Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation

Fritz Henglein   Lasse Nielsen

DIKU, University of Copenhagen

{ henglein, lnielsen }@diku.dk

Abstract

We present a new sound and complete axiomatization of regular expression containment. It consists of the conventional axiomatization of concatenation, alternation, empty set and (the singleton set containing) the empty string as an idempotent semiring, the fixed-point rule $E^* = 1 + E \times E^*$ for Kleene-star, and a general coinduction rule as the only additional rule.

Our axiomatization gives rise to a natural computational interpretation of regular expressions as simple types that represent parse trees, and of containment proofs as *coercions*. This gives the axiomatization a Curry-Howard-style constructive interpretation: Containment proofs do not only certify a language-theoretic containment, but, under our computational interpretation, constructively transform a membership proof of a string in one regular expression into a membership proof of the same string in another regular expression.

We show how to encode regular expression equivalence proofs in Salomaa's, Kozen's and Grabmayer's axiomatizations into our containment system, which equips their axiomatizations with a computational interpretation and implies completeness of our axiomatization. To ensure its soundness, we require that the computational interpretation of the coinduction rule be a hereditarily total function. Hereditary totality can be considered the mother of syntactic side conditions: it "explains" their soundness, yet cannot be used as a conventional side condition in its own right since it turns out to be undecidable.

We discuss application of *regular expressions as types* to bit coding of strings and hint at other applications to the wide-spread use of regular expressions for substring matching, where classical automata-theoretic techniques are *a priori* inapplicable.

Neither regular expressions as types nor subtyping interpreted coercively are novel *per se*. Somewhat surprisingly, this seems to be the first investigation of a general proof-theoretic framework for the latter in the context of the former, however.

Categories and Subject Descriptors F.4.3 [*Formal languages*]: Regular sets; D.1.1 [*Applicative (functional) programming*]

General Terms Languages, Theory

Keywords axiomatization, coercion, coinduction, computational interpretation, containment, equivalence, regular expression, type

POPL'11, January 26–28, 2011, Austin, Texas, USA.

1. Introduction

What is *regular expression matching*? In classical theoretical computer science it is the problem of *deciding* whether a string belongs to the regular *language* denoted by a regular expression; that is, it is *membership testing*. In this sense, $abdabc$ matches `((ab)(c|d)|(abc))*`, but $abdabb$ does not. This interpretation is used for most theoretical computer science results: NFA-generation, DFA-generation by subset construction, DFA-minimization, the Myhill-Nerode Theorem, the Pumping Lemma, closure properties, the Star Height Problem, Brzozowski derivatives, fast regular expression equivalence algorithms and matching algorithms, coalgebraic characterizations, bisimulation, etc. If membership testing is all we are interested in, regular expressions and finite automata denoting the same *language* are completely interchangeable. In that case we may as well implement regular expression matching using a state-minimized DFA and forget about the original regular expression.

In *programming*, however, membership testing is rarely good enough: We do not only want a yes/no answer, we also want to obtain proper *matches* of substrings against the constituents of a regular expression so as to *extract* parts of the input for processing. In a *Perl Compatible Regular Expression (PCRE)*[1] matcher, for example, matching $abdabc$ against E = `((ab)(c|d)|(abc))*` yields a substring match for each of the 4 parenthesized subexpressions: They match abc, ab, c, and ϵ (the empty string), respectively. If we use a POSIX matcher (Institute of Electrical and Electronics Engineers (IEEE) 1992) instead, we get $abc, \epsilon, \epsilon, abc$, however. How is this possible? The reason is that $((ab)(c|d)|(abc))*$ is *ambiguous*: the string abc can match the left or the right alternative of $(ab)(c|d)|(abc)$, and returning substring matches makes this difference observable. In a membership testing setting ambiguity is not observable and thus not much studied.

An oddity and limitation of Perl-style matching is that we only get one match under Kleene star, the last one. This is why we get a match of abc above, but not abd. Intuitively, we would like to get the *list* of matches under the Kleene star, not just a single one. This is possible, with *regular expression types* (Hosoya et al. 2005b): Each group can be named by a variable, and the output may contain multiple bindings to the same variable. For a variable under *two* Kleene stars, however, we cannot discern the bindings between the different level-1 Kleene-star groups. An even more refined notion of matching is thus *regular expression parsing*: Returning a parse tree of the input string under the regular expression read as a *grammar*.

A little noticed fact is that the parse trees for a regular expression are isomorphic to the elements of the regular expression read as a *type*; e.g. the type interpretation $\mathcal{T}[\![E]\!]$ of regular expression $E = ((ab)(c|d)|(abc))*$ is $((a \times b) \times (c + d) +$

[1] See `http://www.pcre.org`.

$a \times (b \times c))\,\mathrm{list}$ with a, b, c, d being singleton types identified with the respective values a, b, c, d they contain. The *values* $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]$ and $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]$ are elements of $((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list}$, representing two different parse trees of the same type. Since their *flattening* (unparsing) yields the same string $abdabc$, this shows that $((ab)(c|d)|(abc))^*$ is grammatically ambiguous.

When we have a parsed representation of a string we sometimes need to transform it into a parsed representation of another regular expression. Consider for example E_d = `(ab(c|d))*`, which is equivalent to E. E_d corresponds to a DFA that can be used to match a string efficiently. But what if we need a parsed representation of the string with respect to E? We need a *coercion*, a function that maps parse trees under one regular expression (here E_d) into parse trees under another regular expression (here E) such that the underlying string is preserved. Since E is ambiguous there are different coercions for doing this. The choice of coercion thus incorporates a particular ambiguity resolution strategy; in particular, we may need to make sure that it always returns the *greedy left-most* parse, as in PCRE matching, or the *longest prefix* parse, as in POSIX matching. Also, we will be interested in favoring *efficient* coercions over less efficient ones amongst extensionally equivalent ones; e.g., for coercing E to E, we prefer the constant-time coercion that copies a reference to its input instead of the linear-time coercion that traverses its input and returns a copy of it. Even if coercions are not used to transform parse trees, they are useful for regular expressions under their language (membership testing) interpretation: The existence of a well-typed coercion from $\mathcal{L}[\![E]\!]$ to $\mathcal{L}[\![F]\!]$ is a proof object that logically certifies that E is contained in F. Once it is constructed, it can be *checked* efficiently for ascertaining that E is contained in F instead of embarking on search of a proof of that each time the containment needs to be checked.[2]

The purpose of this paper is to develop the basic theory of *regular expressions as types* with *coercions* interpreting containment as a conceptual and technical framework for regular expression based programming where the classic language-theoretic view is insufficient.

1.1 Contributions

Before delving into the details we summarize our contributions.

1.1.1 Regular expressions as types

The interpretation of regular expressions as types built from empty, unit, singleton, sum, product, and list types was introduced by Frisch and Cardelli (2004) for the purpose of regular expression matching. We allow ourselves to observe and point out that the elements of regular expressions as types correspond exactly to the parse trees of regular expressions understood as grammars. Frisch and Cardelli refer to types as describing "a concrete structured representation of values", but do not verbalize that those representations are essentially parse trees. Conversely, Brabrand and Thomsen (2010) as well as other works define an inference system for parse trees, but do not make explicit that that is tantamount to a type-theoretic interpretation of regular expressions.

1.1.2 Proofs of containment by coercion

We observe that containment can be *characterized* by finding a *coercion*, a function mapping every parse tree under one regular expression to a parse tree with same underlying string in the other regular expression.

This means that proving a containment amounts to finding a coercion for the corresponding regular types, allowing us to bring functional programming intuitions to bear. For example, $E \times E^* \leq E^* \times E$ for all E can be proved by defining the obvious function `f`

```
fun f : 'a * 'a list -> 'a list * 'a
```

that retains the elements in the input.

The idea of a coercion interpretation of an axiomatically given subtyping relation is not new. Our observation expresses something more elementary and "syntax-free", however: The existence of a coercion between regular types, however specified, implies containment of the corresponding regular expressions. Note the direction of reasoning: from existence of coercion to containment.

1.1.3 Coinductive regular expression containment axiomatization with computational interpretation

We give a general coinductive axiomatization of regular expression containment and show how to interpret containment proofs computationally as string-preserving transformations on parse trees. Each rule in our axiomatization corresponds to a natural functional programming construct. Specifically, the coinduction rule corresponds to the principle of definition by recursion, where the side condition guarantees that the resulting function is total.

We show that the derivations of the axiomatizations by Salomaa (1966), Kozen (1994) and Grabmayer (2005) can be coded as coercion judgements in our inference system. This provides a natural computational interpretation for their axiomatizations.

As far as we know, no previous regular expression *axiomatization* has explicitly been given a sound and complete computational interpretation, where *all* derivations are interpreted computationally. Sulzmann and Lu (2007) come close, however. They provide what can be considered the first coercion synthesis algorithm, implemented in an extension of Haskell. They show how to construct an explicit coercion for each valid regular expression containment by providing a computational interpretation of Antimirov's algorithm (Antimirov 1996; Antimirov and Mosses 1995) for deciding regular expression containment. They show that their treatment is sound (Sulzmann and Lu 2007, Lemma A.3), and state that it is complete. We observe that, being based on the construction of deterministic linear forms, their work can be thought of as implementing a proof search using Antimirov's algorithm in Grabmayer's axiomatization.

1.1.4 Parametric completeness

Let us define $E[X_1, \ldots, X_m] \leq F[X_1, \ldots, X_m]$ if the containment holds for all substitutions of X_i with (closed) regular expressions. Our axiomatization is not only complete, but *parametrically complete* for infinite alphabets:
If $E[X_1, \ldots, X_m] \leq F[X_1, \ldots, X_m]$ for all regular expressions $X_1, \ldots, X_m$ then there exists c such that $\vdash c : E[X_1, \ldots, X_m] \leq F[X_1, \ldots, X_m]$. As a consequence, a schematic axiom such as $E \times E^* \leq E^* \times E$ is *derivable*, not just admissible in our axiomatization: we can prove it once and use the *same* proof for all instances of E.

We observe that Kozen's axiomatization (Kozen 1994) is also parametrically complete, but neither Salomaa's (Salomaa 1966) nor Grabmayer's (Grabmayer 2005) appear to be so: In Salomaa's case we need to make a case distinction as to whether the regular expression E substituted for X has the empty word property; and in Grabmayer's case the proofs use the derivatives of E, which are syntax dependent.

1.1.5 Application: Bit coding

We believe regular expressions as types with coercions have numerous applications in programming, both conceptually—how to *think*

[2] The size of a coercion will necessarily be exponential in the sizes of E and F for complexity-theoretic reasons in the worst case, but it *may* be small in many cases.

about regular expressions—and technically. We sketch one potential application: how *bit coding* can be used to compactly represent parse trees and thus strings. This can be thought of as a regular expression specific string representation that often can be compressed more than the original string. A thorough investigation of this and other applications requires separate treatment, however.

1.2 Prerequisites

We assume basic knowledge of regular expressions as in Hopcroft and Ullman (1979), and denotational semantics as in Winskel (1993).

1.3 Notation and terminology

$\mathcal{A}$ denotes an *alphabet*, a possibly infinite set of *symbols* $\{a_i\}_{i \in I}$. The *strings* over $\mathcal{A}$ is the set of finite sequences $\{s, t, \ldots\}$ with elements from $\mathcal{A}$. The *length* of a string s is denoted by $|s|$. The n-ary concatenation of $s_1, \ldots, s_n$ is denoted by their juxtaposition $s_1 \ldots s_n$; for $n = 0$ it denotes the empty string ϵ.

We use inl and inr as the tags distinguishing the elements of a disjoint sum of two sets such that
$X + Y = \{\mathrm{inl}\, v \mid v \in X\} \cup \{\mathrm{inr}\, w \mid w \in Y\}$. We treat recursive types *iso-recursively*, where $(\mathrm{fold}^{-1}, \mathrm{fold})$ denotes the isomorphism between a recursive type and its unrolling. In particular, we define the list type X^* by $\mu Y.1 + X \times Y$. The empty list $[]$ is an abbreviation for $\mathrm{fold}\,(\mathrm{inl}\,())$; and $\mathrm{cons}\,(x, y)$ stands for $\mathrm{fold}\,(\mathrm{inr}\,(x, y))$. The list notation $[x_1, \ldots, x_n]$ is syntactic sugar for $\mathrm{cons}\,(x_1, \ldots, \mathrm{cons}\,(x_n, []))$.

We say a unary predicate P *universally implies* another unary predicate Q if $\forall x.(P(x) \Rightarrow Q(x))$.

2. Regular expressions as types and coercions

In this section we show that a regular expression E can be interpreted as an ordinary *type* and regular expression containment as the existence of a *coercion* between such types. The elements of the types correspond to *proofs* of membership of strings in the regular language denoted by E, which in turn are the parse trees for E viewed as a right-regular *grammar*. A coercion then is any function that transforms parse trees without changing the underlying string.

