Electronic Notes in Theoretical Computer Science 82 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 16 pages

A Coalgebraic Approach to Kleene Algebra with Tests

Hubie Chen [1]

Department of Computer Science
Cornell University
Ithaca, NY 14853 USA

Riccardo Pucella [2]

Department of Computer Science
Cornell University
Ithaca, NY 14853 USA

Abstract

Kleene Algebra with Tests is an extension of Kleene Algebra, the algebra of regular expressions, which can be used to reason about programs. We develop a coalgebraic theory of Kleene Algebra with Tests, along the lines of the coalgebraic theory of regular expressions based on deterministic automata. Since the known automata-theoretic presentation of Kleene Algebra with Tests does not lend itself to a coalgebraic theory, we define a new interpretation of Kleene Algebra with Tests expressions and a corresponding automata-theoretic presentation. One outcome of the theory is a coinductive proof principle, that can be used to establish equivalence of our Kleene Algebra with Tests expressions.

1 Introduction

Kleene algebra (KA) is the algebra of regular expressions [2,4]. As is well known, the theory of regular expressions enjoys a strong connection with the theory of finite-state automata. This connection was used by Rutten [12] to give a coalgebraic treatment of regular expressions. One of the fruits of this coalgebraic treatment is *coinduction*, a proof technique for demonstrating the equivalence of regular expressions [14]. Other methods for proving the equality of regular expressions have previously been established – for instance, reasoning by using a sound and complete axiomatization [5,15], or by minimization of automata representing the

[1] Email: hubes@cs.cornell.edu
[2] Email: riccardo@cs.cornell.edu

expressions [3]. However, the coinduction proof technique can give relatively short proofs, and is fairly simple to apply.

Recently, Kozen [6] introduced Kleene Algebra with Tests (KAT), an extension of KA designed for the particular purpose of reasoning about programs and their properties. The regular expressions of KAT allow one to intersperse boolean tests along with program actions, permitting the convenient modelling of programming constructs such as conditionals and *while* loops. The utility of KAT is evidenced by the fact that it subsumes propositional Hoare logic, providing a complete deductive system for Hoare-style inference rules for partial correctness assertions [8].

The goal of this paper is to develop a coalgebraic theory of KAT, paralleling the coalgebraic treatment of KA. Our coalgebraic theory yields a coinductive proof principle for demonstrating the equality of KAT expressions, in analogy to the coinductive proof principle for regular expressions. The development of our coalgebraic theory proceeds as follows. We first introduce a form of deterministic automaton and define the language accepted by such an automaton. Next, we develop the theory of such automata, showing that coinduction can be applied to the class of languages representable by our automata. We then give a class of expressions, which play the same role as the regular expressions in classical automata theory, and fairly simple rules for computing derivatives of these expressions.

The difficulty of our endeavor is that the known automata-theoretic presentation of KAT [9] does not lend itself to a coalgebraic theory. Moreover, the notion of derivative, essential to the coinduction proof principle in this context, is not readily definable for KAT expressions as they are defined by Kozen [6]. Roughly, these difficulties arise from tests being commutative and idempotent, and suggest that tests need to be handled in a special way. In order for the coalgebraic theory to interact smoothly with tests, we introduce a *type system* along with new notions of strings, languages, automata, and expressions, which we call *mixed strings*, *mixed languages*, *mixed automata*, and *mixed expressions*, respectively. (We note that none of these new notions coincide with those already developed in the theory of KAT.) All well-formed instances of these notions can be assigned types by our type system. Our type system is inspired by the type system devised by Kozen [7,10] for KA and KAT, but is designed to address different issues.

This paper is structured as follows. In the next section, we introduce mixed strings and mixed languages, which will be used to interpret our mixed expressions. In Section 3, we define a notion of mixed automaton that is used to recognize mixed languages. We then impose a coalgebraic structure on such automata. In Section 4, we introduce our type system for KAT, and connect typed KAT expressions with the mixed language they recognize. In Section 5, we give an example of how to use the coalgebraic theory, via the coinductive proof principle, to establish equivalence of typed KAT expressions. In Section 6, we show that our technique is complete, that is, it can establish the equivalence of any two typed KAT expressions that are in fact equivalent. We conclude in Section 7 with considerations of future work. For reasons of space, we leave the complete proofs for the full paper.

2 Mixed languages

In this section, we define the notions of mixed strings and mixed languages that we will use throughout the paper. Our strings will be defined over two alphabets: a set of primitive programs (denoted $\mathcal{P}$) and a set of primitive tests (denoted $\mathcal{B}$). We allow $\mathcal{P}$ to be infinite, but require that $\mathcal{B}$ be finite.

Primitive tests can be put together to form more complicated tests. A *literal* l is a primitive test $b \in \mathcal{B}$ or its negation $\overline{b}$; the underlying primitive test b is said to be the base of the literal, and is denoted by $base(l)$. When A is a subset of $\mathcal{B}$, $lit(A)$ denotes the set of all literals over A. A *test* is a nonempty set of literals with distinct bases. Intuitively, a test can be understood as the conjunction of the literals it comprises. The *base of a test* t, denoted by $base(t)$, is defined to be the set $\{base(l) : l \in t\}$, in other words, the primitive tests the test t is made up from. We extend the notion of base to primitive programs, by defining the *base of a primitive program* $p \in \mathcal{P}$ as $\varnothing$.

Primitive programs and tests are used to create mixed strings. A *mixed string* is either the empty string, denoted by ϵ, or a sequence $\sigma = a_1 \dots a_n$ (where $n \geq 1$) with the following properties:

(1) each a_i is either a test or primitive program,

(2) for $i = 1, \dots, n-1$, if a_i is a test, then a_{i+1} is a primitive program,

(3) for $i = 1, \dots, n-1$, if a_i is a primitive program, then a_{i+1} is a test, and

(4) for $i = 2, \dots, n-1$, if a_i is a test, then $base(a_i) = \mathcal{B}$.

