Finite State Transducers in Haskell

Markus Forsberg

August 28, 2001

Abstract

This work describes an implementation of a finite state transducer system in the functional programming language Haskell. Finite state transducers are models that recognizes regular relations, and have, for example, proven its usefulness in natural language processing, especially concerning morphological analysis. The work is divided into two parts. The first part gives a theoretical background to finite state transducers, mainly formal language theory and automata theory. The second part describes the implementation of a finite state transducer system in the programming language Haskell called 'FstStudio'. The implementation resulted in two things - a program for constructing and running transducers (manual can be found in Appendix C), and a Haskell library for programmers who want to use transducer technology in their programs (description of this library can be found in Appendix B).

Acknowledgement

I would like to thank my tutor Aarne Ranta, who introduced me to the subject of transducers and has always been there for me throughout the work when I had some questions or if I needed help to get out of a general state of confusion. I would also like to thank my friend Björn Nyberg, who was invaluable in the final step of this work. Finally, I would like to direct a general thanks to all the wonderful people in my life, none mentioned, none forgotten.

Contents

Chapter 1

Introduction

This work is about **regular languages**, about describing, recognizing and generating regular languages and relations over them. Our interest is focused on the models that recognizes and generates regular languages and regular relations - **finite state automata** for ordinary regular languages, and **finite state transducers** for regular relations. For our discussion of regular languages, we need some theoretical background, so let us start with an examination of the concept of languages.

1.1 Languages

There are many alternative ways of describing languages, each description stressing a particular aspect. In this work we are primarily interested in the **syntax** of languages and we have choosen a description that reflects this. Let us first look at a possible definition of a language:

Definition of language 1.1
*A **language** is constructed by arbitrary, abstract entities called **symbols**, which are combined into sequences named **strings**, by a set*[1] *of rules called a **grammar**. A language is a set of strings.*

Symbols and strings

A symbol can be just about anything - a letter, a digit, a word, a musical note or anything else. The symbol itself has no meaning, it is given a meaning, or a **denotation**. This denotation is arbitrary, that is, the symbol has no necessary connection with what it denotes. The number of symbols in our description is always finite, and the set of all symbols of a language is called the **alphabet**, denoted by the Greek symbol Σ.

[1] A set is a collection of unordered items with no duplicates.

There is a special kind of symbol, denoted with ϵ, that stands for 'no symbol'. This symbol should be interpreted as unit element in the same way as 0 is the unit element of addition. An alphabet including ϵ is symbolized with Σ^{ϵ}.

A string is a sequence of symbols, often written with the symbols enclosed by quotes "". Another word for a string is sentence, which may feel more intuitive. A language is described by a set of possible strings in that language. A language can, for example, be all the binary numbers {"0","1","01","10","11"..}, where the alphabet is the binary digits {0,1}.

Note that the concept of a language as a set of strings is in some cases an overly simplified description of a language - it doesn't take into account the hierarchical, tree-shaped, relations between constituents in a language. However, in this work the string-description is adequate and sufficient for our purposes.

1.1.1 Grammar

A grammar is a description of how to combine symbols into strings constituting a language. We will use a grammar model constructed by the linguist Noam Chomsky, often called the **Chomsky's hierarchy**. Noam Chomsky attempted to classify different kinds of languages by the complexity of their grammar. Chomsky identified four language classes (sets of languages), which he named **type 0**, **type 1**, **type 2** and **type 3**. The language classes are hierarchically ordered, that is, all languages describable by a type 3 grammar is also describable by type 2, type 1 and type 0 grammar, but not vice versa. Mathematically speaking: $type3 \subset type2 \subset type1 \subset type0$.

Every class has corresponding models for recognizing/generating strings in a language of that class. These models are often called **automata**, and the structure of a automaton is defined by a grammar. More about automata later on.

This work is primarily about the languages in type 3, called **regular languages**, but first we shortly present all the other classes to give some perspective. We start with type 0, the class of languages with the most powerful grammar, often referred to as **unrestricted grammar**.

Type 0

Definition of type 0 grammar 1.2
A type 0 grammar is a 4-tuple $G = (N, \Sigma\, S, P)$, where N and Σ are disjoint finite sets, $S \in N$, and P is a finite set of formula of the form $\alpha \rightarrow \beta$, where $\alpha, \beta \in (N \cup \Sigma)^*$. Elements of Σ are called terminals or symbols. Elements of N are called non-terminals or variables. S is the start symbol, and the elements in P are called grammar rules or productions.

A language in type 0 can be recognized or generated by a **Turing machine**, an automaton supposedly powerful enough to model all existing computer systems. For further details about Turing machines, look in any elementary book about computer science.

The best way to explain this definition is to start with an example. Let us look at how to describe the language consisting of all binary numbers. This language does not need to be described by a type 0 grammar, a type 3 grammar is sufficient, but it serves well as an example (remember that a type 3 grammar is also a type 0 grammar, but not vice versa).

One way of defining the grammar is the following definition, where ϵ stands for 'no symbol', or the empty string "".
Σ = {0,1}
N = {S}
P = $\{S \to 0S, S \to 1S, S \to \epsilon\}$
There is a shorter way of writing the production rules by using some syntactic sugar:
P = $\{S \to 0S|1S|\epsilon\}$
The "|" symbol can be considered as "or".
Let us now derive the string "01":
$S \to 0(S \to 1(S \to \epsilon))$
which results in the string with the symbols "01".

Note that in the above grammar it is possible to derive the empty string "", which might seem strange. Let us redefine the grammar so that we no longer have the possibility to derive the empty string.
Σ = {0,1}
N = {S}
P = $\{S \to A|B, A \to 1A|B|1, B \to 0B|A|0\}$
The derivation of the string "01" now has the following form:
$S \to (B \to 0(A \to 1))$

Now there is another problem with the grammar, it is **ambiguous**. A grammar is ambiguous when there are several ways to derive a string. In the grammar above it is a infinite number of ways to derive the string "01". For example, we could derive it as follows:
$S \to (A \to (B \to 0(B \to (A \to 1))))$
This problem is fixed by a rewriting of the production rules:
P = $\{S \to 1A|0B, A \to 1A|0B|\epsilon, B \to 0B|1A|\epsilon\}$

Type 1

The second most powerful grammar is the type 1 grammar, also named **context-sensitive grammar**, a name which focus on the essential property of type 1 grammar - transformations of a non-terminal depends on which non-terminals or terminals surrounds that non-terminal - its **context**. We define type 1 grammar in terms of type 0 to reflect that every step from Type n to Type n+1 introduces some further restrictions.

Definition of type 1 grammar 1.3
A type 1 grammar has the same structure as type 0 grammar, with the restriction that only ***one*** *left-hand side non-terminal symbol gets replaced in a production rule.*

This means that we can have a production rule such as:
$aSbN \rightarrow acbN$
but not a rule where two non-terminals are replaced:
$aSbN \rightarrow acbd$

In most cases, it is very difficult to prove that a Type 0 grammar can not be described by a Type 1 grammar. In fact, there are no easy examples on grammars that are not describable by a Type 1 grammar, but they have been proven to exist.

Type 2

An alternative name for Type 2 grammar is **context-free grammar**. The name comes from the fact that the transformation of a non-terminal is independent of its context.

Definition of Type 2 grammar 1.4
A type 2 grammar has the same structure as type 1 grammar, with the further restriction that all production rules has no terminals and only ***one*** *non-terminal symbol in the left-hand side of the production rules.*

A Type 2 grammar can be recognized/generated by a push-down automaton, similar to a finite state automaton (see section about FSA) with the exception that it has a stack that functions as a memory. The stack enables the automaton to remember such things as matching parenthesis, or expressed differently, languages of the form $a^n b^n$. A language of matching parenthesis can be described by the following production rule:
P $\rightarrow$ '(' P ')' | ϵ

1.2 Regular languages

Type 3 grammar is also called regular grammar, and as mentioned in the beginning of this introduction, the languages describable by regular grammar, the so called regular languages, are the main focus of this work. Regular grammar is quite restricted, but in many cases it will be sufficient. Regular languages are, for example, capable of describing all finite languages.

