Declarative Coinductive Axiomatization of Regular Expression Containment and its Computational Interpretation (Preliminary Version)

Fritz Henglein and Lasse Nielsen
DIKU, University of Copenhagen
{henglein, lnielsen}@diku.dk

March 22, 2010

Abstract

We present a new sound and complete coinductive 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, with $E^* = 1 + E \times E^*$ for Kleene-star and the general coinduction rule

$$\frac{\Gamma, E \leq F \vdash E \leq F}{\Gamma \vdash E \leq F}\text{(side condition for soundness)}$$

as the only additional rules. The axiomatization admits a natural Curry-Howard-style computational interpretation where regular expressions are straightforwardly interpreted as types containing not the strings themselves, but *proofs* of string membership in a regular expression. It turns out that this coincides with interpreting concatenation, alternation, empty set, empty string and Kleene star as product, sum, empty, unit and list types, respectively.

A proof of containment between regular expressions can then be interpreted *computationally* as a total function mapping a proof of string membership in one regular expression to a proof of membership in the containing regular expression for the same string. Computationally, the coinduction rule corresponds to definition by recursion, and its side condition in its most general form simply stipulates that the thus defined function be total, yielding soundness of our axiomatization. We provide a syntactically easily checkable totality criterion and show that Kozen's rules can be coded using it, yielding completeness.

We show how to synthesize containment proofs and how to construct and transform regular-expression-specific bit representations of strings.

Since membership proofs correspond to syntax trees, our computational interpretation of regular expressions become a *refinement type theory* for strings represented intensionally by their syntax trees. Regular expressions are mostly used for substring matching, a form of *parsing*, not just membership testing, and containment proofs retain syntactic information thrown away by automata constructions.

1 Introduction

Recall the regular expressions Reg_Σ over finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$:

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

where a ranges over Σ, with $\times$ binding stronger than $+$. In anticipation of our interpretation of regular expressions as types we write $\times$ instead of the more customary juxtaposition for sequential composition.

Each regular expression denotes a *regular language* $\mathcal{L}[\![E]\!] \subseteq \Sigma^*$. Regular expressions E and F are *equivalent* and we say $E = F$ holds (is valid), if $\mathcal{L}[\![E]\!] = \mathcal{L}[\![F]\!]$; we say E is *contained* in F, written $E \leq F$, if $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$. Equivalence and containment are easily related to each other since $E+F = F$ if and only if $E \leq F$, and $E = F$ if and only if both $E \leq F$ and $F \leq E$ hold; for this reason we shall informally switch back and forth between equivalence and containment in our discussions of axiomatizations.

1.1 Previous axiomatizations

At the core of all axiomatizations of regular expression equivalence are the axiomatization of product ($\times$), union ($+$), empty (0), unit (1) as the free idempotent semiring generated by the elements $a_1, \ldots a_n$. See Figures 1 and 2.

1.1.1 Salomaa

Salomaa provided the first sound and complete axiomatizations of regular expression equivalence [Sal66]. His System F_1 arises from adding the rules of Figure 3 to Figures 1 and 2. The *constant part* [Ant96] is defined as $o(F) = 1$ if $\epsilon \in \mathcal{L}[\![F]\!]$ where ϵ denotes the empty string; and $o(F) = 0$ otherwise. The side condition of the inference rule in Figure 3 is called the "no empty word property".

1.1.2 Kozen

Kozen showed that adding the rules in Figure 4 to Figures 1 and 2, which is his axiomatization of *Kleene Algebras*, is sound and complete for regular expression equivalence [Koz94].

He pointed out that Salomaa's rule

$$
\begin{array}{rcl}
E + (F + G) & = & (E + F) + G \\
E + F & = & F + E \\
E + 0 & = & E \\
E + E & = & E \\
E \times (F \times G) & = & (E \times F) \times G \\
1 \times E & = & E \\
E \times 1 & = & E \\
E \times (F + G) & = & (E \times F) + (E \times G) \\
(E + F) \times G & = & (E \times G) + (F \times G) \\
0 \times E & = & 0 \\
E \times 0 & = & 0
\end{array}
$$

Figure 1: 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 2: Rules of equality

$$
\begin{array}{rcl}
E^* & = & 1 + (E^* \times E) \\
E^* & = & (1 + E)^*
\end{array}
$$

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

Figure 3: Salomaa's rules for axiomatization F_1

$$
\begin{array}{rcl}
1 + (E \times E^*) & \leq & E^* \\
1 + (E^* \times E) & \leq & E^*
\end{array}
$$

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

Figure 4: 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 5: Definition of Brzozowski-derivative

$$E^* = 1 + E \times E^*$$

Figure 6: Fold/unfold rule for Kleene-star

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

is *nonalgebraic* in the sense that it is unsound under substitution of alphabet symbols by arbitrary regular expressions: With $E = a, F = b, G = c$ we have neither $a = a \times b + c$ nor $a = c \times b^*$, but after substituting $a \mapsto 1, b \mapsto 1, c \mapsto 0$ the antecedent $1 = 1 \times 1 + 0$ holds, but the consequent $1 = 0 \times 1^*$ does not. An 0-free illustration is $b \mapsto 1, c \mapsto a$: $a = a \times 1 + a$ holds, but $a^* = a \times 1^*$ does not.