Hence, a mixed string is an alternating sequence of primitive programs and tests, where each test in the sequence is a "complete" test, except possibly if it occurs as the first or the last element of the sequence. The length of the empty mixed string ϵ is 0, while the length of a mixed string $a_1 \dots a_n$ is n.

We define the concatenation of two mixed strings σ and σ', denoted by $\sigma \cdot \sigma'$, as follows. If one of σ, σ' is the empty string, then their concatenation is the other string. If both $\sigma = a_1 \dots a_n$ and $\sigma' = b_1 \dots b_m$ have non-zero length, their concatenation is defined as:

(1) $\tau = a_1 \dots a_n b_1 \dots b_m$ if exactly one of a_n, b_1 is a primitive program and τ is a mixed string;

(2) $\tau = a_1 \dots a_{n-1}(a_n \cup b_1) b_2 \dots b_m$ if a_n and b_1 are tests such that $base(a_n) \cap base(b_1) = \varnothing$ and τ is a mixed string; and is

(3) undefined otherwise.

Intuitively, concatenation of the two strings is obtained by concatenating the sequence of string elements, possibly by combining the last test of the first string with the first test of the second string, provided that the result is a valid mixed string. We note that concatenation of strings is an associative operation.

We assign one or more types to mixed strings in the following way. A type is of the form $A \rightarrow B$, where A and B are subsets of $\mathcal{B}$. Intuitively, a mixed string has type $A \to B$ if the first element of the string has base A, and it can be concatenated with an element with base B. The mixed string ϵ has many types, namely it has type $A \to A$, for all $A \in \wp(\mathcal{B})$. A mixed string of length 1 consisting of a single test t has type $base(t) \cup A \to A$, for any $A \in \wp(\mathcal{B})$ such that $A \cap base(t) = \varnothing$. A mixed string of length 1 consisting of a single program p has type $\varnothing \to \mathcal{B}$. A mixed string $a_1 \ldots a_n$ of length $n > 1$ has type $base(a_1) \to \mathcal{B} \setminus base(a_n)$. For example, consider the mixed string $\{b\}p\{\bar{b}, c\}q\{b, \bar{c}\}$, with respect to $\mathcal{P} = \{p, q\}$ and $\mathcal{B} = \{b, c\}$. It is easy to establish that it has type $\{b\} \to \varnothing$.

A *mixed language* is a set of mixed strings, and is typeable, with type $A \to B$, if all of the mixed strings it contains have type $A \to B$. In this paper, we will only be concerned with typeable mixed languages.

We will be interested in different operations on mixed languages in the following sections. When L_1,L_2, and L are mixed languages, we use the notation $L_1 \cdot L_2$ to denote the set $\{\sigma_1 \cdot \sigma_2 : \sigma_1 \in L_1, \sigma_2 \in L_2\}$, L^0 to denote the set $\{\epsilon\}$, and for $n \geq 1$, L^n to denote the set $L \cdot L^{n-1}$. The following two operations will be useful in Section 4. The operator T, defined by

$$T(L) = \{\sigma : \sigma \in L, |\sigma| = 1, \sigma \text{ is a test}\},$$

extracts from a language all the mixed strings made up of a single test. The operator ϵ, defined by

$$\epsilon(L) = L \cap \{\epsilon\},$$

essentially checks if the empty mixed string ϵ is in L, since $\epsilon(L)$ is nonempty if and only if the empty mixed string is in L.

3 Mixed automata

A *mixed automaton* over the set of primitive programs $\mathcal{P}$ and set of primitive tests $\mathcal{B}$ is a 3-tuple $M = (\langle S_A \rangle_{A \in \wp(\mathcal{B})}, o, \langle \delta_A \rangle_{A \in \wp(\mathcal{B})})$, consisting of a set S_A of states for each test base A (we call states in $S_\varnothing$ program states), an output function $o : S_\varnothing \to \{0, 1\}$, and transition functions $\delta_\varnothing : S_\varnothing \times \mathcal{P} \to S_\mathcal{B}$ and (for $A \neq \varnothing$) $\delta_A : S_A \times lit(A) \to \bigcup_{A \in \wp(\mathcal{B})} S_A$, subject to the following two conditions:

A1. $\delta_A(s, l) \in S_{A \setminus \{base(l)\}}$, and

A2. for every state s in S_A, for every test t with base A, and for any two orderings $\langle x_1, \ldots, x_m \rangle$, $\langle y_1, \ldots, y_m \rangle$ of the literals in t, if $s \xrightarrow{x_1} \ldots \xrightarrow{x_m} s_1$ and $s \xrightarrow{y_1} \ldots \xrightarrow{y_m} s_2$ then $s_1 = s_2$.

(For convenience, we write $s \xrightarrow{l} s'$ if $\delta_A(s, l) = s'$ for A the base of s.)

As in the coalgebraic treatment of automata [12], and contrary to standard definitions, we allow both the state spaces S_A and the set $\mathcal{P}$ of primitive programs to be infinite. We also do not force mixed automata to have initial states, for reasons that will become clear.

We define a *homomorphism* between mixed automata M and M' to be a sequence $f = \langle f_A \rangle_{A \in \wp(\mathcal{B})}$ of functions $f_A : S_A \to S'_A$ such that:

(1) for all $s \in S_\varnothing$, $o(s) = o'(f_\varnothing(s))$, and for all $p \in \mathcal{P}$, $f_\mathcal{B}(\delta_\varnothing(s,p)) = \delta'_\varnothing(f_\varnothing(s), p)$, and

(2) for all $s \in S_A$ ($A \neq \varnothing$) and all $l \in \mathit{lit}(A)$, $f_{A \setminus \{\mathit{base}(l)\}}(\delta_A(s,l)) = \delta'_A(f_A(s), l)$.

