UPPSALA
UNIVERSITET

IT 14 002
Examensarbete 15 hp
Januari 2014

Formalizing the Kleene Star for Square Matrices

Teo Asplund

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Formalizing the Kleene Star for Square Matrices

Teo Asplund

This thesis gives a formal description of the Kleene star for square matrices over a Kleene algebra. It builds on previous work on Kleene algebras and their formal description in Isabelle/HOL, and is a step toward a formal proof that square matrices over a Kleene algebra form a Kleene algebra.

Handledare: Tjark Weber
Ämnesgranskare: Joachim Parrow
Examinator: Olle Gällmo
IT 14 002
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

A Kleene algebra [1] is an algebraic structure that can be used to model a variety of problems from areas such as program analysis [2], formal languages [3], automata theory [1], graph theory [4] etc. One example of a Kleene algebra is the set of regular languages over some alphabet together with three operators (concatenation, union, and star). The development of Kleene algebra was motivated by the desire to have an algebraic structure that could be used to derive all equations among regular expressions. In Section 2 we discuss this further.

In this thesis we will give a formal description of the Kleene star for a particular Kleene algebra, namely the Kleene algebra where the elements are square matrices. In this model one may consider matrices as automata, in which case the star operator gives, for each pair of states, the set of inputs that take one state to the other. In Section 3 we discuss this model in more detail and take a look at two problems that the model can be applied to: firstly, the all-pairs shortest path problem (find the shortest path between all pairs of nodes in a graph) [4], for which an efficient algorithm is known and may be motivated by applying the model; and secondly, a proof of completeness of the axioms. This completeness proof is presented in [1] and states that if two regular expressions describe the same language, then one may derive equivalence of the expressions using the axioms of Kleene algebra given in Section 2.2.

Proof assistants are software tools that assist in developing formal proofs by means of an interactive proof editor that can semi-automatically perform proofs in collaboration with a human. Such programs are useful, since they may ensure the correctness of a proof, or generate proofs that could not be done by hand because of their length (such as proofs of programs). In Section 4 we take a closer look at the proof assistant *Isabelle* [5], which has been used in this project.

A lot of work has already been done to formalize Kleene algebra and the specific model of square matrices over a Kleene algebra using proof assistants. In [6] a complete formalization of the model is given in the proof assistant *Coq*. This is then used to provide a tactic for deciding Kleene algebra equalities. In [7] a range of variants of Kleene algebras and some of their models are formalized. This includes an incomplete formalization of the model of square matrices over a Kleene algebra using a different approach than [6]. In this thesis we continue this work by defining the Kleene star for the model as specified in [7], which only lacks the star operation. Thus, the main result of this project is an extension of [7], that results in a full description of the square matrix model of Kleene algebras. The details of this are the subject of Section 5.

2 Kleene Algebras and Regular Languages

2.1 Notation and terminology.

A *monoid* over a set S is a tuple $(S, \oplus, e)$ where $\oplus$ is a binary operation, satisfying the following axioms [8]:

$$\forall a, b \in S : a \oplus b \in S$$

$$\forall a, b, c \in S : (a \oplus b) \oplus c = a \oplus (b \oplus c)$$

$$\exists e \in S : \forall a \in S : e \oplus a = a \oplus e = a$$

That is, S is closed under the binary operation, $\oplus$ is associative, and there exists an identity element $e \in S$ with respect to $\oplus$.

A binary operation $\oplus$ is called *commutative* if

$$\forall x, y \in S : x \oplus y = y \oplus x$$

and *idempotent* if

$$\forall x \in S : x \oplus x = x.$$

A *semiring* over a set R is a tuple $(R, +, \cdot, 0, 1)$, where $+$ and $\cdot$ are two binary operations such that

- $(R, +)$ is a commutative monoid with identity element 0.
- $(R, \cdot)$ is a monoid with identity element 1.
- $\cdot$ distributes over $+$, i.e. $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$
- $0 \cdot a = a \cdot 0 = 0$

A semiring is called idempotent if the binary relation $+$ is idempotent. An idempotent semiring may also be called a *dioid*.

2.2 Kleene Algebras

A Kleene algebra [1] over a set K is an algebraic structure $\mathcal{K} = (K, +, \cdot, ^*, 0, 1)$ satisfying the following axioms:

$$a + (b + c) = (a + b) + c \tag{1}$$
$$a + b = b + a \tag{2}$$
$$a + 0 = a \tag{3}$$
$$a + a = a \tag{4}$$
$$a(bc) = (ab)c \tag{5}$$
$$1a = a \tag{6}$$
$$a1 = a \tag{7}$$
$$a(b + c) = ab + ac \tag{8}$$
$$(a + b)c = ac + bc \tag{9}$$
$$0a = 0 \tag{10}$$
$$a0 = 0 \tag{11}$$
$$1 + aa^* \leq a^* \tag{12}$$
$$1 + a^*a \leq a^* \tag{13}$$

$$b + ax \leq x \Rightarrow a^*b \leq x \tag{14}$$

$$b + xa \leq x \Rightarrow ba^* \leq x \tag{15}$$

where $a \leq b$ is defined as

$$a \leq b \iff a + b = b$$

where a, b, c and x are any elements in K. By axioms (1) - (11), $(K, +, 0)$ is an idempotent commutative monoid, $(K, \cdot, 1)$ is a monoid, and $(K, +, \cdot, 0, 1)$ is a dioid.

The remaining axioms, (12) - (15), deal with the *-operation and describe the fact that the behaviour of * is like that of the Kleene star operator of formal language theory.

2.3 Regular Languages

The set of regular languages over a finite alphabet Σ is denoted by $\mathbf{Reg}_\Sigma$ and defined as follows:

- $\emptyset$, i.e. the empty set, is a regular language.
- $\{\varepsilon\}$, which denotes the set containing only the empty string, is a regular language.
- $\{a\}$, where $a \in \Sigma$, is a regular language.
- If A and B are regular languages over Σ, then $A \cup B$, $A \circ B$, and A^* are regular languages.
- No other languages over Σ are regular.

Here $A \cup B$ is simply the union of the two sets, $A \circ B$ is the concatenation of the sets, i.e. $A \circ B = \{\alpha\beta \mid \alpha \in A \wedge \beta \in B\}$ and A^* denotes the smallest superset of A that contains ε and is closed under concatenation of its elements. For example, $\{0, 1\}^*$ is the set of all binary strings.

For any given finite alphabet Σ, $(\mathbf{Reg}_\Sigma, \cup, \circ, ^*, \emptyset, \{\varepsilon\})$ forms a Kleene algebra [1].

3 Square Matrices over Kleene Algebras

There is a natural way of defining a Kleene algebra $\mathcal{K} = (M_n, +, \cdot, ^*, \mathbf{0}, \mathbf{1})$ over M_n, where M_n is the set of all $n \times n$-matrices with elements in some Kleene algebra K. To do this, simply define $+$ as the usual matrix addition, $\cdot$ as standard matrix multiplication, $\mathbf{0}$ as the zero matrix and $\mathbf{1}$ as the identity matrix.