DEFINITION 1 (Regular expression). *The*regular expressions $\mathrm{Reg}_{\mathcal{A}}$ *is the set of abstract syntax tree defined by the following regular tree grammar:*

$$E, F, G, H ::= 0 \mid 1 \mid a \mid E + F \mid E \times F \mid E^*$$

where $a \in \mathcal{A}$.

In anticipation of our interpretation of regular expressions as types we write $\times$ instead of the more customary juxtaposition or $\cdot$ for concatenation. Our notational convention is that $^*, \times, +$ bind in decreasing order; e.g. $a + a \times b$ stands for $a + (a \times b)$.

2.1 Regular expressions as languages

The *language interpretation* of $\mathrm{Reg}_{\mathcal{A}}$ maps regular expressions to *regular languages* (Kleene 1956). This is also called the *standard interpretation* of $\mathrm{Reg}_{\mathcal{A}}$ since it is isomorphic to the free Kleene algebra over $\mathcal{A}$ (Kozen 1994).

DEFINITION 2 (Language interpretation). *The* language $\mathcal{L}\llbracket E \rrbracket$ *is the set of strings compositionally defined by:*

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{L}\llbracket 0 \rrbracket &=& \emptyset & \mathcal{L}\llbracket E + F \rrbracket &=& \mathcal{L}\llbracket E \rrbracket \cup \mathcal{L}\llbracket F \rrbracket \\
\mathcal{L}\llbracket 1 \rrbracket &=& \{\epsilon\} & \mathcal{L}\llbracket E \times F \rrbracket &=& \mathcal{L}\llbracket E \rrbracket \cdot \mathcal{L}\llbracket F \rrbracket \\
\mathcal{L}\llbracket a \rrbracket &=& \{a\} & \mathcal{L}\llbracket E^* \rrbracket &=& \bigcup_{i \geq 0} (\mathcal{L}\llbracket E \rrbracket)^i
\end{array}$$

where $S \cdot T = \{s\, t \mid s \in S \wedge t \in T\}$, $E^0 = \{\epsilon\}$, $E^{i+1} = E \cdot E^i$.

We write $\models s \in E$ if $s \in \mathcal{L}\llbracket E \rrbracket$; $\models E \leq F$ if $\mathcal{L}\llbracket E \rrbracket \subseteq \mathcal{L}\llbracket F \rrbracket$; and $\models E = F$ if $\mathcal{L}\llbracket E \rrbracket = \mathcal{L}\llbracket F \rrbracket$.

$$\begin{array}{c@{\qquad\qquad}c}
\dfrac{}{\epsilon \in 1} & \dfrac{}{() : 1} \\[2ex]
\dfrac{}{a \in a} & \dfrac{}{a : a} \\[2ex]
\dfrac{s \in E}{s \in E + F} & \dfrac{v : E}{\mathrm{inl}\, v : E + F} \\[2ex]
\dfrac{s \in F}{s \in E + F} & \dfrac{w : F}{\mathrm{inr}\, w : E + F} \\[2ex]
\dfrac{s \in E \qquad t \in F}{s\, t \in E \times F} & \dfrac{v : E \qquad w : F}{(v, w) : E \times F} \\[2ex]
\dfrac{s \in 1 + E \times E^*}{s \in E^*} & \dfrac{v : 1 + E \times E^*}{\mathrm{fold}\, v : E^*}
\end{array}$$

a) Regular expression matching **b)** Type inhabitation

Figure 1. Matching relation and type inhabitation

As expected, $\mathcal{L}\llbracket E^* \rrbracket$ is the set of all finite concatenations of strings from $\mathcal{L}\llbracket E \rrbracket$:

$$\mathcal{L}\llbracket E^* \rrbracket = \{s_1 \ldots s_n \mid n \geq 0 \wedge s_i \in \mathcal{L}\llbracket E \rrbracket \text{ for all } 1 \leq i \leq n\}.$$

DEFINITION 3 (Constant part). *The* constant part $o(E)$ *of E is defined as $o(E) = 1$ if $\epsilon \in \mathcal{L}\llbracket E \rrbracket$ and $o(E) = 0$ otherwise.*

DEFINITION 4 (Matching). *We say s* matches *E and write $\vdash s \in E$ if the statement $s \in E$ is derivable in the inference system in Figure 1a.*

Matching is sound and complete for membership testing:

PROPOSITION 5. $\models s \in E$ *if and only if* $\vdash s \in E$.

The *derivation* of a matching statement $s \in E$ describes a *parse tree* for s under E understood as a regular grammar. This paper is about studying the parse trees, not just the regular language denoted by E.

2.2 Regular expressions as types

Parse trees are in one-to-one correspondence with regular expressions interpreted as *types*; that is, all we need to do is interpret the regular expression constructors as type constructors and we obtain *exactly* the parse trees.

DEFINITION 6 (Type interpretation). *The* type interpretation $\mathcal{T}\llbracket . \rrbracket$ *compositionally maps a regular expression E to a set of structured values:*

$$\begin{array}{l@{\qquad}rcl}
\mathcal{T}\llbracket 0 \rrbracket = \emptyset & \mathcal{T}\llbracket E + F \rrbracket &=& \mathcal{T}\llbracket E \rrbracket + \mathcal{T}\llbracket F \rrbracket \\
\mathcal{T}\llbracket 1 \rrbracket = \{()\} & \mathcal{T}\llbracket E \times F \rrbracket &=& \mathcal{T}\llbracket E \rrbracket \times \mathcal{T}\llbracket F \rrbracket \\
\mathcal{T}\llbracket a \rrbracket = \{a\} & \mathcal{T}\llbracket E^* \rrbracket &=& \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{T}\llbracket E \rrbracket\}
\end{array}$$

We write $\models v : E$ if $v \in \mathcal{T}\llbracket E \rrbracket$.

Note that this is the ordinary interpretation of the regular expression constructors as type constructors: 0 is the empty type, 1 the unit type, a (as a type) the singleton type $\{a\}$, $+$ the sum type constructor, $\times$ the product type constructor, and $.^*$ the list type constructor.

DEFINITION 7 (Inhabitation). *We say v* inhabits *E and write $\vdash v : E$ if the statement $v : E$ is derivable in the inference system in Figure 1b.*

Inhabitation is sound and complete for type membership:

PROPOSITION 8. $\models v : E$ *if and only if* $\vdash v : E$.

By inspection of Figures 1a and 1b we can see that a value v such that $\vdash v : E$ corresponds to a unique derivation of $s \in E$ for a string s that is uniquely determined by *flattening* v.

DEFINITION 9. *The* flattening *function* $\text{flat}(.)$ *from values to strings is defined as follows:*

$$\begin{array}{rclrcl} \text{flat}(()) & = & \epsilon & \text{flat}(a) & = & a \\ \text{flat}(\text{inl}\, v) & = & \text{flat}(v) & \text{flat}(\text{inr}\, w) & = & \text{flat}(w) \\ \text{flat}((v, w)) & = & \text{flat}(v)\,\text{flat}(w) & \text{flat}(\text{fold}\, v) & = & \text{flat}(v) \end{array}$$

In particular we have:

THEOREM 10. $\mathcal{L}[\![E]\!] = \{\text{flat}(v) \mid v \in \mathcal{T}[\![E]\!]\}$

2.3 Regular expression containment as type coercion

Since each regular expression can be thought of as an ordinary type whose elements are all the parse trees for all its strings under the language interpretation, we can characterize regular language containment as the problem of transforming parse trees under one regular expression into parse trees under the other regular expression.

DEFINITION 11 (Coercion). *A function* $f \in \mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]_\perp$ *is a* partial coercion from E to F *if* $\text{flat}(v) = \text{flat}(f(v))$ *for all* $v \in \mathcal{T}[\![E]\!]$ *whenever* $f(v) \neq \perp$*. It is a* total coercion *(or just* coercion*) if* $f(v) \neq \perp$ *for all* $v \in \mathcal{T}[\![E]\!]$.

We write $\mathcal{T}[\![E \leq F_\perp]\!]$ *for the set of partial coercions from E to F; and* $\mathcal{T}[\![E \leq F]\!]$ *for the set of total coercions from E to F.*

In other words, a coercion from E to F is a function that transforms every parse tree under E to a parse tree under F for the same underlying string. Clearly, if there exists a coercion from E to F then $\models E \leq F$: the coercion takes any membership proof of a string in $\mathcal{L}[\![E]\!]$ to a membership proof for the same string in $\mathcal{L}[\![F]\!]$. Conversely, if $\models E \leq F$, we can define a coercion from E to F by mapping any value $v : E$ to a value $w : F$ where $\text{flat}(v) = \text{flat}(w)$.

THEOREM 12 (Containment by coercion). $\models E \leq F$ *if and only if there exists a coercion from E to F.*

An immediate corollary is that two regular expressions are equivalent if and only if there is a pair of coercions between them:

COROLLARY 13 (Equivalence by coercion pairs). $\models E = F$ *if and only if there exists a pair of coercions* (f, g) *such that* $f \in \mathcal{T}[\![E \leq F]\!]$ *and* $g \in \mathcal{T}[\![F \leq E]\!]$.

It may be tempting to expect such pairs to be isomorphisms; that is, $f \circ g = \text{id}_{\mathcal{T}[\![F]\!]}$ and $g \circ f = \text{id}_{\mathcal{T}[\![E]\!]}$). This is generally not the case, however: We have $\models a = a + a$, but there is no isomorphism between them, since there are two values for $a + a$ but only one value for a.

Theorem 12 provides a simple and amazingly useful method for proving regular expression containments by *functional programming*: Find a function from E to F (as *types*!) and make sure that it terminates, outputs each part of the input exactly once and in the same left-to-right order. The latter is usually easily checked when using pattern matching in the definition of the function.

EXAMPLE 14. *We prove the* denesting *rule (Kozen 1997)* $\models (a + b)^* = a^* \times (b \times a^*)^*$*. In one direction, find a function* $f : (a + b)^* \to a^* \times (b \times a^*)^*$ *and make sure that it terminates, uses each part of the input exactly once and outputs them in the same left-to-right order as in the input.*

$$\begin{array}{rcl} f([]) & = & ([], []) \\ f(\text{inl}\, u :: \vec{z}) & = & \textbf{let}\, (\vec{x}, \vec{y}) = f(\vec{z})\, \textbf{in}\, (u :: \vec{x}, \vec{y}) \\ f(\text{inr}\, v :: \vec{z}) & = & \textbf{let}\, (\vec{x}, \vec{y}) = f(\vec{z})\, \textbf{in}\, ([], (v, \vec{x}) :: \vec{y}) \end{array}$$

We can see that f *terminates since it is called recursively with smaller sized arguments, and the output contains the input components in the same left-to-right order. Consequently,* f *defines a coercion, and by Theorem 12 this constitutes a proof that the regular language* $\mathcal{L}[\![(a + b)^*]\!]$ *is contained in* $\mathcal{L}[\![a^* \times (b \times a^*)^*]\!]$*. The other direction is similar.*

In this example we defined an element of the function space $\mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]_\perp$ and then verified manually that it belongs to the subspace $\mathcal{T}[\![E \leq F]\!]$. The following section is about designing a *language* of functions each of which is guaranteed to be a coercion (soundness) and that furthermore is expressive enough so that it contains a term denoting a coercion from E to F whenever $\models E \leq F$ (completeness).

3. Declarative coinductive axiomatization

At the core of all axiomatizations of regular expression equivalence are the axiomatization of product ($\times$), sum ($+$), empty (0) and unit (1) as the free idempotent semiring over $\mathcal{A}$. See Figures 2 and 3. We add the familiar *fold/unfold* axiom for Kleene-star in Figure 4, which models that E^* is a fixed point of $X = 1 + E \times X$. Let us call the resulting inference system *weak equivalence*. It is a sound, but incomplete axiomatization of regular expression equivalence. In particular, we have $\models (a + 1)^* = a^*$, but they are not weakly equivalent (Salomaa 1966, Remark 4).

Intuitively, this is because weak equivalence does not allow invoking *recursively* what we want to prove. The basic idea in our axiomatization is to add recursion by way of a general *finitary coinduction* rule

$$\frac{\begin{array}{c}[E = F] \\ \vdots \\ E = F\end{array}}{E = F} \quad (*)$$

Here $[E = F]$ is a *hypothetical assumption*: It may be used an arbitrary number of times in deriving the premise, but is *discharged* when applying the inference step.

Since the premise is the same as the conclusion, without a side condition $(*)$ restricting its applicability this rule is blatantly unsound: We could simply satisfy the premise by immediately concluding $E = F$ from the hypothetical assumption. By the coinduction rule $E = F$ for arbitrary E, F would be derivable.