We write $f : M \to M'$ when f is a homomorphism between automata M and M'. For convenience, we often write $f(s)$ for $f_A(s)$ when the type A of s is understood. It is straightforward to verify that mixed automata form a category (denoted $\mathcal{MA}$), where the morphisms of the category are mixed automata homomorphisms.

A *bisimulation* between two mixed automata $M = (\langle S_A \rangle_{A \in \wp(\mathcal{B})}, o, \langle \delta_A \rangle_{A \in \wp(\mathcal{B})})$ and $M' = (\langle S'_A \rangle_{A \in \wp(\mathcal{B})}, o', \langle \delta'_A \rangle_{A \in \wp(\mathcal{B})})$ is a sequence of relations $\langle R_A \rangle_{A \in \wp(\mathcal{B})}$ where $R_A \subseteq S_A \times S'_A$ such that the following two conditions hold:

(1) for all $s \in S_\varnothing$ and $s' \in S'_\varnothing$, if $s R_\varnothing s'$, then $o(s) = o'(s')$ and for all $p \in \mathcal{P}$, $\delta_\varnothing(s,p) R_\mathcal{B} \delta'_\varnothing(s',p)$, and

(2) for all $s \in S_A$ and $s' \in S'_A$ (for $A \neq \varnothing$), if $s R_A s'$, then for all $l \in \mathit{lit}(A)$, $\delta_A(s,l) R_{A \setminus \{\mathit{base}(l)\}} \delta'_A(s', l)$.

A bisimulation between M and itself is called a bisimulation on M. Two states s and s' of the same type B are said to be *bisimilar*, denoted by $s \sim s'$, if there exists a bisimulation $\langle R_A \rangle_{A \in \wp(\mathcal{B})}$ such that $s R_B s'$. The relation $\sim$ is the union of all bisimulations, and in fact is the greatest bisimulation.

In order to establish bisimilarity of states, it will be useful to consider a weaker type of bisimulation. A *pseudo-bisimulation* (relative to the ordering $b_1, \ldots, b_{|\mathcal{B}|}$ of the primitive tests in $\mathcal{B}$) between two mixed automata $M = (\langle S_A \rangle_{A \in \wp(\mathcal{B})}, o, \langle \delta_A \rangle_{A \in \wp(\mathcal{B})})$ and $M' = (\langle S'_A \rangle_{A \in \wp(\mathcal{B})}, o', \langle \delta' \rangle_{A \in \wp(\mathcal{B})})$ is a sequence of relations $\langle R_i \rangle_{i = 0, \ldots, |\mathcal{B}|}$ where $R_i \subseteq S_{A_i} \times S'_{A_i}$ (with A_i denoting $\{b_j : j \leq i, j \in \{1, \ldots, |\mathcal{B}|\}\}$) such that the following two conditions hold:

(1) for all $s \in S_\varnothing$ and $s' \in S'_\varnothing$, if $s R_0 s'$, then $o(s) = o'(s')$ and for all $p \in \mathcal{P}$, $\delta_\varnothing(s,p) R_{|\mathcal{B}|} \delta'_\varnothing(s',p)$, and

(2) for all $i = 1, \ldots, |\mathcal{B}|$, for all $s \in S_{A_i}$ and $s' \in S'_{A_i}$, if $s R_i s'$, then for all $l \in \mathit{lit}(b_i)$, $\delta_{A_i}(s,l) R_{i-1} \delta'_{A_i}(s', l)$.

The sense in which pseudo-bisimulation is weaker than a bisimulation is that there need not be a relation for each element of $\wp(\mathcal{B})$. As the following theorem shows, however, we can always complete a pseudo-bisimulation to a bisimulation.

Theorem 3.1 *If $\langle R_i \rangle_{i=0,\ldots,|\mathcal{B}|}$ is a pseudo-bisimulation (relative to the ordering $b_1, \ldots, b_{|\mathcal{B}|}$ of the primitive tests in $\mathcal{B}$), then there exists a bisimulation $\langle R'_A \rangle$ such that $R'_{A_i} = R_i$ for all $i = 1, \ldots, |\mathcal{B}|$ (with A_i denoting $\{b_j : j \leq i, j \in \{1, \ldots, |\mathcal{B}|\}\}$).*

Let us say that two states s, s' are *pseudo-bisimilar* if they are related by some R_i in a pseudo-bisimulation $\langle R_i \rangle$; it follows directly from Theorem 3.1 that pseudo-bisimilar states are bisimilar.

We now define the mixed language recognized by a state of a mixed automaton. Call a sequence $\mu = e_1 \dots e_m$ of primitive programs and literals a *linearization* of a mixed string $\sigma = a_1 \dots a_n$ if μ can be obtained from σ by substituting each test a_i in σ by a sequence of length $|a_i|$ containing exactly the literals in a_i. For example, with respect to $\mathcal{P} = \{p, q\}$ and $\mathcal{B} = \{b, c\}$, the mixed string $\{b\}p\{\overline{b}, c\}q\{b, \overline{c}\}$ (of type $\{b\} \to \varnothing$) has four linearizations: $bp\overline{b}cqb\overline{c}$, $bpc\overline{b}qb\overline{c}$, $bp\overline{b}cq\overline{c}b$, and $bpc\overline{b}q\overline{c}b$. Intuitively, a mixed string σ is recognized by an automaton if a linearization of σ is accepted by the automaton according to the usual definition. Formally, a mixed string σ is *accepted* by a state s of an automaton M if either

(1) σ is ϵ and s is a program state with $o(s) = 1$ (i.e., s is an accepting program state), or

(2) there exists a linearization $e_1 \dots e_m$ of σ such that $s \xrightarrow{e_1} \dots \xrightarrow{e_m} s'$, s' is a program state, and $o(s') = 1$.

If σ is accepted (by a state s) in virtue of satisfying the second criterion, then every linearization is a witness to this fact – in other words, the existential quantification in the second criterion could be substituted by a universal quantification (over all linearizations of σ) without any change in the actual definition. This is because of condition **A2** in the definition of a mixed automaton, which can be thought of as a sort of "path independence".