This means that we can approximate other grammars by setting some upper limit and thereby making it finite - the language of matching parenthesis can be made regular by setting a limit on the depth of the parenthesis, say thousand, or if that is not enough, a million.

Regular languages has many interesting properties, especially its close connection to finite state machines, more about this later on . We first start with a definition of type 3 grammar.

Definition of Type 3 grammar 1.5
A type 3 grammar has the same structure as type 2 grammar, with the further restriction that every production rule must either be of the form
$A \rightarrow aB,\ A \rightarrow a,\ A \rightarrow \epsilon$
or
$A \rightarrow Ba,\ A \rightarrow a,\ A \rightarrow \epsilon$
where $A, B \in N$ (not necessary distinct) and $a \in T$.

There is an alternative way of defining the class of regular languages, one that does not focus on the grammar, but on the forming of sets. These two definitions are in fact equivalent.

Definition of the set of regular languages/expressions 1.6
Let $\mathcal{R}$ be the set of all regular languages defined over an alphabet Σ then:
A1. Symbol $\forall\ a \in \Sigma\ ,\ \{"a"\} \in \mathcal{R}$
denoted by the regular expression: a
A2. Epsilon $\{""\} \in \mathcal{R}$
denoted by the regular expression: ϵ
A3. Empty set $\{\} \in \mathcal{R}$
denoted by the regular expression: $\emptyset$
if $L_1, L_2 \in \mathcal{R}$, denoted by the regular expressions r_1 and r_2 then:
C1. Union $L_1 \cup L_2 \in \mathcal{R} = \{\ x \mid x \in L_1 \vee x \in L_2\ \}$
denoted by the regular expression: $r_1 \mid r_2$
C2. Concatenation $L_1 \cdot L_2 \in \mathcal{R} = \{\ xy \mid x \in L_1 \wedge y \in L_2\ \}$
denoted by the regular expression: $r_1\ r_2$
C3. Kleene's star $L_1^* \in \mathcal{R} = \{\ "", x, xx, xxx... \mid x \in L_1\ \}$
denoted by the regular expression: r_1^*
No other languages than those that can be constructed by the rules above are regular.

In the above definition, we have three atomic sets (A1,A2,A3), and three set combining operations (C1,C2,C3). The first set combining operation C1 is just normal set union (can be thought of as choice) and C2 is concatenation of strings in two sets (can be thought of as sequencing). Kleene's star (C3) creates a, possibly infinite, set that includes the empty string, the strings in the set, and the repeated concatenation of the strings in the set with themselves. (can be thought of as repetition). We will from now on only use regular expressions when describing regular languages, but the reader should keep in mind that this is only a shorthand for the set construction rules defined above.

Kleene's star has the strongest binding in a regular expression, then concatenation and union has the weakest. This binding, or precedence with another word, is canceled with square brackets ('[',']').

Let us look at some examples of regular expressions:

Regular expression	**Regular language**
$[a\ b\ c]$	{”abc”}
$[a \vert b\ c]$	{”a”,”bc”}
$[[a \vert b]\ c]$	{”ac”,”bc”}
$[[a \vert b]^*]$	{””,”a”,”b”,”aa”,”ab”,”bb”,”ba”, ..}
$[[0 \vert 1]\ [0 \vert 1]^*]$	{”0”,”1”,”00”,”01”,”10”,”11” ..}
$[\epsilon^*]$	{””}

The last example may need some explanations - remember that ϵ can be considered as an unit element in the same way as zero in addition; the concatenation of the empty string yields the empty string no matter how many times you concatenate.

The equivalence between regular grammar and the set formation rules are not trivial to notice, so we will present a transformation table to make the connection clearer. Note that the derivation is from left to right.

Regular expression	**Production rules**
a	P $\rightarrow$ a
ϵ	P $\rightarrow \epsilon$
$\emptyset$	No production rule
$.. \mid ..$	P $\rightarrow .. \mid ..$
$a_i\ a_{i+1}\ ..\ a_j$	A $\rightarrow a_iB$, B $\rightarrow a_{i+1}C$, C $\rightarrow$ .. E , E $\rightarrow a_j$
A^*	P $\rightarrow$ A, A $\rightarrow$ .. E , E $\rightarrow \epsilon \mid$ P

Set formation rules and regular grammar will from now on be used interchangeably, depending on what description is the most appropriate in the given situation.

Let us now introduce some further notation to make it easier to describe regular languages. The notation should be understood as macro definitions of regular expressions.

Notation	Regular expression	Meaning
0, []	ϵ	Alternative notation for ϵ
?	$a_1\|..\|a_j,\ a_i \in \Sigma$	Union of all $a_i \in \Sigma$
?*	$(a_1\|..\|a_j)^*,\ a_i \in \Sigma$	The universal language
(A)	$A\|\epsilon$	A is optional
A^+	$A\ A^*$	Kleene's plus, ϵ excluded
$[A]$	?* A ?*	Containment
{abc}	a b c	Concatenation of a b and c.

Simplification

One nice property of regular languages is that because the grammar is described by set forming operations, we can use fundamental set theory rules to simplify the grammar. This is important because a simplified grammar also simplifies the computation that is needed to recognize/generate the described langugage. In Appendix A can some of the possible rules of simplification be found.

1.3 Finite State Automata (FSA)

We will now look at finite state automata, a model that has the interesting property that every regular expression can be transformed into a finite state automaton that recognizes/generates the language described by the expression, and every finite state automaton has a corresponding regular expression that describes the language it recognizes/generates, or in other words, every language a finite state automaton can recognize is regular. This equivalence relation are called the **Kleene's correspondence theorem**[2], a very powerful theorem that makes proving properties about regular languages a quite easy task.

Let us start with a definition of finite state automata.

Definition of a finite state automaton (FSA) 1.7
A finite state automaton is a 5-tuple $(Q, \Sigma\, \delta, S, F)$*, where:*
Q *is a finite set of states,*
Σ^ϵ *is a finite set of symbols,*
$\delta : (Q \times \Sigma^\epsilon) \to \mathcal{P}(Q)$ *is the transition function,*
$S \subseteq Q$ *is the set of initial states,*
$F \subseteq Q$ *is the set of final states, or accepting states.*

An automaton starts at the initial states, moves from states to states in response to the input with respect to the transition function. If it halts at a

[2]Proof, see Martin[10]

final state with all inputs consumed, then the input is accepted, otherwise, it is rejected.

We will use a subset of natural numbers as the set of states Q, but any other set that is discrete, infinite or countably infinite, would work just as well. If an FSA is denoted by M, then the language a FSA recognizes is denoted by L(M). The size of an automaton is measured by the cardinality of the set of states: $|M| = |Q|$.

Let us now investigate if a string is member of the language that a FSA recognizes. We start at the initial states, moves from a state to another in response to the input string according to δ-function. At every transition, we consume a symbol (with the exception of ϵ-transitions, where no symbol is consumed), and when we have consumed all symbols, we examine if we are in a final state(s) - if that is the case, then the string is accepted, otherwise it is rejected.

Note that a function can be understood as a relation, and this is how we sometimes going to interpret the δ-function.

An example is in place, so let us look at the finite state automaton M that accepts the language {"ab"}.

$$\mathrm{M} = \{\mathrm{Q} = \{0,1,2\},\ \Sigma = \{\mathrm{a,b}\},\ \delta = \{(0,\mathrm{a},1),(1,\mathrm{b},2)\},\ \mathrm{S} = \{0\},\ \mathrm{F} = \{2\}\}$$

We have the string "ab", and want to check if it is in L(M). We start at state 0, read an 'a' and move to state 1. At state 1 we read a 'b' and move to state 2. At state 2, we have no more symbols, and checks if state 2 is a final state. That is the case, and we know that "ab" $\in$ L(M).