1.1.3 Grabmeyer

The Brozowski-derivative E_a for regular expression E and $a \in \Sigma$ is defined in Figure 5. Observe that $E = o(E) + \sum_{i=1}^{n} a_i \times E_{a_i}$ can be derived using the rules for idempotent semirings (Figure 1) and equality (Figure 2) extended with the familiar *fold/unfold* axiom for Kleene-star in Figure 6.

Grabmeyer [Gra05] recognized that Brzozowski-derivatives can be combined with the ACI-properties of $+$ and Brandt and Henglein's coinductive fixed point rule for recursive types [BH98] to give a *coinductive* axiomatization of regular expression equivalence. Adding the rules of Figure 7 to Figures 1 and 2 results in a sound and complete axiomatization of regular expression equivalence. Here $[E = F]$ is a *hypothetical assumption*: It may be used an arbitary number of times in deriving the premise, but is *discharged* when applying the inference step. The given formulation is different from Grabmeyer's in the treatment of equality, but captures its essence.

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

Figure 7: Grabmeyer's coinduction rule

1.2 Our axiomatization

Grabmeyer's coinduction rule (Figure 7) can be understood as a specialization of the general *finitary coinduction* rule

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

with the following side condition (*), formulated informally:

> The derivation of the premise must contain subderivations of $E_{a_i} = F_{a_i}$ for all $i \in \{1 \dots n\}$.

Our goal is to find a more liberal side condition that guarantees soundness, yet is syntactically less restrictive. Note that we cannot simply elide the side condition since the coinduction rule is unsound without one: We could simply satisfy the premise by immediately concluding $E = F$ from the hypothetical judgment. By the coinduction rule $E = F$ for arbitrary E, F would be derivable.

The key idea of this paper is a general one: to make the side condition not a property of the premise itself, but of the *derivation* of the premise. To this end we equip our inference system with names for the rules and arrive at a type-theoretic formulation with explicit *proof terms*. We show that proof terms can be computationally interpreted as first-order functions transforming a *proof* of membership of a string in a regular expression to a proof of membership of the same string in another regular expression. For regular expression containment the coinduction rule then suggestively reads

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

as in Brandt and Henglein [BH98, Section 4.4]. The *computational interpretation* of $\mathrm{fix} f.\, c$ is the recursively defined function f such that $f(v) = c(v)$. (Note that c may contain free occurrences of f.)

Now we can state the side condition (*) for the coinduction rule in its most general form:

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

A variety of more restrictive, syntactically easily checkable side conditions are possible, each sufficient to ensure soundness (by totality) and completeness for regular expression containment. The side condition reformulating Grabmeyer's coinduction rule at the beginning of this section is one of them. We provide another one that is fundamentally different from previous axiomatizations and has a number of advantages.

1.3 Contributions

Before delving into the details we summarize our contributions.

1.3.1 Regular expressions as regular types

The central and, as we think conceptually most fundamental, contribution is our intensional interpretation of a regular expression as a set of proofs: An element of a regular expression is not a string, but a proof of membership of the string in that regular expression. Such proofs of membership coincide with syntax trees when viewing regular expressions as grammars, making them a theoretical and practical basis for regular expression *matching*. This provides a type-theoretic treatment of regular expressions as *regular types*.

Amazingly, no new theory is required: all we need are the standard type constructions of empty, unit, singleton, sum, product, and list types for interpreting regular expressions.

1.3.2 Computational interpretation of containment proofs

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

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

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

that retains the elements in the input. No previous regular expression axiomatization has been given a computational interpretation. What is more, the computational interpretation has evident practical applicability for regular expression *matching* (returning substring matches) where automata-based regular expression *membership testing* (just returning "yes" or "no") is insufficient.

1.3.3 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*: If $E[X_1, \ldots, X_m] \leq F[X_1, \ldots, X_m]$ 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 it for all instances of E.

Kozen's axiomatization [Koz94] is also parametrically complete, but neither Salomaa's [Sal66] nor Grabmeyer's [Gra05] 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 Grabmeyer's case E needs to be differentiated, the proof of which depends on the syntax of E.

We show that Kozen's rules are derived in our system. Our system has many more containment proofs, though. This makes proof synthesis easier and admits finding computationally more *efficient* proofs.

