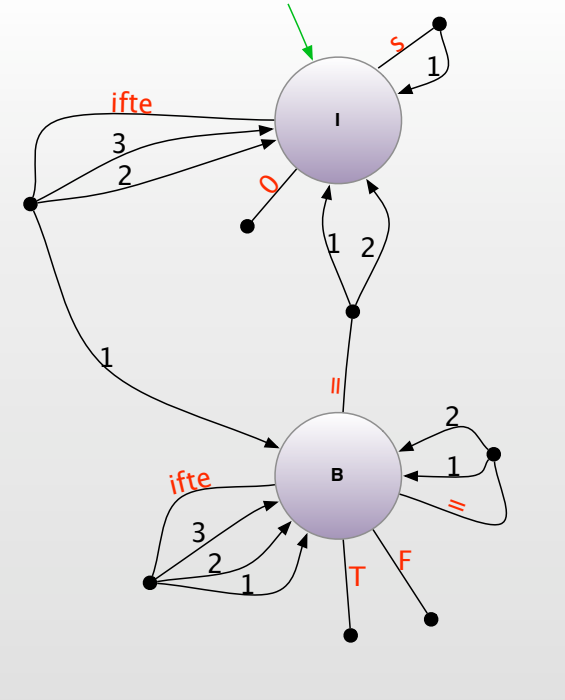
State	Symbol	Transitions
B	0	{ }
B	$S-$	{ }
B	$-+-$	{ }
B	F	{ () }
B	T	{ () }
B	$-\wedge-$	{ (B,B) }
B	$\neg-$	{ (B) }
B	$-=-$	{ (B,B), (N,N) }
B	$if-then-else-$	{ (B,B,B) }
N	0	{ () }
N	$S-$	{ (N) }
N	$-+-$	{ (N,N) }
N	F	{ }
N	T	{ }
N	$-\wedge-$	{ }
N	$\neg-$	{ }
N	$-=-$	{ }
N	$if-then-else-$	{ (B,N,N) }

A valid expression is made up of the following symbols (corresponding to the constructors of the HASKELL datatype) respecting their arities as summed up in table 1.

Reading the symbols in a tree will start from the top level. Each of the possible symbols will determine a transition (or several, as we are considering non-deterministic automata) of an appropriate form. For example, if we are in a certain state s_0 and read a $\wedge$, we can make transitions to a pair of new states s_1 and s_2: reading the symbols in the first argument of $\wedge$ will continue from state s_1 and similarly the symbols in the second argument will be read starting from s_2.

Our automaton will have two states (B and N, for boolean and natural). The informal intended meaning of these is that reading a tree starting in state T will succeed if that expression can have type T. So, our initial state will be N (as we want to recognize expression representing natural numbers). The transitions are summarized in table 2 and are pretty self-explanatory. Figure 3 has a diagramatic representation of the automaton (which we hope to be clear enough, despite the transitions by the symbols $+$, $\wedge$ and $\neg$ having been ommited).

Figure 3: Representation of Example Tree Automaton



3.2 The General Case

3.2.1 Definition

After analyzing the example, the general definition is quite simple. For now, we will restrict ourselves to tree-like data types whose pattern functor is a sum of products of recursive occurrences (thus there are no parameters or constants of any kind).

We saw that each symbol (constructor of the datatype) determines transitions of an appropriate type. If the automaton is deterministic, the transition function could be given the following dependent type.

$$(c : A) \to S \to S^{arity(c)}$$

In the case of a non deterministic automaton, this would be

$$(c : A) \to S \to \mathcal{P}(S^{arity(c)})$$

It is worth noticing that, as we have a powerset in the result, we no longer need such complicated type and can instead consider the transition function to be of type

$$S \to \mathcal{P}(FS)$$

where F is the pattern functor of the data type under consideration. In this case, when given a tree, we cannot directly compute the possible set of children states analysing the top level symbol and the current state. Instead, we compute all the possible top level terms generated from the current state and select only the ones amongst those which match the structure of the tree we are trying to parse.

We will use this last definition as it is a lot simpler to implement. It would still be possible to implement the other approach using a type class, which would define the structure of the transition function from the structure of the pattern functor. This would be somehow a mixture of the type class defining the type of algebras and of coalgebras (because we would have a coalgebraic

structure for each of the alternatives). If one would like to consider constant types in our data type, this would probably be a better approach, as we could make the transition depend on those values. Once again the same mixture: as in the definition of algebras, a constant type would go into the arguments; as in the definition of coalgebras, a recursive occurrence would go into the result.