1.3.1 $\mathcal{R} \rightarrow FSA$

In this section we present an algorithm for converting regular expression to a finite state automaton. The algorithm was given by Thompson.[3] The alphabet in this description is omitted because it remains unchanged under the whole construction process, but the reader should keep in mind that an finite state automaton always is defined over a finite alphabet. In this algorithm ϵ-transitions are permitted, so the alphabet is denoted with Σ^{ϵ}.

build :: $\epsilon \rightarrow FSA$

Let q_0 and q_1 be two new states, then the FSA accepting ϵ is constructed by:
$\mathrm{Q} = \{q_0, q_1\}$
$\delta = \{(q_0, \epsilon, q_1)\}$

[3]For a more detailed description of this algorithm, see [5].

S = $\{q_0\}$
F = $\{q_1\}$

build :: $\emptyset \rightarrow FSA$

Let q_0 and q_1 be two new states, then the FSA accepting $\emptyset$ is constructed by:
Q = $\{q_0, q_1\}$
$\delta = \emptyset$
S = $\{q_0\}$
F = $\{q_1\}$

build :: $a \rightarrow FSA$

Let q_0 and q_1 be two new states, then the FSA accepting ϵ is constructed by:
Q = $\{q_0, q_1\}$
$\delta = \{(q_0, a, q_1)\}$
S = $\{q_0\}$
F = $\{q_1\}$

build :: $A|B \rightarrow FSA$

Let q_0 and q_1 be two new states, and
$(Q_0, \delta_0, S_0, F_0)$ = build A
$(Q_1, \delta_1, S_1, F_1)$ = build B
then the FSA accepting $[A|B]$ is constructed by:
Q = $\{q_0, q_1\} \cup Q_0 \cup Q_1$
$\delta = (\{q_0\} \times \{\epsilon\} \times (S_0 \cup S_1)) \cup ((F_0 \cup F_1) \times \{\epsilon\} \times \{q_1\}) \cup \delta_0 \cup \delta_1$
S = $\{q_0\}$
F = $\{q_1\}$

build :: $AB \rightarrow FSA$

Let
$(Q_0, \delta_0, S_0, F_0)$ = build A
$(Q_1, \delta_1, S_1, F_1)$ = build B
then the FSA accepting [A B] is constructed by:
Q = $Q_0 \cup Q_1$
$\delta = (F_0 \times \{\epsilon\} \times S_1) \cup \delta_0 \cup \delta_1$
S = S_0
F = F_1

build :: $A^* \to FSA$

Let q_0 and q_1 be two new states, and
(Q, δ, S, F) = build A
then the FSA accepting $[A^*]$ is constructed by:
Q = $\{q_0, q_1\} \cup Q$
$\delta = (\{q_0\} \times \{\epsilon\} \times S) \cup (F \times \{\epsilon\} \times S) \cup (F \times \{\epsilon\} \times q_1) \cup \delta$
S = $\{q_0\}$
F = $\{q_1\}$

1.3.2 ϵ-free FSA

In the previous section we introduced an algorithm for constructing a FSA based on a regular expression. This algorithm is easy to describe and understand, but it has a strange property - it introduces many ϵ-transitions. It seems quite natural that it is possible to construct a automaton that has no ϵ-transitions, and that is in fact the case.

Construction of a ϵ-free FSA 1.8
A ϵ-free FSA is constructed by, for every ϵ-transition (s_1, ϵ, s_2), remove the ϵ-transition, and add the transitions at s_2 to s_1, and if $s_2 \in F$ then $F = \{s_1\} \cup F$.

1.3.3 Useful FSA

A **useful** FSA is a FSA where all the states are at least **reachable** from one of the initial states (start-reachable) and one of the final states (final-reachable). A state is start-reachable if it is possible by transitions to get to that state from a initial state. A state is final-reachable, if it can reach one of the final states by transitions. A useful FSA is constructed by traversing all states, and check if they are start- and final-reachable.

1.3.4 Reversed FSA

A reversed FSA is a FSA with all its transitions reversed, and where $S = F$ and $F = S$. If a FSA accepts the language $\{"abc"\}$, then the reversed FSA accepts $\{"cba"\}$.

1.3.5 Deterministic FSA

The definition of FSA in the previous section is **non-deterministic** in the sense that at any given state, there can be more than one transition for every symbol. This is a disastrous property in terms of computational complexity. To understand this, consider a state with two transitions on the same symbol, and let the states the transitions connects also to have two transitions on the same symbol, and continue likewise. The running time of recognizing a string is exponential! This example is of course constructed, but it shows the potential problem, especially when intuitively the running time should be linear. But luckily, there is a solution to the problem, and that is to construct a **deterministic** FSA, a FSA with a linear running time. Let us first define what it means that a finite state automaton is deterministic, and then describe how to construct such an automaton.

Definition of deterministic FSA 1.9
A deterministic finite automata has the following properties:

- *There is only one initial state.*
- *There is no ϵ-transitions.*
- *The transition function doesn't maps pairs in ($Qx\Sigma$) to multiple states.*

A FSA is made deterministic by something called **subset construction**, where the new states are subsets of the powerset of the states of the FSA. To understand why subsets are used, then consider a FSA that has a state 1 with the transitions: $(1, a, 2)$ and $(1, a, 3)$. This means that the FSA is non-deterministic, because we have two transitions for the symbol 'a'. One way to fix this is to consider subsets as states, where the subsets consists of informations about both states. The new transition at state $\{1\}$ is $(\{1\}, a, \{2, 4\})$.

Subset construction

$determinize(FSA = (Q_1, \Sigma, \delta_1, S_1, F_1)) =$
$Q = \mathcal{P}(Q_1) - \ \{\emptyset\}$
$\Sigma = \Sigma$
$\delta = \{(U, a, T) | q \in U \wedge \forall a \in \Sigma, \ \delta_1(a, q) \in T\}$
$S = \{S_1\}$
$F = \{U | U \in \mathcal{P}(Q_1) \wedge U \cap F \neq \emptyset\}$

After the subset construction, the subsets can be renamed with normal state names.

After the subset construction we now have another problem - an exponential space behaviour. We get 2^{Q_1} states after the construction. However, most of these states are not useful and are eliminated by a clever construction algorithm. It is possible to construct a FSA that has exponential space behaviour when it's determinized, but in practice, this is very unusual.

1.3.6 Complete FSA

A complete FSA is a FSA with a transition for every symbol at every state. More formally defined as:

Complete FSA 1.10 *A complete FSA has the following property:* $\delta(s,a) \neq \emptyset, \forall s \in Q \wedge \forall a \in \Sigma.$

Note that a complete FSA describes the universal language $?^*$, but does not necessary accept it.

A complete FSA is easy to construct. First of all, construct a new state s_{sink} that has a transition to itself for every symbol in Σ. Then, for every state, make a transition to s_{sink} on every symbol not present in that state's transitions.

1.3.7 Minimal FSA

One of the most important properties of deterministic FSA, is that there exists a unique, disregarding the names of the states, minimal FSA (minimal number of states) that accepts the same language.

To understand the concept of minimality of FSA, consider the two strings "revolution" and "intrusion", where both strings ends with "ion". In a FSA, this may correspond to two chains of states that derives the same substring. This means that the FSA contains redundant information, because only one chain is needed to derive the substring.

Most of the existing minimization algorithms are based on the concept of equivalence classes. An equivalence class is a partitioning of a set according to a relation that has the properties reflexivity, symmetry and transitivity.

We say that two states are equivalent, if they are indistinguishable. Two states are indistinguishable if they derive the same substring. Minimizing a FSA is the process of finding all states that are equivalent and merge them.