As for the star operator, in the case of 1×1-matrices the definition is trivial, namely, $(a)^* = (a^*)$. For larger matrices, consider the following automaton [9]

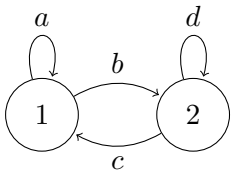


Figure 1: An automaton with two states over the alphabet $\Sigma = \{a, b, c, d\}$.

and the matrix, $E = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, where a, b, c and d are elements of a Kleene algebra. For each pair u, v of states in the automaton depicted in Figure 1, consider the set of inputs that takes state u to state v. These sets can be represented by the following regular expressions:

$$\begin{aligned} 1 \longrightarrow 1 &: (a + bd^*c)^* \\ 1 \longrightarrow 2 &: (a + bd^*c)^*bd^* \\ 2 \longrightarrow 1 &: d^*c(a + bd^*c)^* \\ 2 \longrightarrow 2 &: d^* + d^*c(a + bd^*c)^*bd^* \end{aligned}$$

Let $f = a + bd^*c$, then we define:

$$E^* = \begin{bmatrix} f^* & f^*bd^* \\ d^*cf^* & d^* + d^*cf^*bd^* \end{bmatrix}$$

If E is of a larger dimension, say $n \times n$, where $n > 2$, then we define the star operation by partitioning E into submatrices A, B, C and D, such that A and D are square.

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$$

Let $F = A + BD^*C$ and define

$$E^* = \left[\begin{array}{c|c} F^* & F^*BD^* \\ \hline D^*CF^* & D^* + D^*CF^*BD^* \end{array}\right] \tag{16}$$

With this definition E^* satisfies the axioms concerning the Kleene star [1].

This means that square matrices over a Kleene algebra themselves form a Kleene algebra together with the above operations.

3.1 Applications

In this section we take a look at two applications of Kleene algebras with square matrices as elements. We first examine a practical use (namely solving the all-pairs shortest path problem) and then a more theoretically useful application (the completeness problem).

3.1.1 All-Pairs Shortest Path Problem

The all-pairs shortest path problem consists of finding the shortest path between two nodes in a graph for *every pair* of nodes. To solve this problem we will use matrices over a model of Kleene algebra called the *tropical* algebra (or sometimes the *min-plus* algebra). The tropical Kleene algebra is a tuple $(\mathbb{R}_{\geq 0} \cup \{\infty\}, +, \cdot, ^*, \infty, 0)$, where

- $a + b = \min(a, b)$
- $a \cdot b = a +_{\mathbb{R}} b$, where $+_{\mathbb{R}}$ is the normal addition for reals.
- $a^* = 0_{\mathbb{R}}$, where $0_{\mathbb{R}}$ is the zero of the reals.
- ∞, is the additive identity.
- $0_{\mathbb{R}}$ is the multiplicative identity.

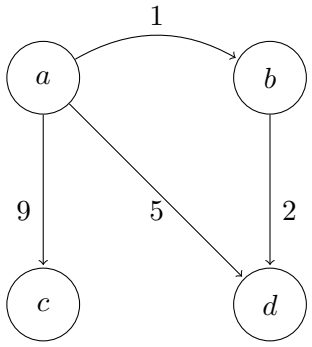
Since this forms a Kleene algebra [4], square matrices over a tropical Kleene algebra also form a Kleene algebra.

Given a directed graph $G = (E, V)$, define the adjacency matrix X as follows:

$$X_{i,j} = \begin{cases} 0 & \text{if} \quad i = j \\ c(v_i, v_j) & \text{if}(v_i, v_j) \in E \\ \infty & \text{otherwise} \end{cases}$$

where $c(v_i, v_j)$ is the length of the edge $(v_i, v_j) \in E$. It then turns out that taking X^* gives the solution to the all-pairs shortest path problem! [4]

For example, if we have the following directed graph:



then, the adjacency matrix is

$$X = \begin{bmatrix} 0 & 1 & 9 & 5 \\ \infty & 0 & \infty & 2 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{bmatrix}$$

then, if

$$A = \begin{bmatrix} 0 & 1 \\ \infty & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 9 & 5 \\ \infty & 2 \end{bmatrix}, \quad C = \begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix}, \quad \text{and} \quad D = \begin{bmatrix} 0 & \infty \\ \infty & 0 \end{bmatrix}$$

we have that

$$D^* = \begin{bmatrix} (0 + \infty \cdot 0^* \cdot \infty)^* & (0 + \infty \cdot 0^* \cdot \infty)^* \cdot \infty \cdot 0^* \\ 0^* \cdot \infty \cdot (0 + \infty \cdot 0^* \cdot \infty)^* & 0^* + 0^* \cdot \infty \cdot (0 + \infty \cdot 0^* \cdot \infty)^* \cdot \infty \cdot 0^* \end{bmatrix} = \begin{bmatrix} 0 & \infty \\ \infty & 0 \end{bmatrix}$$

so

$$F = A + BD^*C = \begin{bmatrix} 0 & 1 \\ \infty & 0 \end{bmatrix} + \begin{bmatrix} 9 & 5 \\ \infty & 2 \end{bmatrix} \begin{bmatrix} 0 & \infty \\ \infty & 0 \end{bmatrix} \begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix} =$$
$$= \begin{bmatrix} 0 & 1 \\ \infty & 0 \end{bmatrix} + \begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \infty & 0 \end{bmatrix}$$

thus

$$F^* = \begin{bmatrix} 0 & 1 \\ \infty & 0 \end{bmatrix}, \quad F^*BD^* = \begin{bmatrix} 9 & 3 \\ \infty & 2 \end{bmatrix}$$

$$D^*CF^* = \begin{bmatrix} \infty & \infty \\ \infty & \infty \end{bmatrix}, \quad D^* + D^*CF^*BD^* = \begin{bmatrix} 0 & \infty \\ \infty & 0 \end{bmatrix}$$

and therefore

$$X^* = \left[\begin{array}{cc|cc} 0 & 1 & 9 & 3 \\ \infty & 0 & \infty & 2 \\ \hline \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{array}\right]$$

3.1.2 Completeness

The following completeness theorem for Kleene algebras can be proved using square matrices over a Kleene algebra [1]:

Theorem 3.1 *Let α and β be two regular expressions over Σ denoting the same language, then $\alpha = \beta$ is a theorem of Kleene algebra.*

This means that we may use Kleene algebra to reason formally about regular languages. Furthermore it implies that one may use finite automata to solve problems in any model of a Kleene algebra.

The main idea of the proof stems from the fact that any finite automaton $\mathcal{A}$ can be represented by a triple $\mathcal{A} = (s, A, f)$, where $s, f \in \{0,1\}^n$ represent the *start* and *final* states respectively, where $s(i) = 1$ if i is the index of a start state (0 otherwise) and similarly if $f(i) = 1$, then i is the index of a final state. The A matrix is a square $n \times n$-matrix over a Kleene algebra $\mathcal{K}$ and is called the *transition matrix*.

For example, the automaton $\mathcal{A}$ in Figure 2 may be represented by the triple

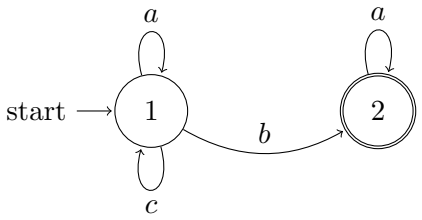


Figure 2: The automaton $\mathcal{A}$ with two states over the alphabet $\Sigma = \{a, b, c\}$.

$$\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} a+c & b \\ 0 & a \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right)$$

The language accepted by an automaton (s, A, f), is the element

$$s^T A^* f \in K$$

so, for the example automaton in Figure 2 we can see that the accepted language is

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} a+c & b \\ 0 & a \end{bmatrix}^* \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} (a+c)^* & (a+c)^* b a^* \\ 0 & a^* \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} =$$

$$= (a+c)^* b a^*$$

The proof in [1] then, step by step, goes about showing that for every regular expression there is an equivalent minimal deterministic finite automaton (s, A, f), which then in turn implies the theorem. The details are left out of this thesis. See [1] for the complete proof.