The key idea of this paper is to make the side condition not a property of the premise, but of the *derivation* of the premise. To this end we switch from axiomatizing equivalence to axiomatizing containment and equip our inference system with names for the rules, arriving at a type-theoretic formulation with explicit *proof terms*. These proof terms can be computationally interpreted as *coercions* as defined in Section 2.

The coinduction rule then suggestively reads

$$\frac{\begin{array}{c}[f : E \leq F] \\ \vdots \\ c : E \leq F\end{array}}{\text{fix} f.\, c : E \leq F} \quad (*)$$

as in Brandt and Henglein (1998, Section 4.4), where the side condition is a syntactic condition specific to recursive subtyping. The *computational interpretation* of $\text{fix} f.\, c$ is the *recursively defined* partial coercion f such that $f = c$ where c may contain free occurrences of f. For soundness all we need is for the coercion to be total, that is terminate on all inputs. This leads us to the side condition in its most general form:

The computational interpretation of $\text{fix} f.\, c$ must be *total*.

$$
\begin{array}{rcll}
E+(F+G) & = & (E+F)+G & (1)\\
E+F & = & F+E & (2)\\
E+0 & = & E & (3)\\
E+E & = & E & (4)\\
E\times(F\times G) & = & (E\times F)\times G & (5)\\
1\times E & = & E & (6)\\
E\times 1 & = & E & (7)\\
E\times(F+G) & = & (E\times F)+(E\times G) & (8)\\
(E+F)\times G & = & (E\times G)+(F\times G) & (9)\\
0\times E & = & 0 & (10)\\
E\times 0 & = & 0 & (11)
\end{array}
$$

Figure 2. Axioms for idempotent semirings

$$
E=E \qquad \frac{E=F}{F=E} \qquad \frac{E=F \qquad F=G}{E=G}
$$

$$
\frac{E=G \qquad F=H}{E+F=G+H} \qquad \frac{E=G \qquad F=H}{E\times F=G\times H}
$$

Figure 3. Rules of equality

$$
E^* \quad = \quad 1+E\times E^* \qquad (12)
$$

Figure 4. Fold/unfold rule for Kleene-star

$$
\begin{array}{llllrcl}
\Gamma & \vdash & \mathrm{shuffle} & : & E+(F+G) & = & (E+F)+G\\
\Gamma & \vdash & \mathrm{retag} & : & E+F & = & F+E\\
\Gamma & \vdash & \mathrm{untagL} & : & 0+F & = & F\\
\Gamma & \vdash & \mathrm{untag} & : & E+E & \leq & E\\
\Gamma & \vdash & \mathrm{tagL} & : & E & \leq & E+F\\
\Gamma & \vdash & \mathrm{assoc} & : & E\times(F\times G) & = & (E\times F)\times G\\
\Gamma & \vdash & \mathrm{swap} & : & E\times 1 & = & 1\times E\\
\Gamma & \vdash & \mathrm{proj} & : & 1\times E & = & E\\
\Gamma & \vdash & \mathrm{abortR} & : & E\times 0 & = & 0\\
\Gamma & \vdash & \mathrm{abortL} & : & 0\times E & = & 0\\
\Gamma & \vdash & \mathrm{distL} & : & E\times(F+G) & = & (E\times F)+(E\times G)\\
\Gamma & \vdash & \mathrm{distR} & : & (E+F)\times G & = & (E\times G)+(F\times G)\\
\Gamma & \vdash & \mathrm{wrap} & : & 1+E\times E^* & = & E^*\\
\Gamma & \vdash & \mathrm{id} & : & E & = & E
\end{array}
$$

$$
\frac{\Gamma\vdash c : E\leq E' \qquad \Gamma\vdash d : E'\leq E''}{\Gamma\vdash c; d : E\leq E''}
$$

$$
\frac{\Gamma\vdash c : E\leq E' \qquad \Gamma\vdash d : F\leq F'}{\Gamma\vdash c+d : E+F\leq E'+F'}
$$

$$
\frac{\Gamma\vdash c : E\leq E' \qquad \Gamma\vdash d : F\leq F'}{\Gamma\vdash c\times d : E\times F\leq E'\times F'}
$$

$$
\Gamma, f : E\leq F, \Gamma'\vdash f : E\leq F
$$

$$
\frac{\Gamma, f : E\leq F\vdash c : E\leq F}{\Gamma\vdash \mathrm{fix} f.c : E\leq F}\ \text{(coinduction rule)}
$$

Figure 5. Declarative coercion inference system for regular expressions as types. With suitable side conditions for the coinduction rule this is sound and complete for regular expression containment. See Sections 3.2 and 3.3 for side conditions.

Unfortunately, totality turns out to be undecidable, and we present efficiently checkable syntactic conditions that entail totality and yet are expressive enough to admit completeness.

3.1 Axiomatization

Consider the coercion inference system in Figure 5. Each axiom of the form $\Gamma \vdash p : E = F$ is a short-hand for two containment axioms: $\Gamma \vdash p : E \leq F$ and $\Gamma \vdash p^{-1} : F \leq E$.

DEFINITION 15 (Coercion judgement). *Let* Γ *be a sequence of* coercion assumptions *of the form* $f : E \leq F$*, with no* f *repeated. A* coercion judgement *is a statement of the form* $\Gamma \vdash c : E \leq F$ *that is derivable in the inference system of Figure 5.*

If Γ *is empty we may omit it and write* $\vdash c : E \leq F$.

By induction on its derivation, a coercion judgement $f_1 : E_1 \leq F_1, \ldots, f_n : E_n \leq F_n \vdash c : E \leq F$ can be interpreted coherently as a continuous function $\mathcal{F}[\![\Gamma \vdash c : E \leq F]\!] : \mathcal{T}[\![E_1 \leq F_{1\perp}]\!] \times \ldots \times \mathcal{T}[\![E_n \leq F_{n\perp}]\!] \to \mathcal{T}[\![E \leq F_\perp]\!]$, which is specified by the equations in Figure 6. For example, the clauses for retag should be understood as

$$
\begin{array}{l}
\mathcal{F}[\![\Gamma \vdash \mathrm{retag} : E+F \leq F+E]\!](f_1,\ldots,f_n) = \\
\qquad \lambda x.\mathbf{case}\ x\ \mathbf{of}\ \mathrm{inl}\, v \Rightarrow \mathrm{inr}\, v \mid \mathrm{inr}\, v \Rightarrow \mathrm{inl}\, v.
\end{array}
$$

The interpretation of $\mathrm{fix} f.c$ is defined to be the least fixed point of a continuous function on $\mathcal{T}[\![E \leq F_\perp]\!]$, which always exists since $\mathcal{T}[\![E \leq F_\perp]\!]$ is empty or a cpo with bottom. The interpretation of $\mathrm{abortL}, \mathrm{abortR}, \mathrm{abortL}^{-1}, \mathrm{abortR}^{-1}$ is the empty function since the type interpretation of their domain is empty. Formally:

DEFINITION 16 (Computational interpretation).
The computational interpretation

$$
\mathcal{F}[\![f_1 : E_1 \leq F_1, \ldots, f_n : E_n \leq F_n \vdash c : E \leq F]\!]
$$

of a coercion judgement is the (domain-theoretically least) continuous function that maps partial coercions from E_i *to* F_i *bound to the* f_i *to a partial coercion from* E *to* F *satisfying the equations of Figure 6.*

We can interpret all computation judgements, but without a side condition controlling the use of the coinduction rule, the coercion inference system is unsound for deducing regular expression containments. To wit, we can trivially derive $\vdash \mathrm{fix} f.f : E \leq F$ for any E, F. We might hope that a simple *guarding rule* would ensure soundness.

DEFINITION 17 (Left-guarded). *Let* $\Gamma \vdash \mathrm{fix} f.c : E \leq F$ *be a coercion judgement. We say an occurrence of* f *in* c *is* left-guarded *by* d *if* c *contains a subterm of the form* $d \times d'$ *and the particular occurrence of* f *is in* d'*. We call* $\mathrm{fix} f.c$ left-guarded *if for each occurrence of* f *there is a* d *that left-guards* f.

Left-guardedness is not sufficient for soundness, however. Consider

$$
\vdash \mathrm{fix} f.(\mathrm{proj}^{-1} ; (\mathrm{id}_1 \times f) ; \mathrm{proj}) : E \leq F,
$$

which is derivable for all E and F. (For emphasis, we have annotated id with a subscript indicating which regular expression it operates on.) Computationally, this coercion judgement does not terminate on any input. This is an instructive case: It contains *both*

$$\begin{array}{lcl}
\mathrm{shuffle}(\mathrm{inl}\, v) & = & \mathrm{inl}\,(\mathrm{inl}\, v) \\
\mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inl}\, v)) & = & \mathrm{inl}\,(\mathrm{inr}\, v) \\
\mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inr}\, v)) & = & \mathrm{inr}\, v \\
\mathrm{shuffle}^{-1}(\mathrm{inl}\,(\mathrm{inl}\, v)) & = & \mathrm{inl}\, v \\
\mathrm{shuffle}^{-1}(\mathrm{inl}\,(\mathrm{inr}\, v)) & = & \mathrm{inr}\,(\mathrm{inl}\, v) \\
\mathrm{shuffle}^{-1}(\mathrm{inr}\, v) & = & \mathrm{inr}\,(\mathrm{inr}\, v) \\
\mathrm{retag}(\mathrm{inl}\, v) & = & \mathrm{inr}\, v \\
\mathrm{retag}(\mathrm{inr}\, v) & = & \mathrm{inl}\, v \\
\mathrm{retag}^{-1} & = & \mathrm{retag} \\
\mathrm{untagL}\,(\mathrm{inr}\, v) & = & v \\
\mathrm{untag}\,(\mathrm{inl}\, v) & = & v \\
\mathrm{untag}\,(\mathrm{inr}\, v) & = & v \\
\mathrm{tagL}\,(v) & = & \mathrm{inl}\, v \\
\mathrm{assoc}(v, (w, x)) & = & ((v, w), x) \\
\mathrm{assoc}^{-1}((v, w), x) & = & (v, (w, x)) \\
\mathrm{swap}(v, ()) & = & ((), v) \\
\mathrm{swap}^{-1}((), v) & = & (v, ()) \\
\mathrm{proj}((), w) & = & w \\
\mathrm{proj}^{-1}(w) & = & ((), w) \\
\mathrm{distL}(v, \mathrm{inl}\, w) & = & \mathrm{inl}\,(v, w) \\
\mathrm{distL}(v, \mathrm{inr}\, x) & = & \mathrm{inr}\,(v, x) \\
\mathrm{distL}^{-1}(\mathrm{inl}\,(v, w)) & = & (v, \mathrm{inl}\, w) \\
\mathrm{distL}^{-1}(\mathrm{inr}\,(v, x)) & = & (v, \mathrm{inr}\, x) \\
\mathrm{distR}(\mathrm{inl}\, v, w) & = & \mathrm{inl}\,(v, w) \\
\mathrm{distR}(\mathrm{inr}\, v, x) & = & \mathrm{inr}\,(v, x) \\
\mathrm{distR}^{-1}(\mathrm{inl}\,(v, w)) & = & (\mathrm{inl}\, v, w) \\
\mathrm{distR}^{-1}(\mathrm{inr}\,(v, x)) & = & (\mathrm{inr}\, v, x) \\
\mathrm{wrap}\,(v) & = & \mathrm{fold}\, v \\
\mathrm{wrap}^{-1}(v) & = & \mathrm{fold}^{-1}\, v \\
\mathrm{id}(v) & = & v \\
\mathrm{id}^{-1} & = & \mathrm{id} \\
(c; d)(v) & = & d(c(v)) \\
(c + d)(\mathrm{inl}\, v) & = & \mathrm{inl}\,(c(v)) \\
(c + d)(\mathrm{inr}\, w) & = & \mathrm{inr}\,(d(w)) \\
(c \times d)(v, w) & = & (c(v), d(w)) \\
(\mathrm{fix} f.c)(v) & = & c[\mathrm{fix} f.c/f](v)
\end{array}$$

Figure 6. Computational interpretation of coercions

a proj^{-1} coercion *and* an f that is left-guarded "only" by (a coercion operating on) a regular expression,, in this case 1, whose language contains the empty string.