1.3.8 Extended regular expressions

Extended regular expressions are expression with intersection and complement or set minus (if we have complement, then we can define set minus, and vica versa, according to the set theory rule $A - B = A \mathbin{\&} \sim B$ and $\sim A = ?^* - A$).

Intersection of two regular languages L_1 and L_2, denoted by the extended regular expression $A \& B$, is the language L_3 whose elements are strings that are members both in L_1 and L_2. Minus of two regular languages L_1 and L_2, denoted by the extended regular expressions $A - B$, is the language L_3 which element are strings in L_1 but not in L_2. Complement of a regular language, denoted by the extended regular expression $\sim A$, is the the language L_2 that contains all the strings in the universal language $?^*$ that is not a member of L_1.

We will start with some examples of extended regular expressions, and then describe how these expressions can be defined with the help of a FSA (remember that according to Kleene's correspondence theorem, every language that a FSA accepts, is regular).

Extended Regular expression	Regular expression
$[a\|b]\&a$	a
$[a\|b]\&c$	$\emptyset$
$[a\|b]-a$	b
$[a\|b]-c$	$a\|b$
$a^{*}-a$	$\epsilon\|[a\ a\ a*]$
$\sim\emptyset$	?*
$\sim ?^{*}$	$\emptyset$

Complement of a regular language

We will construct the complement of a regular language by describing how to change a FSA accepting the regular language in such a way that it accepts the complement.

Complement of FSA 1.11 [4]
Let a deterministic, complete FSA accept the regular language L_1, then the ***complement*** *of the regular language L_1 is constructed by setting the final states of the FSA to $F = Q - F$.*

How should this definition be understood? A complete, deterministic automaton describes the universal language, but does not necessary accept it, and when the set of final states is negated, then every string that was not accepted by the automaton, is now accepted.

Intersection of two regular languages

The intersection of two regular languages is constructed by the **cross product** of two finite automaton, resulting in a FSA that has pairs of states as states.

Definition: Intersection 1.12 [5]
Let
$M_1 = \{Q_1, \Sigma, \delta_1, S_1, F_1\}$ *be a FSA that accepts* L_1
$M_2 = \{Q_2, \Sigma, \delta_2, S_2, F_2\}$ *be a FSA that accepts* L_2
Σ *of* M_1 *and* M_2 *are supposed to be equal.* [6]
The intersection $L_1 \cap L_2$ *is accepted by the following FSA:*
$Q = Q_1 \times Q_2$
$\Sigma = \Sigma$
$\delta = \{((s1, s2), a, (s3, s4)) | (s1, a, s3) \in \delta_1 \wedge (s2, b, s4) \in \delta_2 \wedge a = b\}$
$S = S_1 \times S_2$
$F = F_1 \times F_2$

[4] Proof, see Hopcroft[5].
[5] Proof, see Martin [10] .
[6] This is not a restriction, because if $\Sigma_1 \neq \Sigma_2$, then $\Sigma = \Sigma_1 \cup \Sigma_2$.

1.4 Regular relations

We will now look at regular relations. We will only look at binary relations by reasons of simplicity, but this is no necessary restriction. If we have the pair (a,b) then we will say that a is a string in the **upper language** and b is a string in the **lower language**.

Definition of the class of binary regular relations 1.13
Let $\mathcal{RR}$ be the set of all binary relations over regular languages defined on an alphabet Σ then:
A1. $\forall a_1, a_3 \in \Sigma \ \{(\text{"}a_1\text{"}, \text{"}a_2\text{"})\} \in \mathcal{RR}$
denoted by the regular relation: $(a_1 : a_2)$
A2. $\{(\text{""}, \text{""})\} \in \mathcal{RR}$
denoted by the regular relation: ϵ
A3. $\{\} \in \mathcal{RR}$
denoted by the regular relation: $\emptyset$
if $L_1, L_2 \in \mathcal{RR}$ *denoted by following regular relations* r_1 *and* r_2 *then:*
C1. $L_1 \cup L_2 = \{x | x \in L_1 \vee x \in L_2\} \in \mathcal{RR}$
denoted by the regular relation: $r_1 \mid r_2$
C2. $L_1 \cdot L_2 = \{(xy, zw) | (x, z) \in L_1 \wedge (y, w) \in L_2\} \in \mathcal{RR}$
denoted by the regular relation: $r_1 \ r_2$
C3. $L_1^* = \{(\text{""}, \text{""}), (x, y), (xx, yy), ... | (x, y) \in L_1\} \in \mathcal{RR}$
denoted by the regular relation: r_1^*
No other languages than those that can be constructed by the rules above are binary relations over regular languages.

Let us look at some examples of regular relations to get some further insight in what they denote.

Regular relation	**Relation over Regular language**
$[a : c]$	{("a","c")}
$[(a : b) \ (a : c)]$	{("aa","bc")}
$[[a : b] \| [c : d]$	{("a',"b"),("c","d")}
$[[a : b]^*]$	{("",""),("a","b"),("aa","bb"),("aaa","bbb"), ..}

We can construct a regular relation from a regular expression by applying the identity function **id**, for example, id(a) = (a:a).

1.5 Finite State Transducers (FST)

Finite state transducers are models that recognizes/generates regular relations. We will only consider binary relations, which are accepted by binary finite state transducers, where the first element in the binary relation are said to be in the **upper language**, and the second element in the pair is said to be in the **lower language**. The relation a FST T_1 accepts is denoted by $R(T_1)$, the upper language of T_1 by $L_u(T_1)$ and the lower language of T_1 by $L_l(T_1)$.

Definition of binary finite state transducer (FST) 1.14
A finite state transducer is a 5-tuple (Q,Σ δ,S,F), where:
Q is a finite set of states,
Σ is a finite set of symbols,
$\delta = (Q \times \Sigma \times \Sigma \times Q)$ is the transition relation,
$S \subseteq Q$ is the set of initial states,
$F \subseteq Q$ is the set of final states.

When running a transducers, we either recognize the upper language while generating the lower language, called **apply up**, or recognizing the lower language, and generating the lower language, called **apply down**.

If the symbol relation in the δ-relation is considered as a unit, then every property for FSA is applicable for FST, such as construction algorithms, epsilonfree (removing $\epsilon : \epsilon$), determinization, minimization and so on. There are, however, operations that are only applicable for transducers when the symbol relation is not considered as a unit, such as making a transducer deterministic with respect to the upper symbols of transducer, but this isn't something that will be examined further in this work.

1.5.1 Cross product

We will now describe how to construct an transducer that accepts the cartesian product $(A \times B)$ of two languages A and B. The cartesian product is also called cross product, and is defined as:
$A \times B = \{$ (a,b) $\mid a \in A \wedge b \in B \}$

Cross product of two automata 1.15 [7]
Given two automata $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ that accepts the the language $L(M_1)$ and $L(M_2)$. The Σ is considered equal, without loss of generality. The transducer accepting the cross product $L(M_1) \times L(M_2)$ is constructed by:
$Q = Q_1 \times Q_2$
$\Sigma = \Sigma$
$S_0 = \{q_1, q_2\}$
$F = F_1 \times F_2$
For every state $q_1 \in Q_1 \wedge q_2 \in Q_2$:
$\delta = \delta((q_1, q_2), a, b) = \delta_1(q_1, a) \times \delta_2(q_2, b)$
Cross product of the regular expression $r_1, r_2 \in \mathcal{R}$ is denoted by r_1 .x. r_2.

[7] Proof: See Kaplan and Kay [6].

If we have one automaton M_1 that accepts the language $L(M_1) = \{"a"\}$ and another automaton M_2that accepts the language $L(M_2) = \{"b", "c"\}$, then the cross product of the two automata $M_1 \times M_2$ yields the regular relation $L(M_1 \times M_2) = \{("a", "b"), ("a", "c")\}$.