4 Isabelle

One example of a proof assistant is Isabelle/HOL (which is the specialization of the generic proof assistant Isabelle for higher-order logic, HOL), using which the structure of Kleene algebras has already been formalized. In this section a brief introduction to Isabelle/HOL is presented. See [5] for a more thorough look at the system.

4.1 Proof Assistants

A *proof assistant* is a piece of software that produces formal proofs in collaboration with a human. The proof assistant Isabelle/HOL is a specification and verification system, that is the specialization of Isabelle (a generic system for implementing logical formalisms) for HOL. HOL, which stands for Higher-Order Logic, is a system for expressing mathematical concepts.

4.2 Theories

A *theory* is more or less a named container for types, functions and theorems. A theory can be based on any number of other theories that are used when defining the new concepts. The implementation of the Kleene star for square matrices, which this thesis is concerned with, builds on the collection of theories that can be found at [7], which make up the formal description of Kleene algebras.

4.3 λ-Abstractions

Lambda calculus [10] is a formal system for expressing computation. It is based on the *lambda abstraction*, $\lambda x.t$, which defines an anonymous function on x that maps to $t(x)$. For example, $\lambda x.x^2$, is the anonymous function that takes one value and returns its square. A lambda abstraction only operates on a single variable, but with currying one may rewrite functions on several variables as lambda abstractions, since functions are acceptable as return values.

In Isabelle one may create anonymous functions that take several variables as input by $\lambda x_1, \ldots, x_n.t$, where t depends on $x_1, \ldots, x_n$.

4.4 Types

In Isabelle there are:

- *base types*, which are predefined types available in Isabelle. For example, *bool*, *nat* and *int*.
- *type constructors*, which construct new types from previously defined types. For example the *list* constructor, which takes a type as argument, say *nat*, and generates a new type *nat list*, which is the type of lists with natural numbers as elements.
- *type variables*, which are written as α, β (or $'a$, $'b$), etc. and denote mathematical variables that range over types.
- *function types*, which are written as $\alpha \Rightarrow \beta$ and denote a *total* function that takes elements of type α and returns elements of type β.

One may define new datatypes using the **datatype** keyword:

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\ t = C_1\ \tau_{11}\ \ldots\ \tau_{1k_1}\ |\ \ldots\ |\ C_m\ \tau_{m1}\ \ldots\ \tau_{mk_m}$$

where $\alpha_1, \ldots, \alpha_n$ are distinct type variables, $C_1, \ldots, C_m$ are distinct constructor names, τ_{ij} are types, and with some additional restrictions [11]. As an example, the list type could be defined as

$$\begin{aligned}\textbf{datatype}\ 'a\ list\ &=\ Nil\\ &|\ Cons\ 'a\ "'a\ list"\end{aligned}$$

where the constructor Nil gives the empty list, and $Cons$ adds an element to the front of a list, e.g. $Cons\ 1\ (Cons\ 2\ Nil)$ is a value of type *nat list* (the list containing the elements 1 and 2).

Sometimes Isabelle will be able to infer the type of some term, but other times it is necessary to constrain a term t to some type τ. The syntax for this is $t :: \tau$.

Another way of defining new types is with the **typedef** keyword, which essentially turns a non-empty subset of an existing type into a new type. For example, the subset $\{0, 1, 2, \ldots, k\}$ of *nat*, may be specified as a type by writing

$$\begin{aligned}&\textbf{typedef}\ kSet\ =\ "\{..<k :: nat\}"\\ &\textbf{apply}\ (rule\_tac\ x\ =\ 0\ in\ exI)\\ &\textbf{by simp}\end{aligned}$$

The second and third line simply deal with proving that the type is non-empty. This introduces the new type kSet and asserts that it is a copy of the set {0,1,2,..., k-1}. This is expressed by a bijection between the type $kSet$ and the set $\{0, 1, 2, \ldots, k-1\}$. Thus, the following functions are declared automatically:

$$Rep\_kSet :: kSet \Rightarrow nat$$

$$Abs\_kSet :: nat \Rightarrow kSet$$

where *Rep_kSet* is surjective on the subset and *Rep_kSet* and *Abs_kSet* are inverses of each other.

4.5 Type Classes

The idea of *type classes* originally comes from the Haskell language where the purpose was to allow for elegant implementation of overloading [12]. A typical example would be a polymorphic equality function (see [13]) $eq :: \alpha \Rightarrow \alpha \Rightarrow bool$, which is overloaded on α. This is achieved by creating a type class *eq* and using instance declarations for different types that support the *eq*-function.

We may do something similar in Isabelle, for example we may define the class *semigroup* which introduces an associative binary operation $\otimes$ (a semigroup is in fact a set with such an operator) by writing:

class *semigroup* =
  fixes *mult* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\otimes$ 70)
  assumes *assoc* : $(x \otimes y) \otimes z = x \otimes (y \otimes z)$

This specification consists of two parts, namely the *operational* part (**fixes**), which specifies the class parameters, and the *logical* part (**assumes**), which specifies properties on the parameters. The **infixl** keyword simply specifies that the *mult* operator may be written as the infix notation $\otimes$, which associates to the left and has precedence level 70. This determines the order of operations. One may then make a type an instance of *semigroup* by defining $\otimes$ for that type and proving that *assoc* holds. For example, for *int*, this is written

instantiation *int* :: *semigroup*
begin

definition
  mult-int-def : $i \otimes j = i + (j :: int)$

instance proof
...
qed

end

Here the required $\otimes$ operator is defined for *int* in the **definition** clause. Then a proof is presented that shows the definition to satisfy the class specification. After this is done, *int* will be considered a *semigroup* automatically.

Previously specified classes can be subclassed by extending them with additional parameters and properties. For example, we might extend *semigroup* with a parameter *neutral*:

$$\begin{aligned}&\textbf{class } monoidl \ = \ semigroup + \\ &\quad \textbf{fixes } neutral :: \ 'a \ (\mathbf{1}) \\ &\quad \textbf{assumes } neutl : \ \mathbf{1} \otimes x = x\end{aligned}$$

this subclass specifies a monoid, except the **1** only acts as a neutral element from the left. One may then, of course, subclass *monoidl* to create a class *monoid*

$$\begin{aligned}&\textbf{class } monoid \ = \ monoidl + \\ &\quad \textbf{assumes } neut : \ x \otimes \mathbf{1} = x\end{aligned}$$

Types can then be made instances of these classes by defining how the properties specified in the class correspond to the the type and proving that the definition satisfies the class specification, as we have seen above in the *semigroup* example.

In Isabelle the following notation is used

$$x \ :: \ 'a \ :: \ c$$

to specify that x is of type $'a$ which is of class c.

4.6 Formalization of Kleene Algebra

We have seen above that one may specify type classes in Isabelle and then show that different types are instances of this class. In [7] the type class *dioid* is arrived at by starting with a quite general structure called a *join semilattice*, which is specified in Isabelle and then expanded over the course of several steps to a *dioid* class, using the type class system.

A Kleene algebra is a dioid with an extra operator, * (as noted in Section 2.2). Thus, the specification is well underway when the structure of a *dioid* is specified. The remaining work to specify the Kleene algebra consists of expanding the *dioid* with a star operator that satisfies some of the axioms given in 2.2 to create variants of dioids that are closer to Kleene algebras (i.e. satisfy more of the axioms). This leads to variants of Kleene algebras where some of the axioms are weaker and finally to a specification of the Kleene algebras presented in [1] that this thesis is concerned with.