3.2 Soundness

We have seen that, without a side condition on the coinduction rule, the coercion inference system is unsound for deducing regular expression containments. The key idea now is this: Impose a side condition that guarantees that the coercion in the conclusion of the coinduction rule is total. Since all other rules preserve totality of coercions, this yields a sound axiomatization of regular expression containment by Theorem 12. Since our coercions may contain free variables, we need to generalize totality to second-order coercions:

DEFINITION 18 (Hereditary totality). *We say coercion judgement $\Gamma \vdash c : E \leq F$ for $\Gamma = f_1 : E_1 \leq F_1, \ldots, f_n : E_n \leq F_n$ is* hereditarily total *if $\mathcal{F}[\![\Gamma \vdash c : E \leq F]\!](f_1, \ldots, f_n)$ is total whenever f_i is a total coercion from E_i to F_i for all $i = 1, \ldots, n$.*

We are now ready to define sound restrictions of the coercion inference system. Instead of formulating a specific side condition, we parameterize over side conditions for the coinduction rule to express, generally, what is necessary for such a side condition to guarantee soundness.

DEFINITION 19 (Coercion inference system with side condition). *Consider the coercion inference system of Figure 5 where the coinduction rule is equipped with a side condition P, a predicate on the coercion judgement in the conclusion:*

$$\frac{\Gamma, f : E \leq F \vdash c : E \leq F}{\Gamma \vdash \mathrm{fix} f.c : E \leq F} \qquad (P(\Gamma \vdash \mathrm{fix} f.c : E \leq F)).$$

We write $\Gamma \vdash_P c : E \leq F$, if each application of the coinduction rule in the derivation of $\Gamma \vdash c : E \leq F$ satisfies P.

We arrive at the Master Soundness Theorem, which provides a general criterion for sound side conditions:

THEOREM 20 (Soundness). *Let P be any predicate on coercion judgements that universally implies hereditary totality.*
Then $\vdash_P d : E \leq F$ implies $\models E \leq F$ for all d, E, F.

This theorem shows that hereditary totality is an "upper bound" for how liberal the side condition can be without the risk of losing sound computational interpretation of a regular expression containment proof as a coercion. Interestingly, allowing *partial* coercions does not *necessarily* make the resulting inference system unsound for proving regular expression containment. If we define the side condition

$$P^t(\Gamma \vdash \mathrm{fix} f.c : E \leq F) \iff \models E \leq F,$$

the resulting inference system is trivially sound and complete since $\vdash \mathrm{fix} f.f : E \leq F$ is derivable for those E, F such that $\models E \leq F$. Clearly, $\mathcal{F}[\![\vdash \mathrm{fix} f.f : E \leq F]\!]$ is computationally completely useless, however: it never terminates.

Unfortunately, hereditary totality itself is undecidable even for the restricted language of coercions denotable by coercion judgements:[3]

THEOREM 21. *Whether or not $\Gamma \vdash c : E \leq F$ is hereditarily total is undecidable.*

PROOF Even totality of $\vdash c : 1 \leq 1$ is undecidable. This follows from the undecidability of $\vdash c : 1^* \times 1^* \leq 1^* \times 1^*$, which in turn follows from encoding Minsky machines (2-register machines) as closed coercion judgements, using a unary coding of natural numbers. □

This makes hereditary totality inapplicable as a conventional side condition in an axiomatization, where valid instances of an inference rule are expected to be decidable. Below we provide polynomial-time decidable side conditions that are sufficient to encode existing derivations in previous axiomatizations (see Section 3.3). In each case their soundness follows from application of Theorem 20. In this sense, hereditary totality can be considered the "mother of all side conditions", even though it itself is "too extensional" to be used as a conventional side condition.

DEFINITION 22 (Syntactic side conditions S_i). *Define predicates S_1, S_2, S_3 and S_4 on coercion judgements of the form $\Gamma \vdash \mathrm{fix} f.c : E \leq F$ as follows:*

- *$S_1(\Gamma \vdash \mathrm{fix} f.c : E \leq F)$ if and only if each occurrence of f in c is left-guarded by a d where $\Gamma, \ldots \vdash d : E' \leq F'$ is the coercion judgement for d occurring in the derivation of $\Gamma \vdash \mathrm{fix} f.c : E \leq F$ and $o(E') = 0$ (from Definition 3).*
- *$S_2(\Gamma \vdash \mathrm{fix} f.c : E \leq F)$ if and only if each occurrence of f in c is left-guarded and for each subterm of the form $c_1; c_2$ in c at least one of the following conditions is satisfied:*

[3] Proved by Eijiro Sumii, Yasuhiko Minamide, Naoki Kobayashi, Atsushi Igarashi and Fritz Henglein at the IFIP TC 2 Working Group 2.8 meeting at Shirahama, Japan, April 11-16, 2010

  - c_1 *is closed and* proj^{-1}*-free;*
  - c_2 *is closed.*
- $S_3(\Gamma \vdash \mathrm{fix} f.c : E \leq F)$ *if c is of the form* $\mathrm{wrap}^{-1}; (\mathrm{id} + \mathrm{id} \times f); d$ *where d is closed.*
- $S_4 = S_1 \vee S_3$.

It is easy to see that S_1, S_2, S_3 and S_4 are polynomial-time checkable. They furthermore imply hereditary totality:

LEMMA 23 (Hereditary totality for S_i). *Let* $\Gamma \vdash \mathrm{fix} f.c : E \leq F$ *such that* $S_i(\Gamma \vdash \mathrm{fix} f.c : E \leq F)$ *with* $i \in \{1, 2, 3, 4\}$.
Then $\Gamma \vdash \mathrm{fix} f.c : E \leq F$ *is hereditarily total.*

PROOF (Sketch) Side condition S_3 is a special case of S_2. The case of S_4 follows from S_1 and S_3. We have formulated S_3 separately since S_4 is sufficient to code all derivations in Salomaa's and Grabmayer's axiomatizations. S_2 by itself, without S_1, is sufficient for Kozen's axiomatization.

The general idea behind the side conditions S_1, S_2 is that they ensure that every recursive call f in the body c of a recursively defined coercion $\mathrm{fix} f.c$ is called with an argument whose *size* is properly smaller than the size of the original call. The difference between the two conditions is the definition of size in each case.

Consider S_1. Define the *0-size* $|v|_0$ of a value by $|v|_0 = |\mathrm{flat}(v)|$; that is, it is the length of the underlying string. Values containing $()$ may be of 0-size 0, e.g. $|((), ())|_0 = 0$, and the size of a component of a pair may be the same as the size of the pair: $|((), v)|_0 = |v|_0$. Consider a call of $\mathrm{fix} f.c$ to a value v of 0-size n. The predicate S_1 ensures that all recursive calls to f in c are only applied to a value constructed from the second component of some pair, where the first component has size at least 1. Since coercions never increase the size this guarantees that the recursive call is applied to a value of 0-size at most $n - 1$.

Now consider S_2. Define the 1-size $|v|_1$ of v as follows:

$$\begin{array}{rclcrcl} |()|_1 & = & 1 & \qquad & |a|_1 & = & 1 \\ |\mathrm{inl}\, v|_1 & = & |v|_1 & & |\mathrm{inr}\, v|_1 & = & |v|_1 \\ |\mathrm{fold}\, v|_1 & = & |v|_1 & & |(v, w)|_1 & = & |v|_1 + |w|_1 \end{array}$$

Note that if we had defined $|()|_1$ to be 0 then this would be just the 0-size (hence our terminology).

The idea for ensuring termination of a recursively defined coercion is the same as before, but for 1-size instead of 0-size. With 1-size we have the important property that each component of a pair is *properly* smaller than the pair, in particular $|w|_1 < |(v, w)|_1$ for *all* v. We say a coercion c is *nonexpansive* if $|c(v)|_1 \leq |v|_1$. All primitive coercions *except for* proj^{-1} are nonexpansive, and the inference rules preserve nonexpansiveness. Side condition S_2 guarantees that each recursive call is applied to an argument of size properly smaller than the original call. Informally, this is because S_2 guarantees that a recursive call of f is never applied to a value (constructed from) the *output* (return value) of a proj^{-1}-call. □

From Theorem 20, Lemma 23 and Theorem 12 we obtain:

COROLLARY 24 (Soundness for side conditions S_i). *Let* S_1*,* S_2*,* S_3*,* S_4 *as in Definition 22,* $i \in \{1, 2, 3, 4\}$.
Then $\vdash_{S_i} d : E \leq F$ *implies* $\models E \leq F$.

3.3 Completeness

We show now how to code derivations in Salomaa's, Kozen's and Grabmayer's axiomatizations of regular expression equivalence in our coercion inference system (Figure 5) with side condition S_4 (Salomaa, Grabmayer) or S_2 (Kozen). This provides a computational interpretation for each of these systems. Furthermore, it implies that coercion axiomatization with either S_2 or S_4 is complete. More precisely, we encode every derivation of $\vdash E = F$ as a pair of coercion judgements $\vdash c : E \leq F$ and $\vdash d : F \leq E$, which provides a computational interpretation of a regular expression equivalence as a pair of coercions that witness $\vdash E = F$ according to Corollary 13.

$$\frac{E = F}{E^* = F^*}$$

$$E^* \quad = \quad (1 + E)^*$$

$$\frac{E = F \times E + G}{E = F^* \times G} \qquad (\text{if } o(F) = 0)$$

Figure 7. Salomaa's rules for axiomatization F_1

Even though they are for regular expression *equivalence*, these codings also provide completeness of our coercion axiomatization for regular expression *containment*. Assume $\models E \leq F$. This holds if and only if $\models E + F = F$. By completeness of the regular expression equivalence axiomatizations, $E + F = F$ is derivable, and we can construct a coercion judgement of $\vdash c : E + F \leq F$. Composed with $\mathtt{tagL}$ this yields $\vdash \mathtt{tagL}\,; c : E \leq F$, and we are done.

THEOREM 25 (Completeness). *Let P be either* S_2 *or* S_4.
If $\models E \leq F$ *then there exists c such that* $\vdash_P c : E \leq F$.

It follows that any side condition logically "between" S_2 or S_4 on the one hand and hereditary totality on the other hand yields a sound and complete coercion axiomatization of regular expression containment.

COROLLARY 26 (Soundness and completeness). *Let P be such that either* S_2 *or* S_4 *universally implies P, and P universally implies hereditary totality. Then* $\vdash_P c : E \leq F$ *if and only if* $\models E \leq F$.

Whereas hereditary totality is the natural "upper" bound we suspect that there are natural weaker "lower" bounds than S_2 and S_4.

3.3.1 Salomaa

Salomaa's System F_1 (Salomaa 1966) arises from adding the rules of Figure 7 to the axiomatization of weak equivalence (Figures 2, 3 and 4).[4] The side condition of the inference rule in Figure 7 is called the "no empty word property".

To be precise, we prove by induction on derivations of $E = F$ in System F_1 that there exist coercion judgements $\vdash_{S_4} c : E \leq F$ and $\vdash_{S_4} d : F \leq E$. This is straightforward for the weak equivalence rules. We thus concentrate on the rules in Figure 7.

Consider $\dfrac{E = F}{E^* = F^*}$. By induction hypothesis there exist $\vdash_{S_4} c : E \leq F$ and $\vdash_{S_4} d : F \leq E$. We reason as follows: Assume $E^* \leq F^*$ and call this assumption f.

$$\begin{array}{rcll} E^* & \leq & (1 + E \times E^*) & \text{by } \mathrm{wrap}^{-1} \\ & \leq & (1 + F \times F^*) & \text{by } \mathrm{id} + c \times f \\ & \leq & F^* & \text{by } \mathrm{wrap} \end{array}$$

This shows that

$$f : E^* \leq F^* \vdash_{S_4} (\mathrm{wrap}^{-1}; \mathrm{id} + c \times f; \mathrm{wrap}) : E^* \leq F^*.$$

Note that $\vdash \mathrm{fix} f.(\mathrm{wrap}^{-1}; \mathrm{id} + c \times f; \mathrm{wrap}) : E^* \leq F^*$ satisfies side condition S_3 and thus S_4. Its computational interpretation is the map-function on lists. With S_4 satisfied we can ap-

[4] Technically, this is the "left-handed" dual due to Grabmayer (2005) to Salomaa's original "right-handed" formulation, where the fold-unfold rule for Kleene star is axiomatized as $E^* = 1 + E^* \times E$.

ply the coinduction rule to conclude $\vdash_{S_4}$ $\mathrm{fix} f.(\mathrm{wrap}^{-1}; \mathrm{id} + c \times f; \mathrm{wrap}) : E^* \leq F^*$. Similarly, we get $\vdash_{S_4}$ $\mathrm{fix}\, g.(\mathrm{wrap}^{-1}; \mathrm{id} + d \times g; \mathrm{wrap}) : F^* \leq E^*$.