1.5.2 Composition

Composition of binary relations is defined as follows, where A and B are binary regular relations:
$A \circ B = \{$ (a,d) $\mid (a, b) \in A \wedge (c, d) \in B \wedge (b = c) \; \}$

Composition of two transducers 1.16 [8]
Given two transducers $T_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ *and* $T_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ *that accepts the the relation* $L(T_1)$ *and* $L(T_2)$*. The* Σ *is considered equal, without loss of generality. The transducer accepting the composition* $L(T_1) \circ L(T_2)$ *is constructed by:*
$Q = Q_1 \times Q_2$
$\Sigma = \Sigma$
$S_0 = \{q_1, q_2\}$
$F = F_1 \times F_2$
$\delta = \{((s1, s2), (a, d), (s3, s4)) | (s1, (a, b), s3) \in \delta_1 \wedge (s2, (c, d), s4) \in \delta_2 \wedge b = c\}$

Composition of the regular relations $r_1, r_2 \in \mathcal{R}$ *is denoted by* r_1 *.o.* r_2*.*

If two transducers T_1 and T_2 accepts the regular relations $L(T_1) = \{("a", "cd")\}$ and $L(T_2) = \{("cd", "b")\}$, then the composition of the two transducers generates the regular relation $L(T_1 \circ T_2) = \{("a", "b")\}$.

The composition of two transducers is very similiar to the intersection of two automata, with the exception that we look at pairs instead of single symbols.

[8] Proof: See Kaplan and Kay [6].

Chapter 2

Implementation

This chapter describes the implementation of a binary finite state transducer in the functional programming language Haskell. The system described is called 'Fststudio'.

2.1 Background

The background to this work is two-fold. First of all, it results in an implementation of a transducer system in Haskell, a work that in our knowledge hasn't been done. There are lots of implementations of automata in Haskell, but none has been generalized into a transducer. Secondly, there exists some finite state transducer systems that have been implemented in C and Prolog, and it is of theoretical interest if properties characteristic to Haskell, such as a strict typing system and lazy evaluation order of expressions (only evaluate a value when/if it is needed) may give some positive effects.

2.2 Methodology

Abstraction

The ambition was to build a system with a relatively high level of abstraction, to further useability and to make changes in the system an easy task. For example, every data structure will be an ADT.

Modularity

Every part of the system that can be considered as a unit, should be put in its own module.

Reuse existing program

If there exist a solution to a problem or if there exists a tool that makes the solution of a problem easier, then this should be used instead of 'reinventing the wheel'.

Bottom-up design strategy

A bottom-up design strategy was decided, because it was quite easy to foresee what kind of functions are, if not necessary, so at least simplifying and useful.

Evolutionary design

The process of constructing the system was also a learning process - to understand the problems that needed to be solved. This means that a solution can rather be a step in the direction of understanding the problem, rather than a real solution. When a true understanding of the problem was achieved, then the real solution could be constructed.

2.3 Problems and their solutions

2.3.1 Insufficient heap space

One major problem is that the networks tend to be very large, at least in their intermediate form before they been transformed into their minimal form, and this creates a problem with insufficent heap space. The solution to this was partly to use profiling tools, which has been invaluable for the understanding of where the problems actually were in contrary to where one thought they were. This was a big help, but not a solution. The solution was to provide a way in the system to build subparts of the network, minimizing them, merge them with appropriate operator, for example union, and minimizing the whole structure. Unification of minimal subnetworks, instead of building the whole network, and then minimize it, avoids the intermediate networks to be too big for the heap.

2.3.2 Overloaded operators

One problem concerning parsing was that when given a expression, it is not always possible to decided the type of the expression by looking at the operator. Take, for example, the expression [A |B], you don't know if A and B are regular relations or regular expressions. Union, concatenation, and Kleene's star are all operators with this property - the operators are overloaded in the sense that they are applicable in diverse contexts.

A possible solution is to consider all expressions as regular relations. But this solution has some drawbacks - for one thing, it makes it possible to try to cross product regular relations, which is meaningless.

The choosen solution was to construct a neutral data type that contained both the regular expressions and the regular relations, and then to convert it to the respective data type by means of type checking.

2.4 This implementation

This section describes the implementation made that resulted in the system **fststudio**.

2.4.1 Regular languages and regular relations

First of all, we need a representation for regular expressions and regular relations, and in Haskell, the most natural choice is algebraic data types.

One for regular expressions:

```
data Reg a =
 Empty              | {- Empty language -}
 Epsilon            | {- Empty string -}
 All                | {- The all symbol -}
 Symbol a           | {- Symbol -}
 Reg a :|: Reg a    | {- Union. Note that :|: is an infix constructor. -}
 Reg a :.: Reg a    | {- Concatenation -}
 Reg a :&: Reg a    | {- Intersection -}
 Complement (Reg a) | {- Complement -}
 Star  (Reg a)        {- Kleene's star -}
```

And one for regular relations:

```
type Relation a = (Symbol a, Symbol a)

data Symbol a
 =    S a   | {- Symbol -}
      Eps     {- Epsilon symbol -}

data RReg a  =
 Cross     (Reg a)  (Reg a)      | {- Cross product  -}
 Comp      (RReg a) (RReg a)     | {- Composition  -}
 ProductR  (RReg a) (RReg a)     | {- Concatenation  -}
 UnionR    (RReg a) (RReg a)     | {- Union -}
 StarR     (RReg a)              | {- Kleene star  -}
```

```
 Identity  (Reg a)                | {- Identity relation  -}
 Relation  (Symbol a) (Symbol a) | {- Symbol relation  -}
 EmptyR                            {- Empty language  -}
```

The data type 'Symbol' is needed to be able to describe relations between ϵ and a symbol.

Now we are able to describe regular expressions and regular relations. However, we will not use these data type constructors directly when we construct a data type, instead we will use something called **Combinators** while building the data type. The combinators are overloaded functions that simplify the regular expression/relation while building it. Let us look at the union operator as an example.

```
class Combinators a where
 (<|>) :: a -> a -> a {- Union -}
 (|>)  :: a -> a -> a {- Concatenation -}
 ...

instance Eq a => Combinators (Reg a) where
 Empty <|> b = b
 a <|> Empty = a
 _ <|> (Star All) = Star All
 (Star All) <|> _ = Star All
 a1@(a :.: b) <|> a2@(c :.: d)
  | a1 == a2  = a1
  | a == c    = a |> (b <|> d)
  | b == d    = (a <|> c) |> b
  | otherwise = a1 :|: a2
 a <|> b
  | a == b = a
  | otherwise = a :|: b
 ...

instance Eq a => Combinators (RReg a) where
 EmptyR <|> r2     = r2
 r1     <|> EmptyR = r1
 a1@(ProductR a b) <|> a2@(ProductR c d)
  | a1 == a2  = a1
  | a == c    = a |> (b <|> d)
  | b == d    = (a <|> c) |> b
  | otherwise = UnionR a1 a2
 r1 <|> r2
  | r1 == r2       = r1
  | otherwise      = UnionR r1 r2
```

...

So if we write $A <|> B$, where A and B are arbitrary expressions, then two things has been accomplished. First of all, we do not have to know if A and B are regular expressions or regular relations. This is a good thing when we don't know what A and B are, or if we don't care. Secondly, an expression is simplified on the fly, which is better than first building an expression, and then simplify it.

2.4.2 FSA

The next step was to find an algorithm for constructing a FSA based on a regular expression. We had some requirements about the properties of the algorithm. First of all, we wanted an algorithm that generated an ϵ-free FSA, this primarily because of the memory usage. K. Thompson's algorithm constructs a very large FSA, which is then made small by ϵ-removal and minimization. However, this intermediate large FSA was something that we didn't find acceptable. Secondly, the algorithm must be able to handle extended regular expressions, such as complementation and intersection. Thirdly, the algorithm should result in a FSA that is easy to determinize and minimize. The final requirement was that the resulting representation of the FSA should be saveable/loadable. This last requirement is important, because there are some algorithms that runs very fast, but have the drawback that they are functions and therefore not saveable/loadable.