We define the mixed language accepted by state s of automaton M, written $L_M(s)$, as the set of mixed strings accepted by state s of M. It is easy to verify that all the strings accepted by a state have the same type, namely, if s is in S_A, then every string in $L_M(s)$ has type $A \to \varnothing$, and hence $L_M(s)$ has type $A \to \varnothing$.

We can verify the following relationships between accepted languages, homomorphisms, and bisimulations. They are similar to those given by Rutten [12].

Proposition 3.2 *If s is a state of M and s' is a state of M' with $s \sim s'$, then $L_M(s) = L_{M'}(s')$.*

Proposition 3.3 *If $f : M \to M'$ is a mixed automaton homomorphism, then $L_M(s) = L_{M'}(f(s'))$.*

It turns out that we can impose a mixed automaton structure on the set of *all* mixed languages with type $A \to \varnothing$. We take as states mixed languages of type $A \to \varnothing$. A state is accepting if the empty string ϵ is in the language. It remains to define the transitions between states; we adapt the idea of Brzozowski derivatives [1]. Our definition of derivative depends on whether we are taking the derivative with respect to a program element or a literal.

If the mixed language L has type $\varnothing \to B$ and $p \in \mathcal{P}$ is a primitive program, define

$$D_p(L) = \{\sigma \ : \ p \cdot \sigma \in L\}.$$

If the mixed language L has type $A \to B$ (for $A \neq \varnothing$) and $l \in lit(A)$ is a literal, then

$$D_l(L) = \{\sigma \ : \ \{l\} \cdot \sigma \in L\}.$$

Define $\mathcal{L}_A$ to be the set of mixed languages of type $A \to \varnothing$. Define $\mathcal{L}$ to be $(\langle \mathcal{L}_A \rangle_{A \in \wp(\mathcal{B})}, o_{\mathcal{L}}, \langle \delta_A \rangle_{A \in \wp(\mathcal{B})})$, where $o_{\mathcal{L}}(L) = 1$ if $\epsilon \in L$, and 0 otherwise; $\delta_{\varnothing}(L, p) = D_p(L)$; and $\delta_A(L, l) = D_l(L)$, for $A \neq \varnothing$ and $l \in \mathit{lit}(A)$. It is easy to verify that $\mathcal{L}$ is indeed a mixed automaton. The following properties of $\mathcal{L}$ are significant.

Proposition 3.4 *For a mixed automaton M with states $\langle S_A \rangle_{A \in \wp(\mathcal{B})}$, the maps $f_A : S_A \to \mathcal{L}$ mapping a state s in S_A to the language $L_M(s)$ is a mixed automaton homomorphism.*

Proposition 3.5 *For any mixed language L in $\mathcal{L}$, the mixed language accepted by state L in $\mathcal{L}$ is L itself, that is, $L_{\mathcal{L}}(L) = L$.*

These facts combine into the following fundamental property of $\mathcal{L}$, namely, that $\mathcal{L}$ is a final automaton.

Theorem 3.6 *$\mathcal{L}$ is final in the category $\mathcal{MA}$, that is, for every mixed automaton M, there is a unique homomorphism from M to $\mathcal{L}$.*

Proof. Let M be a mixed automaton. By Proposition 3.4, there exists a homomorphism f from M to the final automaton $\mathcal{L}$, mapping a state s to the language $L_M(s)$ recognized by that state. Let f' be another homomorphism from M to $\mathcal{L}$. To establish uniqueness, we need to show that for any state s of M, we have $f(s) = f'(s)$:

$$\begin{aligned} f(s) &= L_M(s) && \text{(by definition of } f) \\ &= L_{\mathcal{L}}(f'(s)) && \text{(by Proposition 3.3)} \\ &= f'(s) && \text{(by Proposition 3.5)} \end{aligned}$$

Hence, f is the required unique homomorphism. □

The finality of $\mathcal{L}$ gives rise to the following coinduction proof principle for language equality, in a way which is by now standard [14].

Corollary 3.7 *For two mixed languages K and L of type $A \to \varnothing$, if $K \sim L$ then $K = L$.*

In other words, to establish the equality of two mixed languages, it is sufficient to exhibit a bisimulation between the two languages when viewed as states of the final automaton $\mathcal{L}$. In the following sections, we will use this principle to analyze equality of languages described by a typed form of KAT expressions.

4 Mixed expressions and derivatives

A *mixed expression* (over the set of primitive programs $\mathcal{P}$ and the set of primitive tests $\mathcal{B}$) is any expression built via the following grammar:

$$e ::= 0 \mid 1 \mid p \mid l \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid e^*$$

(with $p \in \mathcal{P}$ and $l \in \mathit{lit}(\mathcal{B})$). For simplicity, we often write $e_1 e_2$ for $e_1 \cdot e_2$. We also freely use parentheses when appropriate. Intuitively, the constants 0 and 1

stand for failure and success respectively. The expression p represents a primitive action, while l represents a primitive test. The operation $+$ is used for choice, $\cdot$ for sequencing, and * for iteration. These are, of course, simply KAT expressions, as defined by Kozen [6]. (In addition to allowing negated primitive tests, Kozen also allows negated tests.) We call them mixed expressions to emphasize the different interpretation we have in mind.

In a way similar to regular expressions denoting regular languages, we define a mapping M from mixed expressions to mixed languages inductively as follows:

$$\begin{aligned} M(0) &= \varnothing \\ M(1) &= \{\epsilon\} \\ M(p) &= \{p\} \\ M(l) &= \{\{l\}\} \\ M(e_1 + e_2) &= M(e_1) \cup M(e_2) \\ M(e_1 \cdot e_2) &= M(e_1) \cdot M(e_2) \\ M(e^*) &= \bigcup_{n \geq 0} M(e)^n \end{aligned}$$

The mapping M is a rather canonical homomorphism from mixed expressions to mixed languages. (It is worth noting that we have not defined any axioms for deriving the "equivalence" of mixed expressions, and it is quite possible for distinct mixed expressions to give rise to the same mixed language.)