4.7 Matrix and SqMatrix

In [7] matrices are essentially anonymous functions that take two numbers i, j as arguments where $i \in \{0, 1, \ldots, m\}$ and $j \in \{0, 1, \ldots, n\}$ such that the value of the function at i, j is the value of the element at position $i - 1$, $j - 1$ in the matrix (since we start counting from 1, per usual, in this thesis, but the Isabelle implementation starts from 0). Square matrices are essentially the same, but i and j are in the same range $\{0, 1, \ldots, m\}$. In Isabelle this may be written as

$$\textbf{datatype } ('a, \ 'm) \ sqmatrix = SqMatrix \ 'm \ atMost \Rightarrow \ 'm \ atMost \ \Rightarrow \ 'a.$$

where $atMost$ is defined as the type

typedef $'a\ atMost\ =\ "\{..<len\_of\ TYPE('a :: len)\}"$
by auto

The class $len0$ specifies that a function len_of, that returns a value of type nat, is defined on any type that is an instance of $len0$. The class len is a subclass of $len0$ and specifies that the value of len_of must be non-zero. The purpose of $TYPE$ is essentially to make a type into a term. This boils down to $len\_of\ TYPE('a :: len)$ giving the number of rows and columns in the square matrix.

5 Formal Description

5.1 Idea

When formalizing the Kleene star for square matrices we will make use of the previously seen definition (16) and the Kleene algebra theory for Isabelle/HOL described in the previous section. When defining the star operation for square matrices, we would like to use a recursive function that takes as input a square matrix and splits its input into four submatrices over which the function then recurses. The problem with this approach is that splitting the matrix would mean changing its type, and it is not possible to define a recursive function over types in Isabelle/HOL.

To get around this, we will instead define a function that is called recursively on a square matrix, each call with an altered matrix of the *same* dimensions. We will make use of some properties of matrix multiplication and the 0 element of a Kleene algebra. How this may be achieved is explored in sections 5.2 and 5.4.

For a square $n \times n$-matrix

$$E = \begin{bmatrix} e_{1,1} & \cdots & e_{1,n} \\ \vdots & \ddots & \vdots \\ e_{n,1} & \cdots & e_{n,n} \end{bmatrix}$$

recall the definition (16) of the star operator applied to A. If we choose the dimensions of the submatrices A and D to be $(n-1) \times (n-1)$ and 1×1 respectively, we see that B will be a single column of length $n-1$ and C will be a single row of length $n-1$ ([1] proves that the definition of * is independent of the partition chosen):

$$A = \begin{bmatrix} e_{1,1} & \cdots & e_{1,n-1} \\ \vdots & \ddots & \vdots \\ e_{n-1,1} & \cdots & e_{n-1,n-1} \end{bmatrix} \qquad B = \begin{bmatrix} e_{1,n} \\ \vdots \\ e_{n-1,n} \end{bmatrix}$$

$$C = \begin{bmatrix} e_{n,1} & \cdots & e_{n,n-1} \end{bmatrix} \qquad D = \begin{bmatrix} e_{n,n} \end{bmatrix}$$

By the definition of F, we see that the recursive function we want to define needs to multiply these. Below we shall see that we may in fact pad the submatrices A, B, C, and D, so that they are all $n \times n$-matrices and still get the desired result as the product of the

padded matrices, except it will be padded with zeros as well, to maintain the dimensions $n \times n$. We shall define some help functions that assist with this padding process. We will then reapply the recursive function to this padded F to find F^*, but we need to take into account that the padded F has the wrong dimensions. Therefore we also pass an argument that specifies the dimensions of the unpadded F as a natural number. This number will decrease each time the function is applied, starting at n going down to 2 (in fact starting at $n-1$ and going down to 1, since the matrix implementation in Isabelle starts indexing with 0), at which point a base case has been reached and the function can return.

5.2 Preliminaries

We introduce the following notation:

- $M_{p\times q}$ denotes a matrix M with p rows and q columns.
- $\mathbf{v}_n$ denotes a vector $\mathbf{v}$ of length n.
- $M_{i,j}$ denotes the element in row i and column j.
- $M_{i,*}$ denotes the i:th row as a vector.
- $M_{*,j}$ denotes the j:th column as a vector.
- $\mathbf{0}_n$ denotes the vector of length n that contains only 0. Sometimes the index is left out, when the meaning is clear from the context.
- $\mathbf{e}_{n,i}$ denotes the vector of length n whose elements are all 0, except the element at position i, which is 1.

Using axioms (1 - 11) it is easily seen that the product $c\mathbf{e}_{n,i}$ is the vector that is 0 at every position except at i, where it is c ($0, 1$ and c are elements in some Kleene algebra K).

We will make use of a few properties of matrix multiplication. For two square matrices

$$A^p = \left[\begin{array}{c|c|c} 0 & \begin{matrix} a_1 \\ \vdots \\ a_h \end{matrix} & 0 \\ \hline & 0 & \\ \hline 0 & \begin{matrix} \vdots \\ 0 \end{matrix} & 0 \end{array}\right]_{n\times n} \qquad B^p = \left[\begin{array}{ccc|c|ccc} & & & 0 & & & \\ & 0 & & \vdots & & 0 & \\ & & & 0 & & & \\ \hline 0 & \cdots & 0 & b & 0 & \cdots & 0 \\ \hline & & & 0 & & & \\ & 0 & & \vdots & & 0 & \\ & & & 0 & & & \end{array}\right]_{n\times n}$$

such that

$$A^p_{*,j} = \begin{cases} (\ a_1 \quad \cdots \quad a_h \quad 0 \quad \cdots \quad 0\ )_n & \text{if} \quad j = p \\ \mathbf{0_n} & \text{otherwise} \end{cases}$$

and

$$B^p_{i,j} = \begin{cases} b & \text{if} \quad i = j = p \\ 0 & \text{otherwise} \end{cases}$$

the product A^pB^p is

$$A^pB^p = \left[\begin{array}{c|c|c} \multirow{2}{*}{0} & a_1 b & \multirow{2}{*}{0} \\ & \vdots & \\ & a_h b & \\ \hline \multirow{2}{*}{0} & 0 & \multirow{2}{*}{0} \\ & \vdots & \\ & 0 & \end{array}\right]_{n\times n} \tag{17}$$

since for an element $(A^pB^p)_{i,j}$ in A^pB^p, we have

$$(A^pB^p)_{i,j} = A^p_{i,*} \cdot B^p_{*,j},$$

where $\cdot$ denotes the dot product, and thus

$$A^p_{i,*} = \begin{cases} a_i \mathbf{e}_{n,p} & \text{if} \quad i \leq h \\ \mathbf{0}_n & \text{otherwise} \end{cases} \qquad B^p_{*,j} = \begin{cases} b\mathbf{e}_{n,p} & \text{if } j = p \\ \mathbf{0}_n & \text{otherwise} \end{cases}$$

so if $j \neq p$, then

$$(A^pB^p)_{i,j} = A^p_{i,*} \cdot B^p_{*,j} = A^p_{i,*} \cdot \mathbf{0} = 0.$$

Similarly, if $i > h$, then

$$(A^pB^p)_{i,j} = A^p_{i,*} \cdot B^p_{*,j} = \mathbf{0} \cdot B^p_{*,j} = 0.$$

If, however, $i \leq h$ and $j = p$, then

$$(A^pB^p)_{i,n} = A_{i,*} \cdot B_{*,j} = a_i \mathbf{e}_{n,p} \cdot b\mathbf{e}_{n,p} = 0 + \cdots + 0 + a_i b + 0 \cdots + 0 = a_i b$$