The algorithm choosen for this system was the algorithm called **LBFA** (Left-biased finite automaton). A description of this algorithm can be found in Watson [14].

There are, however, many other algorithms that fullfills the requirements, and are better then the LBFA algorithm in some situations, depending on, for example, the number of symbols or which operator that is the most common in the expression. A further development of the system is therefor to incorporate more algorithms, so that the user of the system can choose the algorithm that is most appropriate for the given expression.

To avoid multiple definitions of normal operations on FSA for every algorithm, such as determinization and minimization, we have constructed a neutral FSA data type and functions for the normal operations on FSA on that data type. This data type could be used for every algorithm, but this isn't always done naturally, so instead we decided to construct a type class for encoding and decoding a structure to the neutral FSA data type and vice versa.

```
data Automaton a =
             Automaton {
```

```
                        stateTrans  :: Array State (Transitions a),
                        initial     :: State,
                        finalStates :: FinalStates,
                        alpha       :: Sigma a
                        }

class Convertible f where
 encode :: Eq a => f a -> Automaton a
 decode :: Eq a => Automaton a -> f a
```

The minimization algorithm that was decided was **Brzozowski's minimization algorithm**, that construct a minimal automaton from a usefulS, ϵ-free, possibly non-deterministic automaton. This is a simple and beautiful algorithm, that constructs a minimal automaton by first reversing the automaton, then determinize it, reverse it again and lastly determinize it again. Determinization guarantee that two different state chains from the initial state do not derive the same substring (because if it did, then some state must have two transitions on the same symbol, which does not exist in a deterministic automaton), and by reversing the automaton twice, we get this condition with respect to both the initial state and the final states, and therefore the result is a minimal automaton.

2.4.3 Transducers

Transducers is a more general form of automata, because the transitions is on relations instead of symbols, but they are still automata. The structure of the transducers in the system is in many respects the same as that of automata, and one possiblity was to merge them into main structure, and have some small subparts that are specific for transducers and automata. However, they are different, relations opens new possibilities such as determinization with respect to the upper/lower language, so we decided to keep them separated, to ease modifications that are specific to automata or transducers.

2.4.4 Parsing FstStudio programs

To construct the parsing of FstStudio programs we used two programs, also written in Haskell, called 'Alex' and 'Happy'. Alex is a lexical analyzator that given a string, finds all symbols that are a unit, such as the equality sign '::=', and construct a token of it - TokenEq. This is done to simplify the next step, the parsing of the program. The parser was constructed with the help of Happy. Happy is parser generator for Haskell similiar to 'yacc' for C.

2.4.5 FstStudio

As a final step, a program was constructed for compiling and running transducers, the program that now goes under the name 'FstStudio'. This program is described in Appendix C.

2.5 Applications

As a final stage of this work, a couple of small applications where constructed/found to evaluate and put the system in practical use. One larger application was constructed, a morphological analysis of some Italian verbs. A subset of the program for morphological analysis of some Italian verbs looks like this:

```
<main> ::=
...
  [ {essere} "+Indic"  "+Imperf"  "+Sg"  "+P1" .x. {ero}]     |
  [ {essere} "+Indic"  "+Imperf"  "+Sg"  "+P2" .x. {eri}]     |
  [ {essere} "+Indic"  "+Imperf"  "+Sg"  "+P3" .x. {era}]     |
  [ {essere} "+Indic"  "+Imperf"  "+Pl"  "+P1" .x. {eravamo}] |
  [ {essere} "+Indic"  "+Imperf"  "+Pl"  "+P2" .x. {eravate}] |
  [ {essere} "+Indic"  "+Imperf"  "+Pl"  "+P3" .x. {erano}]   |
...
```

This can be considered as a lexicon, where one can for example lookup the word "ero", and get the information - "essere" in indicative, imperfective, singularis, first person. Lookup can of course also be done in the other direction.

We tested fstStudio with a program with Italian verbs consisting of ∼5000 lines and a program size of ∼320 kB. The program was split into 12 subparts, every part compiled separately. Every subnetwork was then unified and then union was finally minimized. It took around 15 minutes to construct the network.

The final network had 787 states and 1652 transitions, and the saved transducer size was 40 kB. With this network we have accomplished two things, a linear lookup time and a compression rate at 8.

2.6 Conclusions

The resulting system consists of ∼4500 lines of Haskell code and a compile binary of ∼2 MB, quite a large program to be written in Haskell. The system works fine, but has some problems with larger networks. This is however not a problem, because every program can be divided into smaller parts, and then combined.

Haskell's closeness to mathematical definitions has been a great asset while implementing the system. In most cases, the translation from a definition to a program was done with just some small modifications.

The strict typing system in Haskell has helped to avoid many errors that would be accepted in a more forgiving programming language.

Haskell's lazy evaluation of expressions (only evaluate a expression if it's needed, and only do this one time) did not have any notable positive effects running the transducer. To understand why, consider how minimization is done. To know that a transducer is minimal, we have to compare every state with every other state to decide if they are equivalent, but if we do this, then the whole transducer must be constructed, and none of Haskell's lazy features are used.

The system is far from complete, there are a lot of interesting features that can be added, as well as there are improvements to be made. For instance, one could extend the syntax of FstStudio programs with syntactic sugars such as 'replacement operators', something that is quite possible to express in the existing language, although a bit complex. Another improvement is to add more construction and minimization algorithms, so that the user can choose the construction method that is appropriate to his/her program.

Chapter 3

Appendix A: Simplification of regular expressions

3.1 Simplification

3.1.1 Union

Valid both for regular expressions and regular relations.

Regular expression	Simplified form	Description
$A\|\emptyset$	A	$\emptyset$ is the unit element of union.
$\emptyset\|A$	A	
$A\|?*$	?*	?* dominates every expression A.
$?*\|A$	?*	
$A\|A$	A	A union A equals A (no duplicates).
$A\ B\|A\ C$	$A\ [B\|C]$	Product is distributive over union.
$B\ A\|C\ A$	$[B\|C]\ A$	

3.1.2 Concatenation

Valid both for regular expressions and regular relations.

Regular expression	Simplified form	Description
$A\ \emptyset$	$\emptyset$	$\emptyset$ is the destructor of concatenation.
$\emptyset\ A$	$\emptyset$	
$A\ \epsilon$	A	ϵ is the unit element for concatenation.
$\epsilon\ A$	A	

3.1.3 Kleene's star

Valid both for regular expressions and regular relations.

Regular expression	Simplified form	Description
A^{**}	A^*	Multiple stars are redundant.
$\emptyset^*$	ϵ	ϵ is part of kleene's star.
ϵ^*	ϵ	ϵ is the unit element of concatenation.

3.1.4 Intersection

Valid for regular expressions.

Regular expression	Simplified form	Description
$A\&A$	A	
$A\&?^*$	A	?* is the unit element of intersection.
$?^*\&A$	A	
$\emptyset\&A$	$\emptyset$	$\emptyset$ is the destructor if intersection.
$A\&\emptyset$	$\emptyset$	

3.1.5 Complemenation

Valid for regular expressions.

Regular expression	Simplified form	Description
$\sim\emptyset$	?*	The complement of $\emptyset$ is ?*
$\sim ?^*$	$\emptyset$	The complement of ?* is $\emptyset$
$\sim\sim A$	A	Two complements cancels each other.

3.1.6 Cross product

Valid for regular relations.

Regular expression	Simplified form	Description
$\emptyset.x.A$	$\emptyset$	$\emptyset$ is the destructor of cross product.
$A.x.\emptyset$	$\emptyset$	

3.1.7 Composition

Valid for regular relations.