Consider $E^* = (1 + E)^*$. The case $E^* \leq (1 + E)^*$ follows from the rule above since $E \leq 1 + E$. For the converse containment assume $f : (1 + E)^* \leq E^*$. We have:

$$\begin{array}{rcll}
(1+E)^* & \leq & 1 + (1+E) \times (1+E)^* & \text{by wrap}^{-1} \\
& \leq & 1 + (1+E) \times E^* & \text{by } f \\
& \leq & 1 + 1 \times E^* + E \times E^* & \text{by distR} \\
& \leq & 1 + E^* + E \times E^* & \text{by proj} \\
& \leq & 1 + E \times E^* + E^* & \text{by retag} \\
& \leq & E^* + E^* & \text{by wrap} \\
& \leq & E^* & \text{by untag}
\end{array}$$

We are a bit informal here: We have left associativity, congruence and identity steps implicit. Let us consider $\vdash \mathrm{fix} f.c : (1 + E)^* \leq E^*$ now. Without displaying c in full detail, from the derivation above we can see that f satisfies side condition S_3 and thus S_4, and we can conclude $\vdash_{S_4} \mathrm{fix} f.c : (1 + E)^* \leq E^*$ by the coinduction rule. Operationally, $\mathcal{F}[\![\vdash \mathrm{fix} f.c : (1 + E)^* \leq E^*]\!]$ traverses its input list of type $\mathcal{T}[\![(1 + E)^*]\!]$, removes all occurrences of $\mathrm{inl}\,()$ and returns the v's for each $\mathrm{inr}\, v$ in the input.

Finally, consider $\dfrac{E = F \times E + G}{E = F^* \times G}$ (if $o(F) = 0$). Our induction hypothesis is $\vdash_{S_4} c_1 : E \leq F \times E + G$ and $\vdash_{S_4} d_1 : F \times E + G \leq E$. Let us consider $F^* \times G \leq E$ first. Assume $f : F^* \times G \leq E$, and we can calculate $d_2 : F^* \times G \leq E$ as follows:

$$\begin{array}{rcll}
F^* \times G & \leq & (1 + F \times F^*) \times G & \text{by wrap}^{-1} \\
& \leq & 1 \times G + F \times F^* \times G & \text{by distR} \\
& \leq & G + F \times F^* \times G & \text{by proj} \\
& \leq & G + F \times E & \text{by } f \\
& \leq & F \times E + G & \text{by retag} \\
& \leq & E & \text{by } d_1
\end{array}$$

We can see that $\vdash \mathrm{fix} f.d_2 : F^* \times G \leq E$ satisfies S_2 and, since $o(F) = 0$, also S_1 and thus S_4. We can thus conclude $\vdash_{S_4} \mathrm{fix} f.d_2 : F^* \times G \leq E$ by the coinduction rule. Observe that $\vdash \mathrm{fix} f.d_2 : F^* \times G \leq E$ is hereditarily total, whether or not $o(F) = 0$, since S_2 is also satisfied.

For the other direction, assume $\vdash_{S_4} g : E \leq F^* \times G$, and we can calculate $c_2 : E \leq F^* \times G$ essentially in the reverse direction to the above calculation.

$$\begin{array}{rcll}
E & \leq & F \times E + G & \text{by } c_1 \\
& \leq & G + F \times E & \text{by retag} \\
& \leq & G + F \times F^* \times G & \text{by } g \\
& \leq & 1 \times G + F \times F^* \times G & \text{by proj}^{-1} \\
& \leq & (1 + F \times F^*) \times G & \text{by distR}^{-1} \\
& \leq & F^* \times G & \text{by wrap}
\end{array}$$

Here, the coercion judgement $\vdash \mathrm{fix} g.c_2 : E \leq F^* \times G$ may computationally be nonterminating: Choose, e.g., $c_1 = \mathrm{proj}^{-1}; \mathrm{tagL} : E \leq 1 \times E + 0$. For $o(F) = 0$, however, $\vdash \mathrm{fix} g.c_2 : E \leq F^* \times G$ satisfies side condition S_1 and thus S_4; in particular, it always terminates. We can conclude $\vdash_{S_4} \mathrm{fix} g.c_2 : E \leq F \times E + G$ by the coinduction rule.

3.3.2 Kozen

Kozen (1994) has shown that adding the rules in Figure 8 to weak equivalence is sound and complete for regular expression equivalence. Formally, a containment $E \leq F$ in his axiomatization is an abbreviation for $E + F = F$. We show now that all derivations in his system can be coded as coercion judgements with side condition S_2.

$$1 + (E^* \times E) \quad \leq \quad E^*$$

$$\frac{E \times F \leq F}{E^* \times F \leq F} \qquad \frac{E \times F \leq E}{E \times F^* \leq E}$$

Figure 8. Kozen's rules for axiomatization of Kleene Algebras

$$\begin{array}{rcll}
0_{a_i} & = & 0 & \\
1_{a_i} & = & 0 & \\
(a_i)_{a_i} & = & 1 & \\
(a_j)_{a_i} & = & 0 & (i \neq j) \\
(E + F)_{a_i} & = & E_{a_i} + F_{a_i} & \\
(E \times F)_{a_i} & = & E_{a_i} \times F & (o(E) = 0) \\
(E \times F)_{a_i} & = & E_{a_i} \times F + F_{a_i} & (o(E) = 1) \\
(E^*)_{a_i} & = & E_{a_i} \times E^* &
\end{array}$$

Figure 9. Definition of Brzozowski-derivative

Consider $1 + (E^* \times E) \leq E^*$. It is sufficient to construct a coercion judgement $\vdash_{S_2} c : E^* \times E \leq E \times E^*$, since we then have $\vdash_{S_2} \mathrm{id} + c; \mathrm{wrap} : 1 \times E^* \times E \leq E^*$, as desired.

Assume $f : E^* \times E \leq E \times E^*$. We can calculate $c : E^* \times E \leq E \times E^*$ as follows:

$$\begin{array}{rcll}
E^* \times E & \leq & (1 + E \times E^*) \times E & \text{by wrap}^{-1} \\
& \leq & 1 \times E + E \times E^* \times E & \text{by distR} \\
& \leq & 1 \times E + E \times E \times E^* & \text{by } f \\
& \leq & E \times 1 + E \times E \times E^* & \text{by swap} \\
& \leq & E \times (1 + E \times E^*) & \text{by distL}^{-1} \\
& \leq & E \times E^* & \text{by wrap}
\end{array}$$

Writing c explicitly, we have

$$\begin{array}{l}
c = (\mathrm{wrap}^{-1} \times \mathrm{id}); \mathrm{distR}; \\
\quad \mathrm{swap} + (\mathrm{assoc}^{-1}; \mathrm{id} \times f); \\
\quad \mathrm{distL}^{-1}; \mathrm{id} \times \mathrm{wrap}
\end{array}$$

Observe that $\vdash \mathrm{fix} f.c : E^* \times E \leq E \times E^*$ satisfies side condition S_2, and we can conclude $\vdash_{S_2} \mathrm{fix} f.c : E^* \times E \leq E \times E^*$ by the coinduction rule.

Consider the rule $\dfrac{E \times F \leq F}{E^* \times F \leq F}$. Our induction hypothesis is that there exists $\vdash_{S_2} d : E \times F \leq F$. Assume $f : E^* \times F \leq F$, and we calculate $c : E^* \times F \leq F$ as follows:

$$\begin{array}{rcll}
E^* \times F & \leq & (1 + E \times E^*) \times F & \text{by wrap}^{-1} \\
& \leq & 1 \times F + E \times E^* \times F & \text{by distR} \\
& \leq & 1 \times F + E \times F & \text{by } f \\
& \leq & F + E \times F & \text{by proj} \\
& \leq & F + F & \text{by } d \\
& \leq & F & \text{by untag}
\end{array}$$

Note that $\vdash \mathrm{fix} f.c : E^* \times F \leq F$ satisfies side condition S_2, and we can apply the coinduction rule to conclude $\vdash_{S_2} \mathrm{fix} f.c : E^* \times F \leq F$.

The rule $\dfrac{E \times F \leq E}{E \times F^* \leq E}$ is similar to the previous rule, with an additional step involving $E^* \times E \leq E \times E^*$.

3.3.3 Grabmayer

The following results hold for all alphabets, but for convenience we assume that $\mathcal{A}$ is finite in this section.

The *Brzozowski-derivative* E_a (Antimirov 1996; Brzozowski 1964; Conway 1971; Rutten 1998; Salomaa 1966) for regular expression E and $a \in \mathcal{A}$ is defined in Figure 9.

$$\frac{\begin{array}{ccc}[E=F] & & [E=F]\\ \vdots & \cdots & \vdots\\ E_{a_1}=F_{a_1} & & E_{a_n}=F_{a_n}\end{array}}{E=F}\qquad (o(E)=o(F))$$

Figure 10. Grabmayer's coinduction rule COMP/FIX

Grabmayer (2005) recognized that Brzozowski-derivatives can be combined with the ACI-properties of $+$ and the coinductive fixed point rule for recursive types of Brandt and Henglein (1998) to give a *coinductive* axiomatization of regular expression equivalence. His rule COMP/FIX is given in Figure 10. Indeed, it can be seen that in the presence of a transitivity rule of equational logic, the compatibility-with-context-rules, and ACI-axioms, only the rule COMP/FIX is needed to obtain a complete system for regular expression equivalence, without the other rules of Grabmayer's inference system $\mathbf{cREG}_0(\Sigma)$. A sequent style presentation of COMP/FIX is as follows:

$$\frac{\Gamma, E = F \vdash_G E_a = F_a \text{ for all } a \in \mathcal{A}, \quad o(E) = o(F)}{\Gamma \vdash_G E = F}$$

Let us write $\Gamma_{\leq}$ and $\Gamma_{\geq}$ for Γ where all occurrences of $=$ in Γ are replaced by $\leq$, respectively $\geq$. We can show by rule induction that for each derivation of $\Gamma \vdash_G E = F$ there exist coercion judgements $\Gamma_{\leq} \vdash_{S_4} c : E \leq F$ and $\Gamma_{\geq} \vdash_{S_4} d : F \leq E$.

The only interesting rule to consider is COMP/FIX. By induction hypothesis, we have $\Gamma_{\leq}, f : E \leq F \vdash c_a : E_a \leq F_a$ and $\Gamma_{\geq}, g : F \leq E \vdash d_a : F_a \leq E_a$ for all $a \in \mathcal{A}$, where $o(E) = o(F)$. Note that $\models E = o(E) + \sum_{a\in\mathcal{A}} a \times E_a$. Salomaa (1966) shows that $E = o(E) + \sum_{a\in\mathcal{A}} a \times E_a$ is derivable from the rules for weak equivalence (Figures 2, 3 and 4), extended with Salomaa's Axiom A_{11}: $F^* = (1+F)^*$. (See also Grabmayer (2005, Lemma 5, p. 189).) A_{11} is only required for what Frisch and Cardelli (2004) call *problematic* regular expressions, regular expressions of the form G^* where $o(G) = 1$.

By applying the derivation coding of Salomaa's axiomatization from Subsection 3.3.1 to the derivation of $E = o(E) + \sum_{a\in\mathcal{A}} a \times E_a$, we know that there exist $\vdash_{S_4} c_E : o(E) + \sum_{a\in\mathcal{A}} a \times E_a \leq E$ and $\vdash_{S_4} d_E : E \leq o(E) + \sum_{a\in\mathcal{A}} a \times E_a$. This gives us the following derivable coercion judgements:

$$\begin{array}{l}\Gamma_{\leq}, f : E \leq F \vdash_{S_4} d_E; (\mathrm{id}_{o(E)} + \sum_{a\in\mathcal{A}} \mathrm{id}_a \times c_a); c_F : E \leq F\\ \Gamma_{\geq}, g : F \leq E \vdash_{S_4} d_F; (\mathrm{id}_{o(F)} + \sum_{a\in\mathcal{A}} \mathrm{id}_a \times d_a); c_E : F \leq E\end{array}$$

We can observe that they satisfy side condition S_1 and thus S_4. By the coinduction rule we can thus conclude:

$$\begin{array}{l}\Gamma_{\leq} \vdash_{S_4} \mathrm{fix} f.d_E; (\mathrm{id}_{o(E)} + \sum_{a\in\mathcal{A}} \mathrm{id}_a \times c_a); c_F : E \leq F\\ \Gamma_{\geq} \vdash_{S_4} \mathrm{fix} g.d_F; (\mathrm{id}_{o(F)} + \sum_{a\in\mathcal{A}} \mathrm{id}_a \times d_a); c_E : F \leq E\end{array}$$

This provides an alternative proof to the one based on coding Salomaa's System F_1 for concluding that $\models E \leq F$ implies $\vdash_{S_4} E \leq F$.

3.4 Examples

We give examples of coercion judgements for regular expression containments.