thus

$$(A^pB^p)_{i,j} = \begin{cases} a_i b & \text{if } i \leq h \wedge j = p \\ 0 & \text{otherwise} \end{cases}$$

defines the matrix product (17) of A^p and B^p. Note that for the two matrices

$$A = \begin{bmatrix} a_1 \\ \vdots \\ a_h \end{bmatrix}_{h\times 1} \quad \text{and} \quad B = \begin{bmatrix} b \end{bmatrix}_{1\times 1} \quad \text{their product is } AB = \begin{bmatrix} a_1 b \\ \vdots \\ a_h b \end{bmatrix},$$

so

$$A^p = \left[\begin{array}{c|c|c} 0 & A & 0 \\ \hline 0 & \begin{matrix}0\\ \vdots \\ 0\end{matrix} & 0 \end{array}\right]_{n\times n} \qquad B^p = \left[\begin{array}{c|c|c} 0 & \begin{matrix}0\\ \vdots \\ 0\end{matrix} & 0 \\ \hline 0 \;\cdots\; 0 & B & 0 \;\cdots\; 0 \\ \hline 0 & \begin{matrix}0\\ \vdots \\ 0\end{matrix} & 0 \end{array}\right]_{n\times n}$$

and

$$A^pB^p = \left[\begin{array}{c|c|c} 0 & AB & 0 \\ \hline 0 & \begin{matrix}0\\ \vdots \\ 0\end{matrix} & 0 \end{array}\right]_{n\times n} \tag{18}$$

Similarly, for the same matrix B^p and matrices C and C^p,

$$C = \begin{bmatrix} c_1 & \cdots & c_w \end{bmatrix}_{1\times w} \qquad C^p = \left[\begin{array}{c|c} 0 & 0 \\ \hline C & 0 \;\cdots\; 0 \\ \hline 0 & 0 \end{array}\right]_{n\times n}$$

such that

$$C^p_{i,*} = \begin{cases} (\; c_1 \;\cdots\; c_w \;\; 0 \;\cdots\; 0 \;)_n & \text{if} \quad i = p \\ \mathbf{0}_n & \text{otherwise} \end{cases}$$

or equivalently

$$C^p_{*,j} = \begin{cases} c_j\mathbf{e}_{n,p} & \text{if} \quad j \le w \\ \mathbf{0}_n & \text{otherwise} \end{cases}$$

we may use similar reasoning to conclude that

$$B^pC^p = \left[\begin{array}{c|cccc} 0 & & 0 & \\ \hline BC & 0 & \cdots & 0 \\ \hline 0 & & 0 & \end{array}\right]_{n\times n} \tag{19}$$

where

$$BC = \left[\begin{array}{ccc} bc_1 & \cdots & bc_w \end{array}\right].$$

We will also make use of the fact that for the two square matrices A^p and C^p, in the case where $h = w = p-1$, we have that

$$A^pC^p = \left[\begin{array}{ccc|c} a_1c_1 & \cdots & a_1c_{p-1} & \\ \vdots & \ddots & \vdots & 0 \\ a_{p-1}c_1 & \cdots & a_{p-1}c_{p-1} & \\ \hline & 0 & & 0 \end{array}\right]_{n\times n} = \tag{20}$$

$$= \left[\begin{array}{c|c} AC_{(p-1)\times(p-1)} & 0 \\ \hline 0 & 0 \end{array}\right]_{n\times n}$$

since if $i > p-1$, then

$$(A^pC^p)_{i,j} = A^p_{i,*} \cdot C^p_{*,j} = \mathbf{0} \cdot C^p_{*,j} = 0$$

and if $j > p-1$, then

$$(A^pC^p)_{i,j} = A^p_{i,*} \cdot C^p_{*,j} = A^p_{i,*} \cdot \mathbf{0} = 0,$$

but if $i \leq p-1$ and $j \leq p-1$, then

$$(A^pC^p)_{i,j} = A^p_{i,*} \cdot C^p_{*,j} = a_i\mathbf{e}_{n,p} \cdot c_j\mathbf{e}_{n,p} = 0 + \cdots + 0 + a_ic_j + 0 + \cdots + 0 = a_ic_j.$$

By similar reasoning we have that for matrices D and D^p

$$D = \begin{bmatrix} d_{1,1} & \cdots & d_{1,p-1} \\ \vdots & \ddots & \vdots \\ d_{p-1,1} & \cdots & d_{p-1,p-1} \end{bmatrix}_{(p-1)\times(p-1)} \qquad D^p = \left[\begin{array}{c|c} D & 0 \\ \hline 0 & 0 \end{array}\right]_{n\times n}$$

and A^p as defined above, where $h = p - 1$, their product is

$$D^p A^p = \left[\begin{array}{c|c|c} 0 & DA & 0 \\ \hline & 0 & \\ 0 & \vdots & 0 \\ & 0 & \end{array}\right]_{n\times n} \tag{21}$$

and for C^p as defined above, with $w = p - 1$, we have that

$$C^p D^p = \left[\begin{array}{c|c} 0 & 0 \\ \hline CD & 0 \cdots 0 \\ \hline 0 & 0 \end{array}\right]_{n\times n} \tag{22}$$

5.3 Auxiliary Functions

The functions discussed in this section will be used by the main function presented in Section 5.4.2 to define the Kleene star. Sections 5.3.1-5.3.4 introduce four functions. These will be used by the main function to perform the necessary padding of matrices, as discussed in Section 5.1. Section 5.3.5 presents a function used for the base case (i.e. a 2×2-matrix).

Let the square matrix A be

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

5.3.1 Square Matrix to Row

This function returns an $m \times 1$-matrix based on the input matrix A, except padded to a square $n \times n$-matrix ($m \leq n$).

fun *sqmatrix_to_row* **where**

"*sqmatrix_to_row* (*SqMatrix A* :: ('*a* :: *zero*, '*m* :: *len*) *sqmatrix*) (*row* :: *nat*) (*len* :: *nat*) =
SqMatrix(λ *i j*.
if((*Rep_atMost i*) = *row* ∧ (*Rep_atMost j*) <= *len*) *then*
A i j
else
0)"

Applying this function to the square matrix A and two positive integers row and len we get the $n \times n$-matrix

$$\begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} \quad \text{where } b_{ij} = \begin{cases} a_{ij} & \text{if } i = row \ \wedge \ j \leq len \\ 0 & \text{otherwise} \end{cases}$$

I.e. every element in the returned matrix will be 0, except for those in the row specified by the parameters.

5.3.2 Square Matrix to Column

This function returns a $1 \times m$-matrix based on the input matrix, except padded to a square $n \times n$-matrix ($m \leq n$).

fun *sqmatrix_to_column* **where**

"*sqmatrix_to_column* (*SqMatrix A* :: ('*a* :: *zero*,' *m* :: *len*) *sqmatrix*) (*column* :: *nat*) (*len* :: *nat*) =
SqMatrix(λ *i j*.
if((*Rep_atMost j*) = *column* ∧ (*Rep_atMost i*) <= *len*) *then*
A i j
else
0)"

Similar to the previously described function — *sqmatrix_to_row* — applying this function to the square matrix A and two positive integers *column* and *len* we get the $n \times n$-matrix

$$\begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} \quad \text{where } b_{ij} = \begin{cases} a_{ij} & \text{if } j = column \ \wedge \ i \leq len \\ 0 & \text{otherwise} \end{cases}$$

I.e. every element in the returned matrix will be 0, except for those in the column specified by the parameters.