3.2.2 Bottom Up *versus* Top Down

There are two alternative definitions of tree automata: top-down and bottom-up. The former kind is the one we have been considering. Such an automaton starts parsing a tree-shaped word by its root and each symbol determines a move to a new state for each direct child of the current node. The alternative approach is to start with each leaf in one state and compute the state for the parent from its symbol and the states its children are in. In the classical case of lists, this corresponds to the distinction between left-to-right (top-down) and right-to-left (bottom-up) parsing of the word.

The shape of transitions for bottom-up automata is somehow dual of the top-down case already presented. We have

$$FS \to S$$

for deterministic automata and

$$FS \to \mathcal{P}S$$

for the corresponding non-deterministic.

From the signatures, we can see that bottom-up automata have a algebraic structure, contrasting to the coalgebraic one of top-down automata.

In what concerns the recognition of languages of finite trees, it is worth noticing that bottom-up deterministic automata are more powerful than the top-down counterparts, although the non-deterministic versions are equivalent (and as expressive as deterministic bottom-up).

3.2.3 Coalgebras, again!

For the reasons outlined below, we decided to focus only on (nondeterministic) top down tree automata.

- The coalgebraic approach lets us tackle infinite input trees. This would be impossible in bottom-up automata because the complete tree is needed so that we can start computing the result from the leaves. (This is a recurrent distinction between algebraic and coalgebraic approaches)
- The coalgebraic pattern is somehow closer to the one discussed for F-automata. So, we hope this eases the integration of the two extensions.
- Less importantly, top-down instantiate to the more natural left-to-right version of list automata

3.3 Representation

3.3.1 Datatypes

As we already mentioned, a tree automaton will be a coalgebra in a relational setting or, equivalently, a set of coalgebras or a coalgebra whose image is in a powerset. However, to allow us to compute with this, we need to impose a further restriction: the image of a certain state has to be finite. Thus, the correct type of transition we are interested in is

$$S \to \mathcal{P}_{fin}(FS)$$

Actually, in order to simplify the treatment in HASKELL we will use lists instead of sets.

We begin by giving the type representing a (deterministic) coalgebra and the related combinator *unfold* which gives the anamorphism determined by a coalgebra.

```
type CoAlgebra f r = r → f r
unfold  :: (Regular t) ⇒ CoAlgebra (PF t) r → r → t
unfold g = to · fmap (unfold g) · g
```

We now consider a monadic version, where the result is wrapped in a monad. There is also a naturally defined *unfoldM*.

```
type CoAlgebraM m f r = r → m (f r)
unfoldM  :: (GMap (PF t), Regular t, Monad m) ⇒ CoAlgebraM m (PF t) r → r → m t
unfoldM g = liftM to · join · liftM (fmapM (unfoldM g)) · g
unfoldM g = liftM to · fmapM (unfoldM g) <=< g
```

The type of tree automata is now simple to define:

```
type TreeAutTrans t s = CoAlgebraM [] (PF t) s
data TreeAutomaton t s = TA {δ :: TreeAutTrans t s, initials :: [s]}
```

Note that we are using here the original *Regular* library as we do not need knowledge about type parameters.

3.3.2 Example

The example automaton presented in Section 3.1 would be encoded as follows.

```
instance Regular Expr where
  type PF Expr = 1 + Id + Id × Id + 1 + 1 + Id × Id + Id + Id × Id + Id × Id × Id
  ...
data Types = N | B
texample :: TreeAutTrans Expr Types
texample N = [L 1                                                    -- Zero
             , R (L (Id N))                                          -- Suc
             , R (R (L (Id N × Id N)))                               -- Plus
             , R (R (R (R (R (R (R (Id B × Id N × Id N))))))))       -- IfThenElse - Nat
             ]
texample B = [ R (R (R (L 1)))                                       -- False
             , R (R (R (R (L 1))))                                   -- True
             , R (R (R (R (L (Id B × Id B))))))                      -- And
             , R (R (R (R (R (L (Id B)))))))                         -- Not
             , R (R (R (R (R (R (L (Id B × Id B))))))))              -- Eq - Bool
             , R (R (R (R (R (R (L (Id N × Id N))))))))              -- Eq - Nat
             , R (R (R (R (R (R (R (Id B × Id B × Id B))))))))       -- IfThenElse - Bool
             ]
example :: TreeAutomaton Expr Types
example = TA {initials = [N], δ = texample}
```

3.4 Operations

3.4.1 Recognizing

The anamorphism determined by the coalgebra should produce the whole recognized language.

```
language :: (Regular t, GMap (PF t)) ⇒ TreeAutomaton t s → [t]
language aut = unfoldM (δ aut) =≪ initials aut
```

Of course, this function is not very useful unless the language is finite. Otherwise, as only one word is calculated each time, most likely certain small words will never be reached as the funcion will get stuck with the first recursive occurrence.