1.3.4 Computationally efficient proof encodings and transformation

If the regular expressions are statically known in a program the strings—actually their syntax trees—can be represented compactly using *oracle-based bit coding* of proofs, a technique known from proof coding in proof-carrying code [NR01]. We furthermore show how containment proofs can be compiled into transformations that operate directly on the bit codings without the need to construct proof values (syntax trees) explicitly.

1.4 Prerequisites

We assume basic knowledge of regular expressions as in Hopcroft and Ullman [HU79] and types in programming languages as in Pierce's book [Pie02].

$$\frac{}{\epsilon \in 1} \qquad \frac{}{() : 1}$$

$$\frac{}{a \in a} \qquad \frac{}{a : a}$$

$$\frac{s \in E}{s \in E + F} \qquad \frac{v : E}{\mathrm{inl}\, v : E + F}$$

$$\frac{s \in F}{s \in E + F} \qquad \frac{w : F}{\mathrm{inr}\, w : E + F}$$

$$\frac{s \in E \qquad t \in F}{s\, t \in E \times F} \qquad \frac{v : E \qquad w : F}{(v, w) : E \times F}$$

$$\frac{s \in 1 + E \times E^*}{s \in E^*} \qquad \frac{v : 1 + E \times E^*}{\mathrm{fold}\, v : E^*}$$

a) Membership proofs

b) Inhabitation proofs

Figure 8: Regular expression membership and inhabitation

2 Regular expressions as types

In this section we show that a regular expression E can be interpreted as a *type* whose elements are exactly the *proofs of membership* of strings in the regular expression denoted by E.

Let the *regular language* $\mathcal{L}[\![E]\!]$ denoted by E be compositionally defined as usual:

$$\begin{aligned}
\mathcal{L}[\![0]\!] &= \emptyset \\
\mathcal{L}[\![1]\!] &= \{\epsilon\} \\
\mathcal{L}[\![a]\!] &= \{a\} \\
\mathcal{L}[\![E + F]\!] &= \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!] \\
\mathcal{L}[\![E \times F]\!] &= \mathcal{L}[\![E]\!]\, \mathcal{L}[\![F]\!] \\
\mathcal{L}[\![E^*]\!] &= \bigcup_{i \geq 0} \mathcal{L}[\![E]\!]^i
\end{aligned}$$

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

Now consider the *membership relation* defined inductively by the inference system in Figure 8a. We write $\vdash s \in E$ if $s \in E$ is derivable in it. It characterizes the regular languages as follows:

Proposition 1. $\mathcal{L}[\![E]\!] = \{s \mid \vdash s \in E\}$

Figure 8a lets us focus on the membership *proofs* (derivations) themselves as the main objects instead of only *what* they prove. The practical

motivation comes from the *main* application of regular expressions: string matching, which is essentially *parsing* of strings under regular expressions und returning substring matches, not just determining whether a string is an element of (the language denoted by) a regular expression or not.

To get a term representation of membership proofs we give each inference rule a suggestive name; e.g. giving the rule

$$\frac{s \in E}{s \in E + F} \qquad (\mathrm{inl})$$

the name inl—we will see shortly why—we get the following *type-theoretic* rule with explicit proof terms:[1]

$$\frac{v : s \in E}{\mathrm{inl}\, v : s \in E + F}$$

Since it turns out that the string s in a statement $v : s \in E$ is uniquely determined by v we can elide it, and we end up with the inference system for statements of the form $v : E$ given in Figure 8b.

We call terms v such that $\vdash v : E$ for some E *membership proof values* or just *(proof) values*. A proof value determines the string of the membership statement it proves as follows:

Definition 1. *The* flattening $||v||$ *of proof value* v *is defined as follows:*

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

Proposition 2. $\vdash s \in E$ *if and only if there exists* v *such that* $\vdash v : E$ *and* $||v|| = s$.

A proof value v is essentially a *syntax tree* for s. What is more, Figure 8b shows that proof values are exactly the elements of regular expressions when interpreted as ordinary first-order types: 0 is the empty type, 1 the unit type, a (as a type) the singleton type containing $a \in \Sigma$ as the only element, $+$ the sum type constructor, $\times$ the product type constructor, and $.^*$ the list type constructor. When viewing them as types of proof values we shall refer to regular expressions as *regular types*. These are the types of Frisch and Cardelli [FC04]. Note that *regular expression types* introduced by Hosoya, Vouillon and Pierce [HVP05] are a proper extension.

[1] Type-theoretic for now just means that we derive explicit proof terms representing the derivation of a membership statement.

2.1 Ambiguity

The type-theoretic interpretation of regular expressions lets us easily define what it means for a regular expression to be *ambiguous*.

Definition 2 (Ambiguity)**.** *We say R is* unambiguous *if for all v, w, R such that $\vdash v : R$ and $\vdash w : R$ and $||v|| = ||w||$ we have $v = w$. R is* ambiguous *if it is not unambiguous.*

In other words, a regular expression R is unambiguous if each string can be parsed under R in at most one way.