Inspired by a type system devised by Kozen [7,10] for KA and KAT expressions, we impose a type system on mixed expressions. The types have the form $A \rightarrow B$, where $A, B \in \wp(\mathcal{B})$, the same types we assigned to mixed strings in Section 2. We shall soon see that this is no accident. We assign a type to a mixed expression via a *type judgment* written $\vdash e : A \rightarrow B$. The following inference rules are used to derive the type of a mixed expression:

$$\vdash 0 : A \rightarrow B \qquad \vdash 1 : A \rightarrow A \qquad \vdash p : \varnothing \rightarrow B$$

$$\vdash l : A \cup \{base(l)\} \rightarrow A \setminus \{base(l)\}$$

$$\frac{\vdash e_1 : A \rightarrow B \quad \vdash e_2 : A \rightarrow B}{\vdash e_1 + e_2 : A \rightarrow B} \qquad \frac{\vdash e_1 : A \rightarrow B \quad \vdash e_2 : B \rightarrow C}{\vdash e_1 \cdot e_2 : A \rightarrow C}$$

$$\frac{e : A \rightarrow A}{e^* : A \rightarrow A}$$

It is clear from these rules that any subexpression of a mixed expression having a type judgment also has a type judgment.

The typeable mixed expressions (which intuitively are the "well-formed" expressions) induce typeable mixed languages via the mapping M, as formalized by the following proposition.

Proposition 4.1 *If* $\vdash e : A \rightarrow B$*, then* $M(e)$ *is a mixed language of type* $A \rightarrow B$*.*

Our goal is to manipulate mixed languages by manipulating the mixed expressions that represent them via the mapping M. (Of course, not every mixed language is in the image of M.) In particular, we are interested in the operations $T(L)$ and $\epsilon(L)$, as defined in Section 2, as well as the language derivatives D_p and D_l introduced in the last section.

We now define operators on mixed expressions that capture those operators on the languages denoted by those mixed expressions. We define $\hat{T}$ inductively on the structure of mixed expressions, as follows:

$$\begin{aligned}
\hat{T}(0) &= 0\\
\hat{T}(1) &= 1\\
\hat{T}(p) &= 0\\
\hat{T}(l) &= l\\
\hat{T}(e_1 + e_2) &= \hat{T}(e_1) + \hat{T}(e_2)\\
\hat{T}(e_1 \cdot e_2) &= \hat{T}(e_1) \cdot \hat{T}(e_2)\\
\hat{T}(e^*) &= \hat{T}(e)^*
\end{aligned}$$

(where $p \in \mathcal{P}$ and $l \in \mathit{lit}(\mathcal{B})$). The operator $\hat{T}$ "models" the operator $T(L)$, as made precise in the following way.

Proposition 4.2 *If* $\vdash e : A \to B$*, then* $\hat{T}(e)$ *is a typeable mixed expression such that* $T(M(e)) = M(\hat{T}(e))$.

We define $\hat{\epsilon}$ inductively on the structure of mixed expressions, as follows:

$$\begin{aligned}
\hat{\epsilon}(0) &= 0\\
\hat{\epsilon}(1) &= 1\\
\hat{\epsilon}(p) &= 0\\
\hat{\epsilon}(l) &= 0\\
\hat{\epsilon}(e_1 + e_2) &= \begin{cases} 0 \text{ if } \hat{\epsilon}(e_1) = \hat{\epsilon}(e_2) = 0 \\ 1 \text{ otherwise} \end{cases}\\
\hat{\epsilon}(e_1 \cdot e_2) &= \begin{cases} 1 \text{ if } \hat{\epsilon}(e_1) = \hat{\epsilon}(e_2) = 1 \\ 0 \text{ otherwise} \end{cases}\\
\hat{\epsilon}(e^*) &= 1
\end{aligned}$$

(where $p \in \mathcal{P}$ and $l \in \mathit{lit}(\mathcal{B})$). Note that $\hat{\epsilon}(e)$ is always the mixed expression 0 or 1. In analogy to Proposition 4.2, we have the following fact connecting the ϵ and $\hat{\epsilon}$ operators.

Proposition 4.3 *If* $\vdash e : A \to B$*, then* $\hat{\epsilon}(e)$ *is a typeable mixed expression such that* $\epsilon(M(e)) = M(\hat{\epsilon}(e))$.

Finally, we define, by induction on the structure of mixed expressions, the derivative operator $\hat{D}$ for typeable mixed expressions. There are two forms of the derivative, in analogy to the two forms of derivative for mixed languages: the derivative $\hat{D}_l$ with respect to a literal $l \in \mathit{lit}(\mathcal{B})$, and the derivative $\hat{D}_p$ with respect to a primitive program $p \in \mathcal{P}$. The two forms of derivative are defined similarly, except on the product of two expressions. (Strictly speaking, since the definition of the derivative depends on the type of the expressions being differentiated, $\hat{D}$ should take type derivations as arguments rather than simply expressions. To lighten the notation, we write $\hat{D}$ as though taking mixed expressions as arguments, with the understanding that the appropriate types are available.)