5.3.3 Square Matrix to Square Matrix

This function returns an $m \times m$-matrix based on the input matrix, except padded to a square $n \times n$-matrix ($m \leq n$).

fun *sqmatrix_to_sqmatrix* **where**

"*sqmatrix_to_sqmatrix* (*SqMatrix A* :: ('*a* :: *zero*,' *m* :: *len*) *sqmatrix*) (*at* :: *nat*) =
SqMatrix(λ *i j*.
if((*Rep_atMost i*) <= *at* ∧ (*Rep_atMost j*) <= *at*) *then*
A i j
else
0)"

Applying this function to the square matrix A and the positive integer at we get the $n \times n$-matrix

$$\begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} \quad \text{where } b_{ij} = \begin{cases} a_{ij} & \text{if } i \leq at \ \wedge \ j \leq at \\ 0 & \text{otherwise} \end{cases}$$

I.e. every element in the returned matrix is unchanged, except for those outside the square specified by at.

5.3.4 Square Matrix to Star Element

This function essentially returns a 1×1-matrix based on the input matrix, except padded. The padded matrix is a square $n \times n$-matrix.

fun *sqmatrix_to_star_element* **where**

"*sqmatrix_to_star_element* (*SqMatrix A* :: ('*a* :: *kleene_algebra*,' *m* :: *len*) *sqmatrix*)
(*row* :: *nat*) (*column* :: *nat*) =
SqMatrix(λ *i j*.
if((*Rep_atMost i*) = *row* ∧ (*Rep_atMost j*) = *column*) *then*
(*A i j*)*
else
0)"

Applying this function to the square matrix A and the positive integers row and $column$, we get the $n \times n$-matrix

$$\begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} \quad \text{where } b_{ij} = \begin{cases} a_{ij}^* & \text{if } i = row \ \wedge \ j = column \\ 0 & \text{otherwise} \end{cases}$$

In other words, every element in the returned matrix is 0, except for the one element specified by *row* and *column*. This element is the Kleene star of the corresponding element in A, i.e. $a^*_{row,column}$.

5.3.5 Star of a 2×2-Matrix

The following function is used by the main function for 2×2-matrices.

```
fun star'_of_2_sqmatrix where
  "star'_of_2_sqmatrix (SqMatrix A :: ('a :: kleene_algebra,' m :: len) sqmatrix) =
      (
      let a = (A (Abs_atMost 0) (Abs_atMost 0));
          b = (A (Abs_atMost 0) (Abs_atMost 1));
          c = (A (Abs_atMost 1) (Abs_atMost 0));
          d = (A (Abs_atMost 1) (Abs_atMost 1))
      in
         SqMatrix
            (λ i j.
               if((Rep_atMost i) = 0 ∧ (Rep_atMost j) = 0) then
                  (a + b · d^⋆ · c)^⋆
               else if((Rep_atMost i) = 0 ∧ (Rep_atMost j) = 1) then
                  (a + b · d^⋆ · c)^⋆ · b · d^⋆
               else if((Rep_atMost i) = 1 ∧ (Rep_atMost j) = 0) then
                  d^⋆ · c · (a + b · d^⋆ · c)^⋆
               else if((Rep_atMost i) = 1 ∧ (Rep_atMost j) = 1) then
                  d^⋆ + d^⋆ · c · (a + b · d^⋆ · c)^⋆ · b · d^⋆
               else
                  0
            )
      )"
```

Applying this function to the square matrix A we get the $n \times n$-matrix

$$\begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} \quad \text{where } b_{ij} = \begin{cases} f^* & \text{if } i = 1 \ \wedge \ j = 1 \\ f^*bd^* & \text{if } i = 1 \ \wedge \ j = 2 \\ d^*cf^* & \text{if } i = 2 \ \wedge \ j = 1 \\ d^* + d^*cf^*bd^* & \text{if } i = 2 \ \wedge \ j = 2 \\ 0 & \text{otherwise} \end{cases}$$

for

$$a = a_{11}, \quad b = a_{12}, \quad c = a_{21}, \quad d = a_{22}, \quad f = a + bd^*c.$$

Every element in the returned matrix is 0, except for those in the submatrix consisting of b_{11}, b_{12}, b_{21} and b_{22}. For the submatrix containing these elements, the following holds

$$\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^*$$

5.4 Recursive Star Function

5.4.1 Q_2, Q_3 and Q_4

Below are the three functions *star'_get_Q2*, *star'_get_Q3*, and *star'_get_Q4*, that are used to compute the submatrices of the returned matrix for each iteration (see (16)). The returned matrix consists of the four submatrices, F^*, Q_2, Q_3, and Q_4

$$\left[\begin{array}{c|c} F^* & Q_2 \\ \hline Q_3 & Q_4 \end{array}\right]. \tag{23}$$

fun *star'_get_Q2* **where**
"*star'_get_Q2* ((*SqMatrix E*) :: ('*a* :: *kleene_algebra*,' *m* :: *len*) *sqmatrix*)
(*SqMatrix F_star*) :: ('*a* :: *kleene_algebra*,' *m* :: *len*) *sqmatrix*) (*at* :: *nat*) =
(
let B = *sqmatrix_to_column* (*SqMatrix E*) *at* (*at* − 1);
D_star = *sqmatrix_to_star_element* (*SqMatrix E*) *at at*
in
(*SqMatrix F_star*) ∗ *B* ∗ *D_star*
)"

fun *star'_get_Q3* **where**
"*star'_get_Q3* ((*SqMatrix E*) :: ('*a* :: *kleene_algebra*,' *m* :: *len*) *sqmatrix*)
(*SqMatrix F_star*) :: ('*a* :: *kleene_algebra*,' *m* :: *len*) *sqmatrix*) (*at* :: *nat*) =
(
let C = *sqmatrix_to_row* (*SqMatrix E*) *at* (*at* − 1);
D_star = *sqmatrix_to_star_element* (*SqMatrix E*) *at at*
in
D_star ∗ *C* ∗ (*SqMatrix F_star*)
)"

```
fun star'_get_Q4 where
  "star'_get_Q4 ((SqMatrix E) :: ('a :: kleene_algebra,' m :: len) sqmatrix)
                (SqMatrix F_star) :: ('a :: kleene_algebra,' m :: len) sqmatrix) (at :: nat) =
    (
      let B      = sqmatrix_to_column (SqMatrix E) at (at - 1);
          C      = sqmatrix_to_row (SqMatrix E) at (at - 1);
          D_star = sqmatrix_to_star_element (SqMatrix E) at at
      in
          D_star + D_star * C * (SqMatrix F_star) * B * D_star
    )"
```

Since we are working with square matrices only, the submatrices returned by these functions are padded with zeros where necessary. The padding is done in such a way that adding the four matrices together results in the matrix (23).

5.4.2 The Recursive Function *star'_of_sqmatrix*

If the matrix that the star operation is applied to is of dimensions 1×1 or 2×2 the star function may return the result in a straight-forward manner, simply by returning the 1×1 matrix with the element starred in the former case, or by applying the *star'_of_2_sqmatrix* in the latter case.