EXAMPLE 27 (Denesting as coercion). *We continue Example 14 by implementing the function proving $(a+b)^* \leq a^* \times (b \times a^*)^*$ in the coercion language.*

Abbreviate $E = (a+b)^*$ and $F = a^* \times (b \times a^*)^*$. We can calculate $(a+b)^* \leq a^* \times (b \times a^*)^*$ as follows.

$$\begin{array}{lcll}
E & \leq & 1 + (a+b) \times E & \textit{by}\ \mathrm{wrap}^{-1}\\
 & \leq & 1 + (a+b) \times F & \textit{by}\ f\\
 & \leq & 1 + (b+a) \times F & \textit{by}\ \mathrm{retag}\\
 & \leq & 1 + ((b \times F) + (a \times F)) & \textit{by}\ \mathrm{distR}\\
 & \leq & 1 + (((b \times a^*) \times (b \times a^*)^*) + (a \times F)) & \textit{by}\ \mathrm{assoc}\\
 & \leq & (1 + (b \times a^*) \times (b \times a^*)^*) + ((a \times F)) & \textit{by}\ \mathrm{shuffle}\\
 & \leq & (b \times a^*)^* + (a \times F) & \textit{by}\ \mathrm{wrap}\\
 & \leq & (b \times a^*)^* + ((a \times a^*) \times (b \times a^*)^*) & \textit{by}\ \mathrm{assoc}\\
 & \leq & (b \times a^*)^* + ((1 + a \times a^*) \times (b \times a^*)^*) & \textit{by}\ \mathrm{tagR}\\
 & \leq & (b \times a^*)^* + F & \textit{by}\ \mathrm{wrap}\\
 & \leq & 1 \times (b \times a^*)^* + F & \textit{by}\ \mathrm{proj}^{-1}\\
 & \leq & (1 + a \times a^*) \times (b \times a^*)^* + F & \textit{by}\ \mathrm{tagL}\\
 & \leq & F + F & \textit{by}\ \mathrm{wrap}\\
 & \leq & F & \textit{by}\ \mathrm{untag}
\end{array}$$

Writing it out in full, the coercion coercion judgement is

$$\begin{array}{ll}\vdash \mathrm{fix} f. & \mathrm{wrap}^{-1}; \mathrm{id} + \mathrm{retag} \times f; \mathrm{id} + \mathrm{distR};\\ & \mathrm{id} + (\mathrm{assoc} + \mathrm{assoc}); \mathrm{shuffle}; \mathrm{wrap} + \mathrm{id};\\ & \mathrm{id} + (\mathrm{tagR}\,; \mathrm{wrap}) \times \mathrm{id}; \mathrm{proj}^{-1} + \mathrm{id};\\ & (\mathrm{tagL}\,; \mathrm{wrap}) \times \mathrm{id} + \mathrm{id}; \mathrm{untag}\ :\\ & \qquad (a+b)^* \leq a^* \times (b \times a^*)^*\end{array}$$

In the above example, the coercion is, operationally, basically the function f defined in Example 14: It folds a constant-time computable function over its input list and therefore runs in linear time. Kozen (1994, Proposition 2.7) gives a proof of the same inclusion in his axiomatization of Kleene algebra. When encoding it as in Section 3.3.2 we obtain a similar, linear-time coercion. This raises the question whether computational interpretation of *all* proofs of the *same* containment in the axiomatizations of Salomaa, Kozen and Grabmayer yield coercions of the same, linear-time complexity. Remarkably, this is not the case, as illustrated by the next example.

EXAMPLE 28 (Coercion efficiency). *Consider $a^* \times a^{**} \leq a^*$. The simplest way to prove this with Kozen's rules is to start from $a \times a^* \leq a^*$ proved by* $\mathrm{tagR}\,; \mathrm{wrap}$. *By the left inference rule in Figure 8 we then get $a^* \times a^* \leq a^*$. By the right inference rule in Figure 8 we finally obtain $a^* \times a^{**} \leq a^*$.*

Let us consider the computational interpretation of this proof. We have two (nested) applications of the (left and right, respectively) inference rule from Figure 8. This gives quadratic *runtime. It is unclear to us whether there exists a proof using Kozen's rules whose computational interpretation as a functional program runs in linear time. It is possible to construct a linear-time coercion for $a^* \times a^{**} \leq a^*$ in our coercion inference system, however. This can be systematically obtained by encoding the (minimal) proof in Grabmayer's axiomatization. In fact, the encoding of any proof in Grabmayer's axiomatization will have linear run time. This is because the only admissible application of recursion in Grabmayer's axiomatization is of the form $\mathrm{fix} f.d_E; (\mathrm{id} + \Sigma_{a\in\mathcal{A}} \mathrm{id}_a \times c_a); c_F$, where f does not occur in d_E and c_F, which entails that only constant amount of processing occurs for each constant part of the input.*

3.5 Parametric completeness

Let us extend regular expressions by adding variables that can be bound to arbitrary regular sets. Formally,

$$E, F, G, H ::= 0 \mid 1 \mid a \mid X \mid E + F \mid E \times F \mid E^*$$

where X ranges over a denumerable set of *(formal) variables* $\{X_i\}_{i\in\mathbb{N}}$. Such a regular expression is *closed* if it contains no formal variables. We define $\models \forall X_1, \ldots, X_m.E[X_1, \ldots, X_m] \leq$

$F[X_1, \ldots, X_m]$ if the containment holds for all substitutions of X_i with (closed) regular expressions.

Our axiomatization is immediately applicable to regular expressions with free variables. Without change, it is not only complete, but *parametrically complete* for infinite alphabets:

THEOREM 29 (Parametric completeness). *Let $\mathcal{A}$ be infinite. Let the side condition P for the coinduction rule in Figure 5 be either total hereditariness or S_2. Then: $\models \forall X_1, \ldots, X_m. E \leq F$ if and only if $\vdash_P c : E \leq F$.*

PROOF Only if: By rule induction, coercion axiomatization is closed under substitution, with total hereditariness or S_2 as side condition. Note that this is not the case for S_4. (Technically, S_1 is not even defined for regular expressions with variables, since $o(X)$ is undefined.)

If: Assume $\models \forall X_1, \ldots, X_n. E \leq F$. Let $b_1, \ldots, b_n$ be n distinct alphabet symbols not occurring in E or F. (Since $\mathcal{A}$ is infinite, they exist.) By definition of $\models \forall X_1, \ldots, X_n. E \leq F$, we have $\models E\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\} \leq F\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\}$. By Theorem 25 (completeness), there is a derivable coercion judgement $\vdash_P c : E\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\} \leq F\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\}$. Since coercion axiomatization with hereditary totality or S_2 as side condition is closed under substitution, the $b_1, \ldots, b_n$ can be replaced by arbitrary regular expressions $E_1, \ldots, E_n$, respectively, such that $\vdash c : E\{b_1 \mapsto E_1, \ldots, b_n \mapsto E_n\} \leq F\{b_1 \mapsto E_1, \ldots, b_n \mapsto E_n\}$. In particular, we can choose $X_1, \ldots, X_n$ for $E_1, \ldots, E_n$ and thus obtain $\vdash c : E \leq F$. □

As a consequence of Theorem 29, a schematic containment such as $E \times E^* \leq E^* \times E$ is *derivable*, not just *admissible* in our axiomatization: we can synthesize a *single* coercion judgement for it and use it for all instances of E. For finite alphabets our axiomatization is incomplete, however: $\models \forall X. (X \leq (a+b)^*)$ holds for $\mathcal{A} = \{a, b\}$, but $X \leq (a+b)^*$ is not derivable.

We observe that Kozen's axiomatization (Kozen 1994) is also parametrically complete for infinite alphabets, but not for finite alphabets. Neither Salomaa's (Salomaa 1966) nor Grabmayer's (Grabmayer 2005) appear to be parametrically complete: In Salomaa's case we need to make a case distinction as to whether the regular expression E substituted for X has the empty word property; and in Grabmayer's case E needs to be differentiated, the proof of which depends on the syntax of E.

4. Application: Compact bit representations of parse trees

If the regular expressions are statically known in a program we can code their elements, more precisely their parse trees, compactly as bit strings.

4.1 Bit coded strings

Intuitively, a *bit coding* of a parse tree p factors p into its static part, the regular expression E it is a member of, and its dynamic part, a bit sequence that uniquely identifies p as a particular element of E.

Consider for example the string $s = abaab$ as an element of $H_1 = (a+b)^*$. It has the unique parse tree corresponding to

$$p_s \quad = \quad [\mathrm{inl}\, a, \mathrm{inr}\, b, \mathrm{inl}\, a, \mathrm{inl}\, a, \mathrm{inr}\, b]$$

with $\vdash p_s : H_1$, which shows that $\mathrm{flat}(p_s) = abaab$ is an element of $\mathcal{L}[\![H_1]\!]$.

$$\begin{array}{lcl}
\mathrm{code}(() : 1) & = & \epsilon \\
\mathrm{code}(a : a) & = & \epsilon \\
\mathrm{code}(\mathrm{inl}\, v : E + F) & = & 0\ \mathrm{code}(v : E) \\
\mathrm{code}(\mathrm{inr}\, w : E + F) & = & 1\ \mathrm{code}(w : F) \\
\mathrm{code}((v, w) : E \times F) & = & \mathrm{code}(v : E)\ \mathrm{code}(w : F) \\
\mathrm{code}(\mathrm{fold}\, v : E^*) & = & \mathrm{code}(v : 1 + E \times E^*)
\end{array}$$

Figure 11. Type-directed encoding function from parse trees (values) to bit sequences

$$\begin{array}{lcl}
\mathrm{decode}'(d : 1) & = & ((), d) \\
\mathrm{decode}'(d : a) & = & (a, d) \\
\mathrm{decode}'(0d : E + E') & = & \mathbf{let}\ (v, d') = \mathrm{decode}'(d : E) \\
 & & \mathbf{in}\ (\mathrm{inl}\, v, d') \\
\mathrm{decode}'(1d : E + E') & = & \mathbf{let}\ (w, d') = \mathrm{decode}'(d : E) \\
 & & \mathbf{in}\ (\mathrm{inr}\, w, d') \\
\mathrm{decode}'(d : E \times E') & = & \mathbf{let}\ (v, d') = \mathrm{decode}'(d : E) \\
 & & \quad (w, d'') = \mathrm{decode}'(d' : E') \\
 & & \mathbf{in}\ ((v, w), d'') \\
\mathrm{decode}'(d : E^*) & = & \mathbf{let}\ (v, d') = \mathrm{decode}'(d : 1 + E \times E^*) \\
 & & \mathbf{in}\ (\mathrm{fold}\, v, d') \\
\mathrm{decode}(d : E) = & = & \mathbf{let}\ (w, d') = \mathrm{decode}'(d : E) \\
 & & \mathbf{in}\ \mathbf{if}\ d' = \epsilon\ \mathbf{then}\ w\ \mathbf{else}\ \mathit{error}
\end{array}$$

Figure 12. Type-directed decoding function from bit sequences to parse trees (values)

Figures 11 and 12 define regular-expression directed coding and decoding functions from parse trees to their *(canonical) bit codings* and back. Informally, the bit coding of a parse tree consists of listing the inl- and inr-constructors in preorder traversal, where inl is mapped to 0 and inr is mapped to 1. No bits are generated for the other constructors. For example, the bit coding b_s for p_s is 10 11 10 10 11 0.

We can think of the bit coding of a parse tree p as a bit coding of the underlying string $\mathrm{flat}(p)$. Note that the bit coding of a string is not unique. It depends on

- which regular expression it is parsed under; and
- if the regular expression is ambiguous, which parse tree is chosen for it.

As an illustration of the first effect, the bit representation b'_s of s under $H_2 = 1 + (a+b)^* \times (a+b)$ is 1 10 11 10 10 0 1. Since both H_1 and H_2 are unambiguous there are no other bit representations of s under either H_1 or H_2.

4.2 Bit code coercions

So what if we have a bit representation of a run-time string under one regular expression and we need to transform it into a bit representation under another regular expression? This arises if the branches of a conditional expression each return a bit-coded string, but under different regular expressions E_1, E_2, and we need to ensure that the result of the conditional is a bit coding in a common regular expression F that contains the E_1 and E_2.

Let us consider s again and how to transform b_s into b'_s. As we have seen in Section 3.3, E^* is contained in $1 + (E^* \times E)$ for all E and there is a parametric polymorphic coercion $c_1 : \forall X. X^* \leq 1 + (X^* \times X)$ mapping any value $\vdash p : E^*$ representing a parse tree for string $s' = \mathrm{flat}(p)$ to a parse tree $\vdash p' : 1 + E^* \times E$ for s'. In particular applying c_1 to p_s yields p'_s.