The derivative $\hat{D}_p$ with respect to a primitive program $p \in \mathcal{P}$ is defined as follows:

$$\begin{aligned}
\hat{D}_p(0) &= 0\\
\hat{D}_p(1) &= 0\\
\hat{D}_p(q) &= \begin{cases} 1 \text{ if } p = q \\ 0 \text{ otherwise} \end{cases}\\
\hat{D}_p(l) &= 0\\
\hat{D}_p(e_1 + e_2) &= \hat{D}_p(e_1) + \hat{D}_p(e_2)\\
\hat{D}_p(e_1 \cdot e_2) &= \begin{cases} \hat{D}_p(e_1) \cdot e_2 & \text{if } B \neq \varnothing \\ \hat{D}_p(e_1) \cdot e_2 + \hat{\epsilon}(e_1) \cdot \hat{D}_p(e_2) & \text{otherwise} \end{cases}\\
&\quad \text{where} \vdash e_1 : A \to B \text{ and} \vdash e_2 : B \to C\\
\hat{D}_p(e^*) &= \hat{D}_p(e) \cdot e^*
\end{aligned}$$

The derivative $\hat{D}_l$ with respect to a literal $l \in lit(\mathcal{B})$ is defined as follows:

$$\begin{aligned}
\hat{D}_l(0) &= 0\\
\hat{D}_l(1) &= 0\\
\hat{D}_l(p) &= 0\\
\hat{D}_l(l') &= \begin{cases} 1 \text{ if } l = l' \\ 0 \text{ otherwise} \end{cases}\\
\hat{D}_l(e_1 + e_2) &= \hat{D}_l(e_1) + \hat{D}_l(e_2)\\
\hat{D}_l(e_1 \cdot e_2) &= \begin{cases} \hat{D}_l(e_1) \cdot e_2 & \text{if } base(l) \notin B \\ \hat{D}_l(e_1) \cdot e_2 + \hat{T}(e_1) \cdot \hat{D}_l(e_2) & \text{otherwise} \end{cases}\\
&\quad \text{where} \vdash e_1 : A \to B \text{ and} \vdash e_2 : B \to C\\
\hat{D}_l(e^*) &= \hat{D}_l(e) \cdot e^*
\end{aligned}$$

We have the following proposition, similar to the previous two, connecting the derivative $\hat{D}$ to the previously defined derivative D on mixed languages.

Proposition 4.4 *Suppose that* $\vdash e : A \to B$.

If $A = \varnothing$*, then for all* $p \in \mathcal{P}$, $D_p(M(e)) = M(\hat{D}_p(e))$.

If $A \neq \varnothing$*, then for all* $l \in lit(A)$, $D_l(M(e)) = M(\hat{D}_l(e))$.

5 Example

In this section, we use the notions of pseudo-bisimulation and the coinduction proof principle (Corollary 3.7), along with the derivative operator $\hat{D}$, to prove the equivalence of two mixed languages specified as mixed expressions.

Fix $\mathcal{P}$ to be the set of primitive programs $\{p, q\}$, and $\mathcal{B}$ to be the set of primitive tests $\{b, c\}$. Let $[b]$ be a shorthand for $(b + \overline{b})$. Define α to be the mixed expression

$$(bp([b]cq)^*\overline{c})^*\overline{b},$$

and β to be the mixed expression

$$bp([b]cq + b\overline{c}p)^*\overline{c}\overline{b} + \overline{b}.$$

Our goal is to prove that α and β are equivalent, in the sense that they induce the same language via the mapping M. In other words, we want to establish that $M(\alpha) = M(\beta)$. This example demonstrates the equivalence of the program

```
while b do {
  p;
  while c do q
}
```

and the program

```
if b then {
  p;
  while b + c do
    if c then q else p
}
```

This equivalence is a component of the proof of the classical result that every *while* program can be simulated by a *while* program with at most one while loop, as presented by Kozen [6]. We refer the reader there for more details.

There are a few ways to establish this equivalence. One is to rely on a sound and complete axiomatization of the equational theory of KAT, and derive the equivalence of α and β algebraically [11]. Another approach is to first construct for each expression an automaton that accepts the language it denotes, and then minimize both automata [9]. Two expressions are then equal if the two resulting automata are isomorphic.

In this paper, we describe a third approach, using the coinductive proof principle for mixed languages embodied by Corollary 3.7. Since the theory we developed in Section 3 applies only to mixed languages of type $A \to \varnothing$, we verify that indeed we have $\vdash \alpha : \{b\} \to \varnothing$ and $\vdash \beta : \{b\} \to \varnothing$, so that, by Proposition 4.1, $M(\alpha)$ and $M(\beta)$ are languages of type $\{b\} \to \varnothing$.

We prove the equivalence of α and β by showing that the mixed languages $M(\alpha)$ and $M(\beta)$ are pseudo-bisimilar, that is, they are related by some pseudo-bisimulation. More specifically, we exhibit a pseudo-bisimulation, relative to the ordering $b_1 = b$, $b_2 = c$, on the final automaton $\mathcal{L}$, such that $M(\alpha)$ and $M(\beta)$ are pseudo-bisimilar. This is sufficient for proving equivalence, since by Theorem 3.1, the languages $M(\alpha)$ and $M(\beta)$ are then bisimilar, and by Corollary 3.7, $M(\alpha) = M(\beta)$.

Define α' to be the mixed expression

$$([b]cq)^*\overline{c}\alpha,$$

and define β' to be the mixed expression

$$([b]cq + b\overline{c}p)^*\overline{c}\overline{b}.$$

Notice that $\beta = bp\beta' + \overline{b}$.

We note that (using the notation of the definition of pseudo-bisimulation), $A_0 = \varnothing$, $A_1 = \{b\}$, and $A_2 = \{b, c\}$. We claim that the following three relations form a pseudo-bisimulation:

$$\begin{aligned} R_2 = \{&(M(\alpha'), M(\beta')), \\ &(M(0), M(0))\} \end{aligned} \qquad \begin{aligned} R_1 = \{&(M([b]q\alpha'), M([b]q\beta')), \\ &(M(\alpha), M(\beta))\} \end{aligned}$$

$$\begin{aligned} R_0 = \{&(M(p\alpha'), M(p\beta')), \\ &(M(q\alpha'), M(q\beta')), \\ &(M(1), M(1)), \\ &(M(0), M(0))\} \end{aligned}$$

It is straightforward to verify that $\langle R_0, R_1, R_2 \rangle$ is a pseudo-bisimulation on $\mathcal{L}$, using the operators defined in the previous section. For instance, consider $D_b(M(\alpha))$, which is equal to $M(\hat{D}_b(\alpha))$ by Proposition 4.4. We compute $\hat{D}_b(\alpha)$ here.