To define the recursive function that produces the star of an $n \times n$-matrix E, where $n > 2$, recall the definition of E^* presented in Section 3. We choose to split the matrix into the following submatrices

$$\left[\begin{array}{c|c} A_{(n-1)\times(n-1)} & B_{(n-1)\times 1} \\ \hline C_{1\times(n-1)} & D_{1\times 1} \end{array}\right],$$

This is not an efficient way of doing the split, but since D is a 1×1-matrix, D^* and thus F would be easily defined if we could split the matrix like this (remember that the star operator is already defined in [7] for elements of the matrix). However, as discussed in Section 5.1, this is what we are trying to avoid! Instead of defining D as a 1×1-matrix, we shall see that we may instead use an $n \times n$-matrix.

For a matrix

$$E = \begin{bmatrix} e_{11} & \cdots & e_{1n} \\ \vdots & \ddots & \vdots \\ e_{n1} & \cdots & e_{nn} \end{bmatrix}_{n\times n}$$

let us define

$$A^p = \begin{bmatrix} e_{1,1} & \cdots & e_{1,p-1} \\ \vdots & \ddots & \vdots \\ e_{p-1,1} & \cdots & e_{p-1,p-1} \end{bmatrix}_{(p-1)\times(p-1)} \qquad B^p = \begin{bmatrix} e_{1,p} \\ \vdots \\ e_{p-1,p} \end{bmatrix}_{(p-1)\times 1} \tag{24}$$

$$C^p = \begin{bmatrix} e_{p,1} & \cdots & e_{p,p-1} \end{bmatrix}_{1\times(p-1)} \qquad (D^p)^* = \begin{bmatrix} e^*_{p,p} \end{bmatrix}_{1\times 1}$$

and

$$A_0^p = \left[\begin{array}{c|c} A^p & 0 \\ \hline 0 & 0 \end{array}\right]_{n\times n} \qquad B_0^p = \left[\begin{array}{c|c|c} 0 & B^p & 0 \\ \hline 0 & \begin{array}{c}0\\ \vdots \\ 0\end{array} & 0 \end{array}\right]_{n\times n} \tag{25}$$

$$C_0^p = \left[\begin{array}{c|c} 0 & 0 \\ \hline \begin{array}{c}C^p\end{array} & \begin{array}{ccc}0 & \cdots & 0\end{array} \\ \hline 0 & 0 \end{array}\right]_{n\times n} \qquad (D_0^p)^* = \left[\begin{array}{c|c|c} 0 & \begin{array}{c}0\\ \vdots \\ 0\end{array} & 0 \\ \hline \begin{array}{ccc}0 & \cdots & 0\end{array} & (D^p)^* & \begin{array}{ccc}0 & \cdots & 0\end{array} \\ \hline 0 & \begin{array}{c}0\\ \vdots \\ 0\end{array} & 0 \end{array}\right]_{n\times n}$$

where only column p of B_0^p may be non-zero, and only row p of C_0^p may be non-zero, and only the element in row p and column p of D_0^p may be non-zero. Then

$$(A^p + B^p(D^p)^*C^p) = F^{p-1}_{(p-1)\times(p-1)}.$$

Now, notice that by (18) and (20)

$$A_0^p + B_0^p(D_0^p)^*C_0^p = \left[\begin{array}{c|c} F^{p-1} & 0 \\ \hline 0 & 0 \end{array}\right]_{n\times n} = F_0^{p-1}$$

To get the four matrices A_0^p, B_0^p, C_0^p and $(D_0^p)^*$, we may use the functions defined earlier, namely *sqmatrix_to_sqmatrix* for A_0^p, *sqmatrix_to_column* for B_0^p, *sqmatrix_to_row* for C_0^p, and *sqmatrix_to_star_element* for $(D_0^p)^*$.

Defining F is then easily done by using the existing functions for matrix addition and matrix multiplication defined in [7]: $A_0^p + B_0^p(D_0^p)^*C_0^p$. This will result in F_0^{p-1}, which contains F^{p-1}, but there are also some padding zeros, as we shall see this will resolve itself in the end and the final result will not contain any padding.

Importantly, the dimensions of F^{p-1} is $(p-1)\times(p-1)$. Applying the star function recursively we will therefore have smaller and smaller matrices, until finally the function is called on a 2×2-matrix F^2.

Instead of splitting matrices as in (24), which cannot be done recursively, we may pass the dimensions d of the square matrix being recursed over along with itself as parameters to the recursive function. We then create the matrices in (25) with $p = d$

and may then define F_0^{d-1}. This is the new matrix which is passed recursively along with its dimensions (i.e. $d-1$).

When the dimensions passed is 2×2, i.e. the matrix is F_0^2, the result is already defined by the function $star'\_of\_2\_sqmatrix$ discussed above. This result is then used to calculate $(F_0^3)^*$, which may then be used to find $(F_0^4)^*$ and so on all the way up to $(F_0^d)^* = E^*$, where E is the original matrix. Notice that each $(F_0^i)^*$ has the same dimensions (namely $d \times d$).

To calculate $(F_0^{i+1})^*$ using $(F_0^i)^*$ we only need the already defined matrix multiplication and matrix addition:

$$(F_0^{i+1})^* = (F_0^i)^* + Q_2^i + Q_3^i + Q_4^i \qquad \text{where} \quad i \geq 2$$

and

$$Q_2^i = (F_0^i)^* B_0^{i+1} (D_0^{i+1})^*, \quad Q_3^i = (D_0^{i+1})^* C_0^{i+1} (F_0^i)^*,$$

$$Q_4^i = (D_0^{i+1})^* + (D_0^{i+1})^* C_0^{i+1} (F_0^i)^* B_0^{i+1} (D_0^{i+1})^*$$

and

$$(F_0^2)^* = \left[\begin{array}{c|c} (F^2)^*_{2\times 2} & 0 \\ \hline 0 & 0 \end{array}\right]_{d\times d}$$

These Q-values are defined by the functions described in Section 5.4.1. By (21) and (18), and (19) and (22) respectively we have that

$$Q_2^i = \left[\begin{array}{c|c|c} 0 & (F^i)^* B^{i+1} (D^{i+1})^* & 0 \\ \hline 0 & \begin{array}{c} 0 \\ \vdots \\ 0 \end{array} & 0 \end{array}\right]_{d\times d}$$

$$Q_3^i = \left[\begin{array}{c|c} 0 & 0 \\ \hline (D^{i+1})^* C^{i+1} (F^i)^* & 0 \quad \cdots \quad 0 \\ \hline 0 & 0 \end{array}\right]_{d\times d}$$

where only column i of Q_2^i may contain non-zero elements and only row i of Q_3^i may contain non-zero elements.

For Q_4^i we have that

$$Q_4^i = \left[\begin{array}{ccc|c|ccc} & & & 0 & & & \\ & 0 & & \vdots & & 0 & \\ & & & 0 & & & \\ \hline 0 & \cdots & 0 & (D^{i+1})^* + (D^{i+1})^* C^{i+1} (F^i)^* B^{i+1} (D^{i+1})^* & 0 & \cdots & 0 \\ \hline & & & 0 & & & \\ & 0 & & \vdots & & 0 & \\ & & & 0 & & & \end{array}\right]_{d \times d}$$

where only the element in row and column i may be non-zero. Compare these to the submatrices of E^* in (16) and note that for each $i \leq d$

$$(F_0^i)^* = \left[\begin{array}{c|c} (F^i)^*_{i \times i} & 0 \\ \hline 0 & 0 \end{array}\right]_{d \times d}$$