Regular expression	Simplified form	Description
$\emptyset.o.A$	$\emptyset$	$\emptyset$ is the destructor of composition.
$A.o.\emptyset$	$\emptyset$	

Chapter 4

Appendix B: Programmer's interface

Importing the module 'TransducerInterface.hs' gives you access to the following functions.

4.0.8 Regular expressions

Functions for constructing a simplified regular expression.

```
s          :: a -> Reg a                -- symbol
eps        :: Reg a                     -- epsilon
empty      :: Reg a                     -- empty set
allS       :: Reg a                     -- all symbol
star       :: Reg a -> Reg a            -- kleene's star
plus       :: Reg a -> Reg a            -- kleene's plus
complement :: Reg a -> Reg a            -- complement
(<|>)      :: Reg a -> Reg a -> Reg a -- union
(|>)       :: Reg a -> Reg a -> Reg a -- product
(<&>)      :: Reg a -> Reg a -> Reg a -- intersection
(<->)      :: Reg a -> Reg a -> Reg a -- set minus
symbols    :: Reg a -> a                -- collect all symbols.
```

4.0.9 Regular relations

Functions for constructing a simplified regular relation.

```
r          :: a -> a -> Reg a              -- relation
empty      :: RReg a                       -- empty set
idR        :: Reg a -> RReg a              -- identity
star       :: Reg a -> Reg a               -- kleene's star
plus       :: Reg a -> Reg a               -- kleene's plus
```

```
(<|>)      :: Reg a -> Reg a -> Reg a    -- union
(|>)       :: Reg a -> Reg a -> Reg a    -- product
(<*>)      :: Reg a -> Reg a -> RReg a   -- cross product
(<.>)      :: RReg a -> RReg a -> RReg a -- composition
symbols    :: RReg a -> a                -- collect all symbols.
```

4.0.10 Parsing regular relations

Functions for parsing regular relations.

'parseProgram' takes a string containing a fstStudio program, and try to parse it - if unsuccessful, it returns a error message. 'parseExp' parses a string containing a regular relation.

```
parseProgram :: String -> Either String (RReg String)
parseExp     :: String -> Either String (RReg String)
```

4.0.11 Construction and running

Functions for constructing and running a finite state transducer.

The function 'compile' construct a deterministic, epsilonfree, minimal transducer, and 'compileN' construct a epsilonfree, possibly non-deterministic, non-minimal transducer. The 'Sigma' type provides a way to add symbols that is not present in the regular relation. 'applyDown' and 'applyUp' are used to run the transducer.

```
type Sigma a = [a]

compile         :: Ord a => RReg a -> Sigma a -> Transducer a
compileN        :: Ord a => RReg a -> Sigma a -> Transducer a
determinize     :: Ord a => Transducer a -> Transducer a
minimize        :: Ord a => Transducer a -> Transducer a
unionT          :: Ord a => Transducer a -> Transducer a -> Transducer a
productT        :: Ord a => Transducer a -> Transducer a -> Transducer a
starT           :: Ord a => Transducer a -> Transducer a
compositionT    :: Ord a => Transducer a -> Transducer a -> Transducer a
emptyTransducer :: Transducer a
applyDown       :: Ord a => Transducer a -> [a] -> Maybe [[a]]
applyUp         :: Ord a => Transducer a -> [a] -> Maybe [[a]]
load            :: FilePath -> IO (Either String (Transducer String))
save            :: FilePath -> Transducer String -> IO (Either String ())
```

4.0.12 Transducer Information

Functions for getting information about a built transducer.

```
type State = Int

states              :: Transducer a -> [State]
isFinal             :: Transducer a -> State -> Bool
initial             :: Transducer a -> State
finals              :: Transducer a -> [State]
transitionsU        :: Transducer a -> (State,a) -> [(a,State)]
transitionsD        :: Transducer a -> (State,a) -> [(a,State)]
showTransducer      :: Transducer a -> String
numberOfStates      :: Transducer a -> Int
numberOfTransitions :: Transducer a -> Int
```

Chapter 5

Appendix C: Manual

5.1 FstStudio User Manual

5.1.1 Introduction

FstStudio (Finite State Transducer Studio) is a program for construction and running of finite state transducers. If you're not familiar with the concept of transducers and regular relations, then Xerox's fst homepage
http://www.xrce.xerox.com/research/mltt/fst/
is recommended as an introduction (primarily because the syntax of Xerox's fst program has functioned as an inspiration for the syntax of fstStudio).

5.1.2 Installation

If you have access to the binary file, then all you have to do is to copy the file to an appropriate library and run the program according to the instructions below. If you have access to the source code, compile the system by typing "make". This will construct a file named "fst" in the source library. To be able to compile the system, you must have ghc installed at your system. If you don't have it already, a free distribution of ghc can be downloaded at http://www.haskell.org/ghc/. It can also be run in Hugs. Load 'Main.hs' and type 'main'.

5.1.3 Syntax

fstStudio takes a program consisting of regular relations that denotes the relation between two regular languages and constructs a transducer. If a regular expression, not a relation, is given, then it is interpreted as the identity relation. The syntax is very similar to Xerox's finite state transducer syntax with two fundamental differences: a distinction is made between functions (definitions) and strings, and fststudio allows functional definitions.

"a"

symbol.
Example: ["b"] denotes the language {"b"}.

a

variable. A symbol without quotes is a variable.

"a":"b"

Describes a relation between the symbol a and b. This relation is ordered and a is said to be a part of the upper language and b is said to be part of the lower language.
Example: ["a":"b"] denotes the relation {("a","b")}.

0

epsilon symbol. The epsilon symbol denotes the string with no symbols.
Example: [0] denotes the language {""}.

?

all symbol. The all symbol denotes the union of all symbols in the alphabet.
Example: [?] and an alphabet {a, b, c} denotes the language {"a","b","c"}.

""

quotes cancels every special meaning of the symbols.
Example: ["? | 0"] denotes the language {"? | 0"}.

[A]

brackets are used to change the precedence of a regular relation.

(A)

parenthesis expresses optionality, and has the same meaning as $[A|0]$.

A B

Concatenation of the expressions or relations A and B.
Example: [[a | b] [c | d]] denotes the language {"ac", "ad", "bc", "bd"}

A ˆ n

Concatenation of A n times. Aˆ0 is defined as the empty string.
Example: [a]ˆ3 describes the language {"aaa"}.

A | B

Union of the languages or relations A and B.
Example: [a | b] describes the language {"a", "b"}.

A & B

Intersection of the languages A and B.
Example: [a | b] & [a] describes the language {"a"}.

A - B

Minus of the languages A and B, and has the same meaning as [A & $\sim B$]
Example: [a | b] - [a] describes the language {“b”}.

~A

Describes the complement of an expression, and has the same meaning as [?* - A]. Note that complement is always defined over an alphabet - the expression $\sim[A]$ is only unambiguous with respect to an alphabet.
Example: $\sim[a]$ denotes the language that doesn't contain the string “a”. If the alphabet is {a,b}, then $\sim[a]$ denotes the language {””, ”b”, ”aa”, ”ba”, ...}

A+

Repetition (Kleenes plus). A concatenated with itself an arbitrary number of times, including zero times.
Example: [a]+ denotes the infinite language {“a”,”aa”,”aaa”,...}

A*

Kleene's star: [A+ | 0].
Example: [a]* denotes the infinite language {””,”a”,”aa”,...}

$A

Containment. The set of strings where A appear at least once as a substring. Containment is the same thing as [?* A ?*]

A .x. B

Cross product of the languages A and B.
Example: [[a | b] .x. c] describes the relations {(”a”, ”c”), (“b”, ”c”)}.

A .o. B

Composition of the relations A and B.
Example: [a:b | c: d] .o. [d:e] describes the relation {(”c”,”e”)}.