$$\begin{aligned} \hat{D}_b(\alpha) &= \hat{D}_b((bp([b]cq)^*\overline{c})^*)\overline{b} + \hat{T}((bp([b]cq)^*\overline{c})^*)\hat{D}_b(\overline{b}) \\ &= \hat{D}_b(bp([b]cq)^*\overline{c})(bp([b]cq)^*\overline{c})^*\overline{b} + \hat{T}((bp([b]cq)^*\overline{c})^*)0 \\ &= p([b]cq)^*\overline{c}(bp([b]cq)^*\overline{c})^*\overline{b} \\ &= p\alpha' \end{aligned}$$

Hence, $D_b(M(\alpha)) = M(\hat{D}_b(\alpha)) = M(p\alpha')$. The other cases are similar.

As we shall see shortly, there is a way to mechanically construct such a bisimulation to establish the equivalence of two mixed expressions.

We remark that an alternative approach to establish equivalence of while programs based on coalgebras is described by Rutten [13]. This approach uses the operational semantics of the programs instead of an algebraic framework.

6 Completeness

Thus far, we have established a coinductive proof technique for establishing the equality of mixed languages (Section 3), and illustrated its use by showing the equality of two particular mixed languages specified by mixed expressions (Section 5), making use of the derivative calculus developed in Section 4. A natural question about this proof technique is whether or not it can establish the equivalence of *any* two mixed expressions that are equivalent (in that they specify the same mixed language). In this section, we answer this question in the affirmative by formalizing and proving a completeness theorem for our proof technique. In particular, we show that given two equivalent mixed expressions, a finite bisimulation relating them can be effectively constructed, by performing only simple syntactic manipulations. In fact, we exhibit a deterministic procedure for deciding whether or not two mixed expressions are equivalent.

In order to state our completeness theorem, we need a few definitions. We say that two mixed expressions e_1 and e_2 are *equal up to ACI properties*, written $e_1 \overset{\text{ACI}}{=} e_2$, if e_1 and e_2 are syntactically equal, up to the associativity, commutativity, and idempotence of $+$. That is, e_1 and e_2 are equal up to ACI properties if the following three rewriting rules can be applied to subexpressions of e_1 to obtain e_2:

$$e + (f + g) = (e + f) + g$$

$$e + f = f + e$$

$$e + e = e$$

Given a relation $\hat{R}$ between mixed expressions, we define an induced relation $\hat{R}^{\text{ACI}}$ as follows: $e_1 \hat{R}^{\text{ACI}} e_2$ if and only if there exists e_1', e_2' such that $e_1 \overset{\text{ACI}}{=} e_1'$, $e_2 \overset{\text{ACI}}{=} e_2'$, and $e_1' \hat{R} e_2'$.

We define a *syntactic bisimulation* between two mixed expressions e_1 and e_2 having the same type $B \to \varnothing$ (for some $B \subseteq \mathcal{B}$) to be a sequence $\hat{R} = \langle \hat{R}_A \rangle_{A \in \wp(\mathcal{B})}$ of relations such that

(1) for all mixed expressions e, e', if $e \hat{R}_A e'$, then $\vdash e : A \to \varnothing$ and $\vdash e' : A \to \varnothing$,

(2) $e \hat{R}_B e'$,

(3) for all mixed expressions e, e', if $e \hat{R}_\varnothing e'$, then $\hat{\epsilon}(e) = \hat{\epsilon}(e')$, and for all $p \in \mathcal{P}$, $\hat{D}_p(e) \hat{R}^{\text{ACI}}_{\mathcal{B}} \hat{D}_p(e')$, and

(4) for all mixed expressions e, e', if $e \hat{R}_A e'$ (for $A \neq \varnothing$), then for all $l \in \mathit{lit}(A)$, $\hat{D}_l(e) \hat{R}^{\text{ACI}}_{A \setminus \{\mathit{base}(l)\}} \hat{D}_l(e')$.

A syntactic bisimulation resembles a bisimulation, except that it is defined over mixed expressions, rather than over mixed languages. The next theorem shows that any two equivalent mixed expressions are related by a *finite* syntactic bisimulation, that is, a syntactic bisimulation $\hat{R}$ where the number of pairs in each relation $\hat{R}_A$ is finite.

Theorem 6.1 *For all mixed expressions e_1, e_2, of type $A \to \varnothing$, $M(e_1) = M(e_2)$ if* *and only if there exists a finite syntactic bisimulation between e_1 and e_2.*

Proof. ($\Leftarrow$) It is easy to check that a syntactic bisimulation $\hat{R}$ induces a bisimulation R such that $e_1 \hat{R}_A e_2$ if and only if $M(e_1) R_A M(e_2)$. The result then follows by Corollary 3.7.

($\Rightarrow$) We first show how to construct, for every mixed expression e with $\vdash e : A_e \rightarrow B_e$, a finite-state automaton $M = (\langle S_A \rangle_{A \in \wp(\mathcal{B})}, \langle \delta_A \rangle_{A \in \wp(\mathcal{B})})$ with transition functions $\delta_\varnothing : S_\varnothing \times \mathcal{P} \rightarrow S_\mathcal{B}$ and (for $A \neq \varnothing$) $\delta_A : S_A \times \mathit{lit}(A) \rightarrow \bigcup_{A \in \wp(\mathcal{B})} S_A$, satisfying the conditions (i) $\delta_A(s, l) \in S_{A \setminus \{base(l)\}}$, (ii) the states of S_A are mixed expressions having type $A \rightarrow B_e$, (iii) e is a state of S_{A_e}, (iv) if $\delta_\varnothing(s_1, p) = s_2$, then $\hat{D}_p(s_1) \overset{\text{ACI}}{=} s_2$, and (v) if $\delta_A(s_1, l) = s_2$, then $\hat{D}_l(s_1) \overset{\text{ACI}}{=} s_2$. This automaton can be defined by induction on the structure of e. Roughly speaking, the states of the automaton are the mixed expressions (equal up to ACI properties) obtainable from e by taking one or more derivatives.