and

$$(F_0^d)^* = E^*.$$

We have now defined the star operation for square matrices using functions that only operate on square matrices. This function has been implemented in Isabelle by extending the theories in [7] with the functions discussed above, and the following function:

```
fun star'_of_sqmatrix where
  "star'_of_sqmatrix (SqMatrix E :: ('a :: kleene_algebra,' m :: len) sqmatrix) (at :: nat) =
      (
      if(at = 0) then
        SqMatrix(λ i j. if((Rep_atMost i) = 0 ∧ (Rep_atMost j) = 0) then (E i j)^⋆ else 0)
      else if(at = 1) then
        star'_of_2_sqmatrix (SqMatrix E)
      else
        let  new_at = at − 1;
          A = sqmatrix_to_sqmatrix (SqMatrix E) new_at;
          B = sqmatrix_to_column (SqMatrix E) at new_at;
          C = sqmatrix_to_row (SqMatrix E) at new_at;
          D_star = sqmatrix_to_star_element (SqMatrix E) at at;
          F_star = star'_of_sqmatrix (A + B * D_star * C) new_at
        in
          F_star +
          (star'_get_Q2 (SqMatrix E) F_star at) +
          (star'_get_Q3 (SqMatrix E) F_star at) +
          (star'_get_Q4 (SqMatrix E) F_star at)
      )"
```

which is called by the main function

```
fun star_of_sqmatrix where
  "star_of_sqmatrix (A :: ('a :: kleene_algebra, 'm :: len) sqmatrix) =
        (star'_of_sqmatrix A (dim_of_sqmatrix A − 1)"
```

where *dim_of_sqmatrix* is the function

```
fun dim_of_sqmatrix where
  "dim_of_sqmatrix (A :: ('a, 'm :: len0) sqmatrix) = len_of TYPE('m)"
```

which returns the number of rows in a square matrix.

Notice that for matrices of dimension 1, i.e. (a), for any element a, the returned value is (a^*) (this is the first block of the conditional, i.e. when $at = 0$). Dimension 2 is the base case and for larger dimensions the function is called recursively.

The recursive function *star'_of_sqmatrix* is called with matrix dimension argument *dim_of_sqmatrix* $A - 1$ because the matrices count 0 as the first index in the Isabelle implementation.

6 Previous work

This thesis mainly deals with the extension of the work done in [7], which uses Isabelle/HOL to (among other things) specify the algebraic structure known as a Kleene

algebra. Some other results, relevant to this thesis, that can be found in [7] include proofs that some types are instances of Kleene algebras (e.g. regular languages and binary relations).

As discussed in Section 3, the fact that square matrices over a Kleene algebra form a Kleene algebra allows for proving that two elements of a Kleene algebra, regardless of model, are equal. This is the main interest of [6], which uses another proof assistant called *Coq*, to do this. Their approach is similar to the work we build upon here (i.e. [7]) in that they use type classes to define classes corresponding to different algebraic structures, starting with semilattices and monoids, and incrementally building up to Kleene algebras. They then use this to prove that their algorithm for deciding whether two elements are the same is correct. The Coq implementation, however, uses dependent types (i.e. the output type of a function may depend on the input value), which Isabelle does not have.

7 Discussion

This was my first time working with Isabelle — or any proof assistant for that matter — and the experience has been, overall, a pleasant one. There were, however, a few sticking points. One problem I encountered was that the error messages from Isabelle were sometimes rather cryptic, to me. Luckily, often times these could be clarified simply by searching the Internet for some relevant keywords.

Other problems I encountered were the occasionally strange syntax (e.g. *rule_tac*, *exI*, **infixl ...**, etc.), and some unexpected "behind the scenes" work that is performed as a result of some commands (an example would be **typedef**, which quietly declares some constants in the background). The solution for both of these problems was simply to read the available documentation. In particular [14] seems to be very comprehensive, and has been a big help with these types of issues. Apart from this my supervisor, and the good introductory resources available online, e.g. [15] have been a big help when starting out.

There is still a lot to learn. I have, for example, barely started looking at writing proofs, since this has (more or less) not been necessary for this project, which simply uses the specification language. I do however think that I now have a good foundation on which to build a greater understanding of Isabelle.

8 Conclusions

The main topic of this thesis has been the formalization of the Kleene algebra over square matrices in Isabelle/HOL. A formal description of the Kleene star that builds on the previous work done in [7] is suggested. This description has been implemented in Isabelle but not yet been proved correct, since this was outside the scope of the project. The details of this thesis should, however, provide some confidence that the implementation is sound and checking the implementation by hand for small matrix instances does indeed yield correct results. The next step should, however, be to prove that this description satisfies the axioms (12) to (15).

When the formalization has been validated one might attempt to prove the completeness theorem discussed in Section 3.1.2. A formalization in Isabelle/HOL of the completeness theorem is of interest in particular since it should prove a useful result for

application in future work. One example would be extending the decision procedure presented in [16], which describes an equivalence checker for regular expressions formalized using Isabelle/HOL. Using the completeness theorem one should be able to generalize the procedure, such that a pair of elements of any Kleene algebra may be checked for equivalence (as in [6]).

References

[1] Kozen, D.: *A completeness theorem for Kleene algebras and the algebra of regular events*. Infor. and Comput., 110(2):366-390, May 1994.

[2] Kozen, D.: *Kleene Algebra with Tests*. ACM Transactions on Programming Languages and Systems (TOPLAS) 19.3 (1997): 427-443.

[3] Hopkins, M. W., and Kozen, D. C.: *Parikh's Theorem in Commutative Kleene Algebra*. Logic in Computer Science, 1999. Proceedings. 14th Symposium on. IEEE, 1999.

[4] Kozen, D.: *The Design and Analysis of Algorithms*. Springer-Verlag, New York, 1991.

[5] Nipkow, T., Paulson, L. C., and Wenzel, M.: *A Proof Assistant for Higher-Order Logic*. Springer-Verlag Berlin Heidelberg, February 2013.

[6] Braibant, T., and Pous, D.: *An Efficient Coq Tactic for Deciding Kleene Algebras*. Interactive Theorem Proving. Springer Berlin Heidelberg, 2010, 163-178.

[7] Armstrong, A., Struth, G., and Weber, T.: *Kleene Algebra*. http://afp.sourceforge.net/entries/Kleene_Algebra.shtml, October 2013

[8] Gondran, M., and Minoux, M.: *Graphs, Dioids and Semirings: New Models and Algorithms*. Vol. 41. Springer, 2008.

[9] Desharnais, J.: *Kleene algebra with relations*. Relational and Kleene-Algebraic Methods in Computer Science. Springer Berlin Heidelberg, 2004, 8-20.

[10] Sørensen, M. H., and Urzyczyn, P.: *Lectures on the Curry-Howard Isomorphism*. Vol. 149. Elsevier, 2006.

[11] Nipkow, T., Paulson L. C., and Wenzel, M.: *Isabelle's Logics: HOL*. http://isabelle.in.tum.de/website-Isabelle2012/dist/Isabelle2012/doc/logics-HOL.pdf, November 2013.

[12] Wadler, P., and Blott, S.: *How to Make Ad-Hoc Polymorphism Less Ad Hoc*. Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1989.

[13] Haftmann, F.: *Haskell-Style Type Classes with Isabelle/Isar*. http://isabelle.in.tum.de/doc/classes.pdf, November 2013.

[14] Nipkow, T., Paulson L. C., and Wenzel, M.: *A Proof Assistant for Higher-Order Logic*. http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/tutorial.pdf, December 2013.

[15] *Tutorials and manuals for Isabelle2013-2.* http://isabelle.in.tum.de/documentation.html, December 2013.

[16] Krauss, A., and Nipkow, T.: *Proof Pearl: Regular Expression Equivalence and Relation Algebra.* Journal of Automated Reasoning 49.1, 2012, 95-106.