The precedence of the operators is as follows, where 4 is the highest precedence:

4 ~ ˆ * + $

3 Concatenation

2 | & -

1 .x. .o.

A file containing a program must end with *.fst, and an input file must end with *.dat. A program is a collection of functions defining regular relations. A function with zero arguments is called a **definition** or a **macro**.

A definition, or a macro, can for example look like this:

```
<digits> ::= "1" | "2" | "3" | "4" | "5" |
             "6" | "7" | "8" | "9" | "0";
```

and a function can look like this:

```
<swap,a,b> ::= b a ;
```

Note that strings are marked with quotes, and variables have no quotes.

Every program must contain a main definition (a program without a main definition will result in a parse error).

```
<main> ::= ... ;
```

The alphabet of a program is the symbols in the regular relation defined in the program.

5.1.4 Example program

The example was found at http://www.xrce.xerox.com/research/mltt/fst/fsexamples.html. The program simulates a soda machine and takes a collection of symbols that symbolises different coins. If the amount of money applied to the machine corresponds to 65 cent, then the machine produces a PLONK (enough money has been put into the machine to get a soda).

```
<nickel>  ::= ["n" .x. "c"^5];
<dime>    ::= ["d" .x. "c"^10];
<quarter> ::= ["q" .x. "c"^25];
<cent>    ::= ["c" .x. "c"];
<money>   ::= [ <nickel> | <dime> | <quarter> | <cent>]*;
<drink>   ::= ["c"^65 .x. "PLONK"];
<main>    ::= [ <money> .o. <drink> ];
```

5.1.5 Running FstStudio

fstStudio can be run in two modes, batch mode and interactive mode.

Batch mode

fst

Starts fstStudio in interactive mode. See below.

fst FILE [Options]

Run fstStudio in batch mode. FILE must end with *.fst, which defines a FstStudio program, or *.net, which defines a saved transducer. If no options is given, then input is taken from standard input, the transducer is applied down, and the output, if any, is produced on standard output.

Options	Meaning
-u	apply the transducer up
-d	apply the transducer up
-i FILE	take input from FILE
-o FILE	write output to FILE

Interactive mode

Interactive mode is entered by typing the following command:

```
$ fstStudio <return>

****************************************************
* Welcome to Finite State Transducer Studio!       *
* Written purely in Haskell.                       *
* Version : 0.9                                    *
* Date    : 11 August 2001                         *
* Author  : Markus Forsberg                        *
* Please send bug reports/suggestions to:          *
* d97forma@dtek.chalmers.se                        *
****************************************************

Type 'h' for help.

>
```

List of commands

r REG

Read a regular relation from standard input. If a regular expression is typed, then it is interpreted as the identity relation.

b

Build a epsilonfree, deterministic, minimal transducer from a loaded/typed regular relation.

bn
: Build a epsilonfree, possibly non-deterministic, non-minimal transducer from a load/typed regular relation.

m
: Minimize a built transducer.

det
: Determinize a built transducer

s FILE
: Save to FILE. If FILE ends with *.net, then the built transducer is saved. Any other suffix, saves the produced output in the system to FILE, if any.

l FILE
: Load from FILE. FILE must end with *.fst, *.net or *.dat. If FILE ends with *.fst, then a FstStudio program is loaded into FstStudio. If FILE ends with *.net, then a transducer is loaded into FstStudio. If FILE ends with *.dat, then input is loaded into FstStudio.

l a|b
: Load and union two transducers. a and b must either be a file ending with *.net or the symbol *, which refers to the interior transducer. The produced transducer is possibly non-deterministic and non-minimal.

l a b
: Load and concatenate two transducers. a and b must either be a file ending with *.net or the symbol *, which refers to the interior transducer. The produced transducer is possibly non-deterministic and non-minimal.

l a *
: Load and apply Kleene's star on a transducer. a must either be a file ending with *.net or the symbol *, which refers to the interior transducer. The produced transducer is possibly non-deterministic and non-minimal.

l a .o. b
: Load and compose two transducers. a and b must either be a file ending with *.net or the symbol *, which refers to the interior transducer. The produced transducer is possibly non-deterministic and non-minimal.

vt
: View loaded/built transducer.

vr
: View loaded/typed regular relation.

vi
: View loaded input.

vo
: View produced output.

d
: Apply transducer down with loaded input.

u
: Apply transducer up with loaded input.

d SYMBOLS
: Apply tranducer down with SYMBOLS.

u SYMBOLS
: Apply transducer up with SYMBOLS.

c
: Clear memory.

h
: List commands.

q
: End session.

Example on FstStudio in interactive mode

First, save example program above to the file 'drink.fst'. Then start Fst-Studio in interactive mode.

```
>l drink.fst

Loaded a regular relation from drink.fst.

>b

Built a deterministic, minimal transducer with 26 states and 46 transitions.

>u PLONK

Input accepted. Type 'vo' to view outputs.
```

```
>vo

...
q n n d d d
q n n d d n n
q n n d n n d
q n n d n n n n
q n n d n d n
q d n q
q d n n d d
q d n n d n n
q d n n n n d
q d n n n n n n
q d n n n d n
q d n d n d
q d n d n n n
q d n d d n
q d q n
q d d d d
q d d d n n
q d d n n d
q d d n n n n
...

>s drink.out

Saved outputs to file drink.out.

>q

Do you really want to quit? (y):y

Session ended.
```

In 'drink.out' we now have all possible ways to feed the soda machine with coins, with regard to order, in such a way that we get a soda. There are 634 different ways.

Bibliography

[1] Antimirov, Valentin M., *Partial Derivatives of Regular Expressions and Finite Automata Constructions.*

[2] Antimirov, Valentin M., Mosses, Peter D. *Rewriting Extended Regular Expressions.* Nov 1993

[3] Chang, Chia-Hsiang. Paige, Robert. *From Regular Expressions to DFA's Using Compressed NFA's.* Computer Science Department, Courant Institute of Mathematical Sciences, New York University, May 1996

[4] Hagenah, Christian. Muscholl, Anca. *Computing epsilon-free NFA from Regular Expressions in O(n log2(n)) Time.* Institut fur Informatik, Universitt Stuttgart, Breitwieserstr.

[5] Hopcroft, John E. Ullman, Jeffery D. *Introduction to automata theory, languages, and computation.* Addison-Wesley Publishing Co., Reading, M.A.,1979.

[6] Kaplan, Ronald M., Kay, Martin, *Regular Models of Phonological Rule Systems.* Sept. 1994

[7] Karttunen, L., Chanod, J.P., Grefenstette, G., Schille, A., *Regular expressions for language engineering*, Natural Language Engineering 2 (4) 305-328, Cambridge University Press, 1996

[8] Karttunen, Lauri. *Directed Replacement.* ACL 1996

[9] Kozen, Dexter C. *Automata and Computability* Springer-Verlag New York, Inc,1997.

[10] Martin, John C. *Introduction to languages and the theory of computation* second edition. The McGraw-Hill Companies, Inc. 1997

[11] Thompson, Simon. *Regular Expressions and Automata using Haskell*, Computing Laboratory, University of Kent at Canterbury, Jan. 2000

[12] Van Noord, Gertjan, Gerdemann, Dale. *An Extendible Regular Expression Compiler for Finite-state Approaches in Natural Language Processing.*

[13] Van Noord, Gertjan, Gerdemann, Dale. *Transducers from Rewrite Rules with Backreferences.* April 1999

[14] Watson, Bruce W. *A taxonomy of finite automata construction algorithms.* Faculty of Mathematics and Computing Science, Eindhoven University of Technology. May 1994

[15] Watson, Bruce W. *A taxonomy of finite automata minimization algorithms.* Faculty of Mathematics and Computing Science, Eindhoven University of Technology. May 1994