A more useful function is *accept*, which determines whether a tree is recognized by an automaton.

```
accept :: (Regular t, Zip (PF t)) ⇒ TreeAutomaton t s → t → Bool
accept aut t = (isJust · msum) [accept' (δ aut) t i | i ← initials aut]
accept' :: (Regular t, Zip (PF t)) ⇒ TreeAutTrans t s → t → s → Maybe ()
accept' δ t = msum · map (liftM (!) · fzipM (accept' δ) (from t)) · δ
  where (!) = ()
```

Note that we use the generic zip function, which is defined in the original Regular library. The function works as we have already explained: it takes the next possible continuations (trees with one root and states at the children level). From those, it selects the ones which match the structure of the tree of interest (hence the *fzipM*) and recurses into all the children in these cases. The bang and the *msum* are used to indicate the interesting information: whether any of the possible transitions happened to be successful.

We can see this function working in two examples.

```
*TreeAutomata> accept example (IfThenElse (Eq Zero Zero) (Suc Zero) (Plus Zero Zero))
True
*TreeAutomata> accept example (IfThenElse (Eq Zero Zero) (Suc Zero) (Eq Zero Zero))
False
```

3.4.2 Partial Runs

Another interesting function is one that performs only one step and returns the possible continuations (association of a state to each child tree).

```
runStep :: (Regular t, Zip (PF t)) ⇒ TreeAutTrans t s → t → s → Maybe [PF t (t, s)]
runStep δ t s = sequence [fzip (,) (from t) m | m ← δ s]
```

This function is particularly useful in the context of infinite trees. Note that the *accept* function runs as a depth-first search. Although this suits checking for acceptance of finite trees, the only way to deal with infinite structures is by a breath-first search. That can be used to run a finite part of a infinite tree (truncate it at a certain depth) or even for determining whether the recognized language is empty under a certain acceptance condition (for example, a Buchi condition generalized for tree automata). The latter application is quite useful in such a context of parsing infinite trees.

3.4.3 Set-like Operations on Languages

The automaton that recognizes the union of the languages recognized by two other automata is simple to define. We consider a disjoint union (sum) of the set of states and of the transitions. The set of initial states is also the union of the sets of initial states.

```
union :: (Regular t) ⇒
        TreeAutomaton t s1 → TreeAutomaton t s2 → TreeAutomaton t (Either s1 s2)
union aut1 aut2
        = TA {initials = map Left (initials aut1) ++ map Right (initials aut2)
             ,δ        = map (fmap Left) · (δ aut1) ||| map (fmap Right) · (δ aut2)
             }
```

The intersection is somewhat more complicated. As usual, we need to consider the product of the set of states. A transition corresponds to a simultaneous transition in the two automata. We need to bear in mind that these two simultaneous transitions have to generate the same structure (symbol/ constructor). Hence, the use of $fzip$.

$$
\begin{aligned}
&intersection :: (Regular\ t, Zip\ (PF\ t)) \Rightarrow \\
&\qquad TreeAutomaton\ t\ s1 \to TreeAutomaton\ t\ s2 \to TreeAutomaton\ t\ (s1, s2) \\
&intersection\ aut1\ aut2 \\
&\qquad = TA\ \{ initials = cartesian\ (initials\ aut1, initials\ aut2) \\
&\qquad , \delta = map\ fromJust \cdot filter\ isJust \cdot map\ (\widehat{fzip\ (,)}) \cdot cartesian \cdot (\delta\ aut1 \times \delta\ aut2) \\
&\qquad \}
\end{aligned}
$$

3.4.4 Other

A last set-like operation we would like to implement is the complement. It seems to be possible to define that operation if the set of states is finite (still the input trees can be infinite). These operations combined could be used to calculate inclusion of (regular) tree languages:

$$\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2) \equiv \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\bar{\mathcal{A}}_2) = \emptyset \equiv \mathcal{L}(\mathcal{A}_1 \cap \bar{\mathcal{A}}_2) = \emptyset$$

In the spirit of the typechecking example, this can be read as a kind of type refinement. In the context of processes (where the trees are infinite), this can also be seen as some kind of refinement of a specification or satisfaction of some property. Note that the only missing part here is the check for emptiness. This is simple to do in for finite word languages. For infinite word languages, this can also be done: in fact, this emptiness check on the intersection of an automata and a negation of another is a common pattern in model checking.