We can compose c_1 with code and decode from Figures 11 and 12 to compute a function $\hat{c_1}$ operating on bit codings:

$$\hat{c_1} = \mathrm{code} \cdot c_1 \cdot \mathrm{decode}.$$

Instead of converting to and from values we can define a *bit coding coercion* by providing a computational interpretation of

$$
\begin{array}{lcl}
\mathrm{retag}(0d) & = & 1d \\
\mathrm{retag}(1d) & = & 0d \\
\mathrm{retag}^{-1} & = & \mathrm{retag} \\
\mathrm{tagL}\,(d) & = & 0d \\
\mathrm{untag}\,(bd) & = & d \\
\mathrm{shuffle}(0d) & = & 00d \\
\mathrm{shuffle}(10d) & = & 01d \\
\mathrm{shuffle}(11d) & = & 1d \\
\mathrm{shuffle}^{-1}(00d) & = & 0d \\
\mathrm{shuffle}^{-1}(01d) & = & 10d \\
\mathrm{shuffle}^{-1}(1d) & = & 11d \\
\mathrm{swap}(d) & = & d \\
\mathrm{swap}^{-1} & = & \mathrm{swap} \\
\mathrm{proj}(d) & = & d \\
\mathrm{proj}^{-1}(d) & = & d \\
\mathrm{assoc}(d) & = & d \\
\mathrm{assoc}^{-1}(d) & = & d \\
\mathrm{distL}(d : E \times (F + G)) & & \\
 & = & \mathbf{let}\,(d_1, bd_2) = \mathrm{split}(d : E) \\
 & & \mathbf{in}\, bd_1 d_2 \\
\mathrm{distL}^{-1}(bd : (E \times F) + (E \times G)) & & \\
 & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
 & & \mathbf{in}\, d_1 bd_2 \\
\mathrm{distR}(d) & = & d \\
\mathrm{distR}^{-1}(d) & = & d \\
\mathrm{wrap}\,(d) & = & d \\
\mathrm{wrap}\,^{-1}(d) & = & d \\
(c + c')(0d) & = & 0\ c(d) \\
(c + c')(1d') & = & 1\ c'(d')) \\
(c \times c')(d : E \times F) & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
 & & \mathbf{in}\, c(d_1)\ c'(d_2) \\
(c; c')(d) & = & c'(c(d)) \\
\mathrm{id}(d) & = & d \\
(\mathrm{fix}\, f.c)(d) & = & c[\mathrm{fix}\, f.c/f](d)
\end{array}
$$

Figure 13. Coercions operating on typed bit sequence representations instead of values

$$
\begin{array}{lcl}
\mathrm{split}(d : 1) & = & (\epsilon, d) \\
\mathrm{split}(d : a) & = & (\epsilon, d) \\
\mathrm{split}(0d : E + E') & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
 & & \mathbf{in}\,(0d_1, d_2) \\
\mathrm{split}(1d' : E + E') & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}(d' : E') \\
 & & \mathbf{in}\,(1d_1, d_2) \\
\mathrm{split}(d : E^*) & = & \mathrm{split}(d : 1 + E \times E^*) \\
\mathrm{split}(d : E \times E') & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
 & & \quad (d_3, d_4) = \mathrm{split}(d_2 : E') \\
 & & \mathbf{in}\,(d_1 d_3, d_4)
\end{array}
$$

Figure 14. Type-directed function for splitting bit sequence into subsequences corresponding to components of product type

$$
\frac{v : \alpha[\mu X.\alpha/X]}{\mathrm{fold}\, v : \mu X.\alpha}
$$

Figure 15. Inhabitation rule for μ

coercion judgements that operates directly on bit coded strings. See Figure 13. It uses the type-directed function split from Figure 14 for splitting a bit sequence into a pair of bit sequences.

Consider for example the coercion $\vdash c_0 : E^* \times E$ to $E \times E^*$ from Section 3.3. By interpreting c_0 according to Figure 13 we arrive at the following highly efficient function g_E, which transforms bit codings of values of $E^* \times E$ into corresponding bit representations for $E \times E^*$.

$$
\begin{array}{lcl}
g_E(0d) & = & 0d \\
g_E(1d) & = & 1\, f_E(d) \\
f_E(0d) & = & d0 \\
f_E(1d) & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}_E(d)\mathbf{in}\, d_1\, 1\, f_E(d_2)
\end{array}
$$

The bit coded version of $c_1 : E^* \leq 1 + E^* \times E$ gives us a linear-time function h_E that operates directly on bit codings of (parse trees) of strings in E^* and transforms them to bit codings in $1 + E^* \times E$:

$$
\begin{array}{lcl}
h_E(0d) & = & 0d \\
h_E(1d) & = & 1\, g_E(d)
\end{array}
$$

Note that $h_{(a+b)}$ is the bit coding coercion from H_1 to H_2. It transforms 10 11 10 10 11 0 into 1 10 11 10 10 0 1 without ever materializing a string or value.

4.3 Tail-recursive μ-types

The presented bit sequences are compact, but their sizes depend on the regular expression used. Thus it is necessary to use *reasonable* regular expressions to obtain compact bit sequences. In fact the most compact representations can be found only by generalizing regular expressions to *tail-recursive μ-types*. We will use the remainder of this section to study this extension and the compression it allows.

A common representation of strings over an alphabet
$\Sigma = \{a_1, \ldots, a_{255}\}$ of 255 characters from the Latin-1 (ISO/IEC 8859-1:1998) alphabet employs a sequence of 8-bit bytes representing each of the 255 different characters and uses the remaining byte to indicate the end of the string. This gives a total size of $8 \cdot (n + 1)$ bits to represent a string of length n.

Consider now the size of the bit sequence from Section 4.1 of a string under regular expression E_Σ^* where E_Σ is a sum type holding all the 255 characters in Σ. This can be written in many ways using permutations and associations of the characters. For example, if we define E_Σ as $a_1 + (a_2 + (a_3 + \ldots + a_{255}) \ldots)$ this means that the size of the bit coding of a_k is k bits long. This can be improved by ensuring that the type is balanced such that each path to a character has the same length. As there are 255 characters this means we will use 8 bits to represent each characters, leaving one path unused (so one character only uses 7 bits). Now we can look at the space required for the bit coding of a string under type E_Σ^*. Since E_Σ^* is unfolded to $1 + E_\Sigma \times E_\Sigma^*$ the representation of the empty string requires 1 bit, while the representation of other strings is 1 bit plus 8 bits for representing the first character, plus the bits to represent the rest of the string for the type E_Σ^*. Thus up to $9 \cdot n + 1$ bits are used to represent a string of length n.

The reason why bit codings for regular types use one bit more per character is due to the very restrictive recursion in regular expressions. The extra bit is used to say for each character that we do not want to end the string yet. This is because we can only use the $.^*$ constructor to define recursive types, and a regular expression E^* always unfolds to $1 + E \times E^*$. Therefore we use one bit for each character to choose the right hand side after the unfold. This is equivalent to using a unary integer representation to state how many times the E inside the $.^*$ is used, which leaves room for optimization.

We now generalize the recursion to *tail-recursive μ-types* in order to obtain more compact bit codings. Consider the language of expressions $\text{UnReg}^{\mu}_{\Sigma}$ over a finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$:

$$\alpha ::= 0 \mid 1 \mid a \mid \alpha_1 + \alpha_2 \mid \alpha_1 \times \alpha_2 \mid \mu X.\alpha \mid X$$

We define the free variables of α to be the set of variables X that occur in α without a binder μX. If there are no free variables in α then we say that α is closed. We call α tail-recursive if α_1 is closed in all subterms of the form $\alpha_1 \times \alpha_2$.

We can now define the language $\text{Reg}^{\mu}_{\Sigma}$ as the closed, tail-recursive expressions from $\text{UnReg}^{\mu}_{\Sigma}$.

We need to define semantics, type-interpretation and inhabitation for the new expressions, but we can reuse the definitions from regular expressions ($\mathcal{L}[\![]\!], \mathcal{T}[\![]\!], v : E$), simply by adding new rules for the new μ and X constructs.

We can extend the type-interpretation from Definition 6 with an environment σ, and replace the definition of $\mathcal{T}[\![E^*]\!]$ with

$\mathcal{T}[\![X]\!]_\sigma = \sigma(X)$ and $\mathcal{T}[\![\mu X.\alpha]\!]_\sigma = \bigcup_{i \geq 0} \mathcal{T}[\![\alpha]\!]_{\sigma^{(i)}}$ where $\sigma^{(0)} = \sigma[X \mapsto \emptyset]$ and $\sigma^{(n+1)} = \sigma[X \mapsto \mathcal{T}[\![\alpha]\!]_{\sigma^{(n)}}]$.

Similarly, $\mathcal{L}[\![.]\!]$ can be extended to $\text{Reg}^{\mu}_{\Sigma}$.

Finally, we can use the inhabitation rules from Figure 1, except the fold rule is replaced with the rule for μ in Figure 15.

We can now prove that $\text{Reg}^{\mu}_{\Sigma}$ expresses exactly the same languages as Reg_{Σ}:

THEOREM 30 (Conservativity of tail-recursive μ-types).

1. *For all $E \in Reg_\Sigma$ there is $\alpha \in Reg^{\mu}_{\Sigma}$ such that* $\{\text{flat}(v) \mid \vdash v : E\} = \{\text{flat}(v) \mid \vdash v : \alpha\}$.
2. *For all $\alpha \in Reg^{\mu}_{\Sigma}$ there is $E \in Reg_\Sigma$ such that* $\{\text{flat}(v) \mid \vdash v : \alpha\} = \{\text{flat}(v) \mid \vdash v : E\}$.

PROOF (Sketch) The first statement is proved by encoding E^* as $\mu X.1 + \alpha \times X$ where α is the encoding of E. The second statement is proved by first rewriting μ-types to the form $\mu X.(\alpha \times X + \beta)$, where X is not free in α or β. Now it can be seen that $\mathcal{L}[\![\mu X.(\alpha \times X + \beta)]\!] = \mathcal{L}[\![E^* \times F]\!]$ where E is a regular expression encoding of α and F is an encoding of β. □

The equivalence of regular expressions with right-regular grammars is well known (Chomsky 1959). Tail-recursive μ-types are like right-regular grammars, but do not *exactly* correspond to them: tail-recursive μ-types lack mutual recursion, but offer locally scoped recursion, where grammars only provide top-level recursion. [5]

Even though $\text{Reg}^{\mu}_{\Sigma}$ expresses exactly the same languages as Reg_{Σ}, the new expressions allow us to define $(a + b + c)^*$ as $\mu X.(1 + a \times X) + (b \times X + c \times X)$ and thus saving us one bit per character we need to express.

Using this optimization the representation of any string with respect to the generalized regular expression type α_Σ will use eight bits per Latin-1 character plus eight bits to terminate the string. This is exactly the same size as the standard Latin1-representation. In fact the bit-representations for this type will be exactly the same as the bit-representations for the standard Latin-1 representation if the same permutation of characters is chosen in α_Σ.

It may not seem very impressive to reinvent the Latin-1 representation this way, but it can guarantee that bit coding uses at most as much space as the Latin-1 representation. The benefit of bit coding comes when we no longer consider all Latin-1 strings, but a subset specified by a regular expression. In this case the bit codings will generally be more compact. The ultimate example of this is when the regular expression allows exactly one string. For example the bit sequence of `'abcbcba'` under regular expression $a \times b \times c \times b \times c \times b \times a$ uses *zero* bits since its value contains no $\text{inl}\,/\text{inr}$ -choices. Of course the program needs to know the regular expression in order to interpret the bit sequences, but that can be shared across interpretation of multiple bit sequences.

We end this section with two examples showing the bit sizes of strings under different regular expressions.

EXAMPLE 31. *In the table below Z denotes a designated end-of-string character; and characters a, b, c their 8-bit Latin-1 codings.*

Regular expression	*Representation*	*Size*
Latin1	*abcbcbaZ*	*64*
Σ^*	*1a1b1c1b1c1b1a0*	*64*
$((a+b)+(c+d))^*$	*1001011101011101011000*	*22*
$((a+b)+c)^*$	*10010111101111011000*	*20*
$a \times (b+c)^* \times a$	*10111011100*	*11*
$a \times b \times c \times b \times c \times b \times a$		*0*

The following is a more realistic example, where we also consider the data size before and after text compression.

EXAMPLE 32 (Sizes for XML record collection string). *Consider the following regular expression, corresponding to a regular XML schema (× and associativity have been omitted for simplicity).*

```
<CATALOG>
 (<CD><TITLE>Σ*</TITLE><ARTIST>Σ*</ARTIST>
     <COUNTRY>Σ*</COUNTRY><COMPANY>Σ*</COMPANY>
     <PRICE>Σ*</PRICE><YEAR>Σ*</YEAR>
 </CD>)*
</CATALOG>
```

This regular expression describes an XML-format for representing a list of CDs. We have found the sizes for representing a specific list containing 26 CDs to be as follows:

	Uncompressed	*Compressed*
Latin-1	*32760*	*7248*
bit representation using E_Σ	*11187*	*6654*
bit representation using α_Σ	*9947*	*6552*

As we can see, there is almost a factor 3 *reduction in the space requirement when using the regular expression specific bit codings. The benefit is reduced by compression of the bit codes with bzip (Seward) but an* 8% *space reduction is still obtained.*

5. Discussion

Regular expressions are fundamental to computer science with numerous applications in semi-structured text processing, natural language processing, program analysis, graph querying, shortest path computation, compilers, program verification, bioinformatics and more.