Given equivalent mixed expressions e_1 and e_2 of type $A \rightarrow \varnothing$, a finite syntactic bisimulation $\hat{R}$ can be constructed as follows. First, construct the automata M_1 and M_2 corresponding to e_1 and e_2. Then, initialize $\hat{R}$ to contain the pair (e_1, e_2), and iterate the following process: for every (e, e') in $\hat{R}$, add the pairs $(\delta_{1,B}(e, x), \delta_{2,B}(e', x))$ (where e, e' have type $B \rightarrow \varnothing$), for all x. Perform this iteration until no new pairs are added to $\hat{R}$. This must terminate, because there are finitely many pairs of states (e, e') with e in M_1 and e' in M_2. It is straightforward to check that $\hat{R}$ is a syntactic bisimulation, under the assumption that $M(e_1) = M(e_2)$. □

The procedure described in the proof of Theorem 6.1 in fact can be easily turned into a procedure for deciding if two mixed expressions are equivalent. To perform this decision, construct $\hat{R}$, and verify that at all pairs of states (e, e') in $\hat{R}$, $\hat{\epsilon}(e) = \hat{\epsilon}(e')$. If this verification fails, then the two mixed expressions are not equivalent, otherwise; they are equivalent.

The bisimulation in Section 5 is indeed a bisimulation induced by a syntactic bisimulation on the mixed expressions α and β.

7 Conclusions and future work

We believe that proofs of equivalence between mixed expressions such as α and β via bisimulation are in general more easily derived than ones obtained through a sound and complete axiomatization of KAT. Given two equivalent mixed expressions, we can exhibit a bisimulation using the purely mechanical procedure underlying Theorem 6.1: use the derivative operators to construct a finite bisimulation in which the two expressions are paired. In contrast, equational reasoning typically requires creativity.

The "path independence" of a mixed automaton (condition **A2**) gives any mixed automaton a certain form of redundancy. This redundancy persists in the definition of bisimulation, and is the reason why a pseudo-bisimulation, a seemingly weaker notion of bisimulation, gives rise to a bisimulation. An open question is to cleanly eliminate this redundancy; a particular motivation for doing this would be to make proofs of expression equivalence as simple as possible. Along these lines, it would be of interest to develop other weaker notions of bisimulation that give rise to bisimulations; pseudo-bisimulations require a sort of "fixed variable ordering" that does not seem absolutely necessary.

Another issue for future work would be to give a class of expressions wider than our mixed expressions for which there are readily understandable and applicable rules for computing derivatives. In particular, a methodology for computing derivatives of the KAT expressions defined by Kozen [6] would be nice to see. Intuitively, there seems to be a tradeoff between the expressiveness of the regular expression language and the simplicity of computing derivatives (in the context of KAT). Formal tools for understanding this tradeoff could potentially be quite useful.

Acknowledgments

The authors wish to thank Dexter Kozen for helpful comments on a draft of this paper. The second author was supported by NSF under grant CTC-0208535, by ONR under grants N00014-00-1-03-41 and N00014-01-10-511, and by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the ONR under grant N00014-01-1-0795.

References

[1] Brzozowski, J. A., *Derivatives of regular expressions*, Journal of the ACM **11** (1964), pp. 481–494.

[2] Conway, J. H., "Regular Algebra and Finite Machines," Chapman and Hall, London, UK, 1971.

[3] Hopcroft, J. E. and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison Wesley, 1979.

[4] Kleene, S. C., *Representation of events in nerve nets and finite automata*, in: C. E. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton University Press, Princeton, NJ, 1956 pp. 3–41.

[5] Kozen, D., *A completeness theorem for Kleene algebras and the algebra of regular events,* Information and Computation **110** (1994), pp. 366–390.

[6] Kozen, D., *Kleene algebra with tests*, Transactions on Programming Languages and Systems **19** (1997), pp. 427–443.

[7] Kozen, D., *Typed Kleene algebra*, Technical Report 98-1669, Computer Science Department, Cornell University (1998).

[8] Kozen, D., *On Hoare logic and Kleene algebra with tests*, in: *Proceedings of the Conference on Logic in Computer Science (LICS'99)* (1999), pp. 167–172.

[9] Kozen, D., *Automata on guarded strings and applications*, Technical Report 2001-1833, Computer Science Department, Cornell University (2001), to appear in *Matématica Contemporânea*.

[10] Kozen, D., *On Hoare logic, Kleene algebra, and types*, in: *Scope of Logic, Methodology, and Philosophy of Science: Volume 1 of the 11th Int. Congress Logic, Methodology and Philosophy of Science, Cracow, August 1999*, Studies in Epistemology, Logic, Methodology, and Philosophy of Science **315**, Kluwer, 2002 pp. 119–133.

[11] Kozen, D. and F. Smith, *Kleene algebra with tests: Completeness and decidability*, in: *Proceedings of the 10th Workshop on Computer Science Logic (CSL'96)*, Lecture Notes in Computer Science **1258** (1996), pp. 244–259.

[12] Rutten, J. J. M. M., *Automata and coinduction (an exercise in coalgebra)*, in: *Proceedings of CONCUR'98*, Lecture Notes in Computer Science **1466**, 1998, pp. 193–217.

[13] Rutten, J. J. M. M., *A note on coinduction and weak bisimilarity for while programs*, Theoretical Informatics and Applications (RAIRO) **33** (1999), pp. 393–400.

[14] Rutten, J. J. M. M., *Universal coalgebra: a theory of systems*, Theoretical Computer Science **249** (2000), pp. 3–80.

[15] Salomaa, A., *Two complete axiom systems for the algebra of regular events*, Journal of the ACM **13** (1966), pp. 158–169.