Another interesting extension to do is to consider trees with holes. It looks like these trees can be used to express tree transformations, substitutions, etc.

4 Integrating the two abstractions

The two extensions discussed in Sections 2 and 3 were implemented quite independently, although some decisions were taken with integration in view, such as selecting the library used for F-automata or opting for the coalgebraic approach for tree automata.

This latter decision means that both extensions have a lot of similarities. In fact, it seems to be possible to integrate both by considering transitions of the form

$$S \to (F \cdot G)S$$

where G is the pattern functor of the input tree type and F is an arbitrary collection-like functor as in Section 2. However, this might not yet be the optimal solution as it looks more like output that input. These are more or less equivalent in a nondeterministic setting (with $F = \mathcal{P}$) but not in general. So, it might be good to consider other approaches.

Also, other important question is whether the generic definitions of operations for F-automata actually extend when we consider tree-like input. This seems (almost) to be the case, but we still need to think more accurately about this.

5 Conclusions

5.1 Generic Programming

This project was an exercise on using generic programming to build an actual library. Here we leave an account of our personal opinions taken from this experience.

- Generic programming involves a higher level of abstraction. Thinking generically helps focusing on the essential problems and thus better understanding the domain.
- A generic library can be instantiated for many different specific applications. In our case, lots of different classes of automata can be expressed in a common interface.
- An important feature of a generic library is extensibility. This was perhaps not easy to achieve in our project: we needed quite a workaround to express powersets. But, apart from that, our universe of functors seems to be extensible (in Section 5.2, we refer another useful extension that could be done).
- Generic programming techniques seem to suit the problems addressed in this project. We used an existing library: although we needed to adapt it, the underlying ideas were very similar.
- Most of our domain-specific functions made use of generic functions that are usually available on GP libraries: maps, folds, zips, etc... This indicates the adequacy of these programming patterns to be used in several applications.
- We experienced some problems as we did not know of any library that suited all our needs: knowledge of the pattern functor and indication of the type parameter. PolyP, a language extension, supports this, but we do not know if any actual library does. We think that a good comparison summary of all the approaches available would have been quite useful (although it was probably our fault that we did not find one).

5.2 F-Automata

In our opinion, we were able to develop this first extension to a satisfactory state. Although, there are still some issues which should be corrected (or, at least, rethought) and a lot of ways in which this work can be continued. The following list is a brief summary as most of these issues were already discussed through the report.

- The universe of functors should be reconsidered. We have implemented weighted automata (edges with costs) using the library to test the approach. An important step now is to try other classes and check how well they can be described, so that we can adapt the library (if necessary) to find an optimal point.
- An interesting class of automata that could be introduced are probabilistic automata. This would probably require us to extend our universe with a probabilistic space, that would enforce some restrictions and implement some operations. This is feasible. We are not sure however if this can be achieved without such an extension.
- While mixing use of $Functor$ and $EFunctor$ (introduced to allow powersets), we ran into some hard to solve compile errors. This probably indicates that there is something wrong with the approach that we should try to correct.
- There are certainly a lot more things that can be done in this generic setting: IO automata should be implemented, together with composition, and we should think of other interesting operations, such as interleaving of automata, etc.
- A cycle detection algorithm could be implemented. This would be useful for automata with infinite words, particularly for model-checking.
- The problem discussed in Section 2.6 about a generic flattening operation seems to be a interesting one.

5.3 Tree Automata

We were also able to implement the tree automata generalization to some extent. Also, a lot of things remain undone.

- Allow less restricted pattern functors.
- Consider whether it is useful to generalize this to mutually recursive datatypes.
- Implement the complement operation so that we can define language inclusion.
- Implement Buchi-like acceptance conditions for infinite-trees, that is, an empty check according to those conditions.
- Take further the $runStep$ and $runPartial$ functions and consider tree with arbitrarily placed holes as well as some of its interesting applications.
- Integrate the two extensions.