Salomaa (1966) and, independently, Aanderaa provided the first sound and complete axiomatizations of regular expression equivalence , based on a unique fixed point rule, with Krob (1990), Pratt (1990) (for extended regular expressions), and Kozen (1994) providing alternatives in the 90s.

Recently, coinductive axiomatizations based on finitary cases of Rutten's coinduction principle (Rutten 1998) for simulation relations have become popular: Grabmayer (2005) for regular expressions; Chen and Pucella (2004) and Kozen (2008) for Kleene Algebra with Tests. Silva, Bonsangue and Rutten show how to specialize their coalgebraic framework to regular languages and how to translate regular languages into so-called *deterministic expressions*

[5] Milner (1984) presents a sound and complete axiomatization of behavioral equivalence of μ-terms denoting labeled transition systems. Behavioral equivalence is properly weaker than regular expression equivalence, however. Crucially, distributivity $E \times (F + G) = E \times F + E \times G$ does not hold.

(Silva et al. 2010, Example 3.4). Such expressions appear to correspond to (nondeterministic) linear forms (Antimirov 1996), which in turn represent ϵ-free nondeterministic automata. (Their particular translation may generate exponentially bigger expressions than the original regular expressions, however, which may limit practical applicability.)

Grabmayer's axiomatization (Grabmayer 2005) is based on Brzozowski derivatives (Antimirov 1996; Brzozowski 1964), which allow automata constructions and pairwise regular expression rewriting (Antimirov 1995; Ginzburg 1967; Lu and Sulzmann 2004) to be understood as proof search.

In this paper we present a *declarative* coinductive axiomatization of regular expression containment, generalizing Grabmayer's *algorithmic* coinductive axiomatization of regular expression equality.[6] Ours is the first axiomatization of regular expression containment with a Curry-Howard-style computational interpretation of containment proofs as functions (coercions) operating on regular expressions read as *regular types*. Like Kozen's (noncoinductive) axiomatization, but unlike Salomaa's (noncoinductive) and Grabmayer's (coinductive) it is parametrically complete for infinite alphabets: If $E[X] \leq F[X]$ for all X, then there is a parametric polymorphic coercion $c : \forall X. E[X] \leq F[X]$.

Frisch and Cardelli (2004) were, as far we know, the first to state the precise connection between regular expressions as languages and regular expressions as types (Section 2.2 in this paper).

Coinductive axiomatizations with a Curry-Howard style computational interpretation have been introduced by Brandt and Henglein (1998) for recursive type equivalence and subtyping, expounded on by Gapeyev et al. (2002), as an alternative to the axiomatization of Amadio and Cardelli (1993), which uses the unique fixed point principle. In this fashion, classical unification closure can be understood as proof search for a type isomorphism, and the product automaton construction of Kozen et al. (1995) as search for the coercion embedding a subtype into another type. Di Cosmo et al. (2005) provide a coinductive characterization of recursive subtyping with associative-commutative products, but it is not a proper *axiomatization* since it appeals directly to bisimilarity.[7] Recursive type isomorphisms have also been studied by Abadi and Fiore (1996); Fiore (2004, 1996) and have been used for stub generation (Auerbach et al. 1999).

The (finitary) coinduction rule in its most general form is

$$\frac{\Gamma, P \vdash P}{\Gamma \vdash P}$$

It requires a side condition for soundness, which is usually specific to the syntax of the formulae P and the particular logical system at hand. Numerous syntactic variations may be possible, and care must be applied to retain soundness; e.g. by not allowing a transitivity rule (Gapeyev et al. 2002). This work represents the end of a quest for a general *semantic* side condition, at least for containment formulae: Interpret a *proof* of containment computationally as a function, and let the side condition be that the conclusion (now with explicit proof object) under this interpretation be (hereditarily) total. For regular expression containment, hereditary totality turns out to be undecidable, but it justifies the soundness of our side conditions S_2 and S_4, which each yield sound and complete axiomatizations of regular expression containment. Their disjunction is expressive enough to facilitate a compositional encoding of derivations in Salomaa's, Kozen's and Grabmayer's axiomatizations.

Brandt and Henglein (1998, Section 4.3) discuss how the finitary coinduction rule can be understood as a rule for finding a *finite* set of formulae that are intrinsically consistent. Intuitively this corresponds to constructing proofs of such formulae that are finite, but may be circular: they may contain occurrences of formulae to be proved as assumptions. This is also the essence of circular coinduction for behavioral equivalences (Rosu and Goguen 2000). Rosu and Lucanu (2009) provide a general proof-theoretic framework for applying circular coinduction soundly. It allows marking certain equations as *frozen* to prevent them from being applied prematurely, which would lead to unsound conclusions. Our approach is fundamentally different in the following sense: We allow circular equations without any restriction. An equation is interpreted as a pair of potentially partial functions, whose definition depends on the particular (circular) proof of the equation. An equation is *valid* (sound), however, only if the two functions making up its computational interpretation are total.

Our regular-expression specific bit representation of (parse trees of) strings corresponds to composing regular expression parsing with the flattening function of Jansson and Jeuring (1997), who attribute the technique to work in the 80s on text compression with syntactic source information models (see references in their paper). Bit coding captures the idea that only choices made—the tags of sum types—need to be encoded, which is also the key idea of oracle-based coding in proof-carrying code (Necula and Rahul 2001). Our direct compilation of containment proofs to bit manipulation functions appears to be novel, however. It should be noted that compaction by bit coding is orthogonal to statistical text data compression. As we have illustrated, combining them may yield significantly shorter bit strings than either technique by itself.

Type- and coercion-theoretic techniques appear to be applicable to *regular expression types* (Hosoya et al. 2005a,b; Nielsen 2008; Sulzmann and Lu 2007) and other nonregular extensions, notably context-free languages.[8] This remains to be investigated thoroughly, however. Note that regular expression types are a proper extension of regular expressions *as* types.

There are also numerous practically motivated topics to investigate: Inference of regular type containments and search for practically *efficient* coercions implementing them; disambiguation of regular expressions by annotating them; instrumenting automata constructions for fast input processing that yields parse trees; and more. We hope that this may lead the way to putting logic and computer science into a new generation of expressive, generally applicable high-performance regular expression processing tools.

Acknowledgements

We would like to thank the anonymous reviewers for their comprehensive critical and constructive comments, and for detailed recommendations for improvement. Any remaining problems are solely the authors' responsibility. The alphabetically first author is grateful to Eijiro Sumii, Yasuhiko Minamide, Naoki Kobayashi, and Atsushi Igarashi for jointly solving the question of decidability of hereditary totality (Theorem 21). We would like to thank Dexter Kozen for sharing many insights on Kleene Algebras as part of a mini-course held at DIKU, as well as for ideas and suggestions for the present–and for further–work. Thanks also to Alexandra Silva for explaining and commenting her work on Kleene coalgebras and its relation to regular expressions. Finally we would like to thank the participants of our graduate course "Topics in Programming Languages: Theory and Practice of Regular Expressions", held at DIKU in Spring 2010, for exploring some of the applications of regular expressions as types.

[6] Here, "declarative" and "algorithmic" are used in the same sense as in Benjamin Pierce's book *Types and Programming Languages*, MIT Press.

[7] Bisimilarity is coinductively defined (that is, as a greatest fixed point), but not necessarily *finitarily* as required in an ordinary (recursive) axiomatization.

[8] Completeness is out of the question, of course, since context-free grammar containment is not recursively enumerable.

References

M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proc. 1996 IEEE 11th Annual Symp. on Logic in Computer Science (LICS), New Brunswick, New Jersey*. IEEE Computer Society Press, June 1996.

R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, September 1993.

V. Antimirov. Rewriting regular inequalities. In *Proc. 10th International Conference, FCT '95 Dresden, Germany*, volume 965 of *Lecture Notes in Computer Science (LNCS)*, pages 116–125. Springer-Verlag, August 1995.

V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(95)00182-4.

V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theor. Comput. Sci.*, 143(1):51–72, 1995. doi: http://dx.doi.org/10.1016/0304-3975(95)80024-4.

J. S. Auerbach, C. Barton, M. Chu-Carroll, and M. Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *ICDCS*, pages 393–402, 1999.

C. Brabrand and J. Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. In *Proc. 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2010.

M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998.

J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321239.321249.

H. Chen and R. Pucella. A coalgebraic approach to Kleene algebra with tests. *Theor. Comput. Sci.*, 327(1-2):23–44, 2004.

N. Chomsky. On certain formal properties of grammars*. *Information and control*, 2(2):137–167, 1959.

J. H. Conway. *Regular Algebra and Finite Machines*. Printed in GB by William Clowes & Sons Ltd, 1971. ISBN 0-412-10620-5.

R. Di Cosmo, F. Pottier, and D. Remy. Subtyping recursive types modulo associative commutative products. In *Proc. Seventh International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, 2005.

M. Fiore. Isomorphisms of generic recursive polynomial types. *SIGPLAN Not.*, 39(1):77–88, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/982962.964008.

M. P. Fiore. A coinduction principle for recursive data types based on bisimulation. *Information and Computation*, 127:186–198, 1996. Conference version: Proc. 8th Annual IEEE Symp. on Logic in Computer Science (LICS), 1993, pp. 110-119.

A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture notes in computer science*, pages 618–629, Turku, Finland, July 2004. Springer.

V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.

A. Ginzburg. A procedure for checking equality of regular expressions. *J. ACM*, 14(2):355–362, 1967. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321386.321399.

C. Grabmayer. Using proofs by coinduction to find "traditional" proofs. In *Proc. 1st Conference on Algebra and Coalgebra in Computer Science (CALCO)*, number 3629 in Lecture Notes in Computer Science (LNCS). Springer, September 2005.

J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In J. Palsberg and M. Abadi, editors, *POPL*, pages 50–62. ACM, 2005a. ISBN 1-58113-830-X.

H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005b.

Institute of Electrical and Electronics Engineers (IEEE). *Standard for information technology — Portable Operating System Interface (POSIX) — Part 2 (Shell and utilities), Section 2.8 (Regular expression notation)*. New York, 1992. IEEE Standard 1003.2.

P. Jansson and J. Jeuring. Polyp—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, page 482. ACM, 1997.

S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, May 1994.

D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical report, Computing and Information Science, Cornell University, March 2008. URL `http://hdl.handle.net/1813/10173`.

D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science (MSCS)*, 5(1), 1995. Conference version presented at the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), 1993.

D. Krob. A complete system of b-rational identities. In M. Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 60–73. Springer, 1990. ISBN 3-540-52826-1.

K. Z. M. Lu and M. Sulzmann. Rewriting regular inequalities. In *Proc. Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004*, volume 3302 of *Lecture Notes in Computer Science (LNCS)*, pages 57–73. Springer, November 2004.

R. Milner. A complete inference system for a class of regular behaviours. *J. Comput. Syst. Sci.*, 28(3):439–466, 1984.

G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL*, pages 142–154, 2001.

L. Nielsen. A coinductive axiomatization of XML subtyping. Graduate term project report, DIKU, University of Copenhagen, 2008.

V. Pratt. Action logic and pure induction. In *Proc. Logics in AI: European Workshop JELIA*, volume 478 of *Lecture Notes in Computer Science (LNCS)*, pages 97–120. Springer, 1990.

G. Rosu and J. Goguen. Circular coinduction. In *Proc. International Joint Conference on Automated Reasoning*, 2000.

G. Rosu and D. Lucanu. Circular coinduction: A proof theoretical foundation. In *Proc. 3rd Conference on Algebra and Coalgebra in Computer Science (CALCO)*, number 5728 in Lecture Notes in Computer Science (LNCS), pages 127–144. Springer, September 2009. ISBN 978-3-642-03740-5.

J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorgi and R. de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998. ISBN 3-540-64896-8.

A. Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321312.321326.

J. Seward. Bzip. URL `http://www.bzip.org/`.

A. Silva, M. M. Bonsangue, and J. J. M. M. Rutten. Non-deterministic Kleene coalgebras. *Logical Methods in Computer Science*, 6(3), 2010. URL `http://arxiv.org/abs/1007.3769`.

M. Sulzmann and K. Z. M. Lu. XHaskell - adding regular expression types to Haskell. In O. Chitil, Z. Horváth, and V. Zsók, editors, *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2007. ISBN 978-3-540-85372-5.

G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing series. MIT Press, Feb. 1993. ISBN 0-262-23169-7.