What makes regular expression processing treacherous in practice is that many regular expressions are ambiguous, with multiple ways of resolving ambiguity [Van06] and that ambiguity is *not* invariant under regular expression equivalence.

Example 1. *Let $\Sigma = \{a, b\}$ and consider $H_1 = a^*$ and $H_2 = (1 + a)^*$. They are equivalent, but H_1 is unambiguous, and H_2 is ambiguous. Indeed each element s of $\mathcal{L}[\![H_2]\!]$ has* infinitely *many proofs: Informally, they consist of all the sequences over Σ extended with an* explicit symbol *ϵ having s as a subsequence and with all other symbols being ϵ.*

More generally, the proofs in equivalent regular expressions are not necessarily in one-to-one correspondence. Finite automata constructions for regular expressions, e.g. to NFAs with ϵ-transitions, then to ϵ-free NFAs, and eventually to DFAs, generally *reduce* ambiguity; in particular, they destroy parsing information under the *original* regular expression. We believe this explains in part why regular expression processing for substring matching in practice does *not* use standard automata constructions [Fri97] and abandons their performance advantages.

2.2 Regular expression containment by regular subtyping

Since each regular expression can be treated as an ordinary type inhabited by the syntax trees of strings parsed under the regular expression, we can formulate regular expression containment as the problem of transforming syntax trees under one regular expression into syntax trees under the other regular expression.

Proposition 3. *$\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$ if and only if for all $\vdash v : E$ there exists $\vdash w : F$ such that $||v|| = ||w||$.*

In particular, $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$ if and only if there exists a total function $f : E \to F$ such that $||f(v)|| = ||v||$. We write $\models c : E \leq F$ if c denotes such a function. We shall now present a "declarative" inference system for regular expression containment whose proofs can be interpreted computationally as such functions.

3 Declarative coinductive axiomatization

We restrict ourselves to axiomatizing regular expression containment since equivalence can be characterized as containment in both directions. We furthermore restrict ourselves to 0-free regular expressions; 0 and its rules are easily added separately.

Our axiomatization is given in Figure 9. It is given type-theoretically using judgements of the form $\Gamma \vdash c : E \leq E'$. The proof terms c are called *coercions* in anticipation of their functional interpretation. An equality axiom $\Gamma \vdash c : E = E'$ in Figure 9 represents the two containment axioms in both directions. We write c for the left-to-right containment, and c^{-1} for the right-to-left containment. The notation is suggestive as in each case the computational interpretation of c and c^{-1} will constititute an isomorphism. Note that we only have tagL as a rule since the dual tagR $: F \leq E + F$ is definable: tagR $=$ tagL ; retag.

Definition 3 (Computational interpretation)**.** *The* computational interpretation *of (closed) coercion $c : E \leq F$ is the partial function $\mathcal{F}[\![c]\!] : E \to F$ defined by $\mathcal{F}[\![c]\!](v) = w$ if $c(v) = w$ can be derived by the equational theory in Figure 10. We say c is* total *if $\mathcal{F}[\![c]\!]$ is total.*

Computational interpretations can be canonically extended to coercions containing free coercion variables.

3.1 Soundness

The last rule in Figure 9, the *coinduction rule*, is unsound without a side condition. The most general side condition (*) ensuring soundness is as follows: $\boxed{\text{fix}f.c \text{ is total}}$.

Since we don't know how hard it is to check for totality of coercions we use an efficiently checkable criterion which will turn out to be good enough for our purposes.

Definition 4 (Guarded coercion)**.** *We say f* occurs guarded *in c if*

- *c does not contain an occurrence of* proj^{-1} *or a free coercion variable to the left of some applied occurrence of f, and*
- *each occurrence of f occurs in a subexpression of c that has the form $d_1 \times d_2$.*

A coercion fix$f.c$ *is* locally guarded *iff f occurs guarded in c. A coercion c is* guarded *if all coercions of the form* fix$f.c$ *occurring in it are locally guarded.*

The guardedness condition ensures that each recursive call of a recursively defined coercion is called with an argument of strictly smaller size, where size is defined as follows:

$$
\begin{array}{lllrcl}
\Gamma & \vdash & \text{shuffle} & : & E + (F + G) & = & (E + F) + G \\
\Gamma & \vdash & \text{retag} & : & E + F & = & F + E \\
\Gamma & \vdash & \text{untag} & : & E + E & \leq & E \\
\Gamma & \vdash & \text{tagL} & : & E & \leq & E + F \\
\Gamma & \vdash & \text{assoc} & : & E \times (F \times G) & = & (E \times F) \times G \\
\Gamma & \vdash & \text{swap} & : & E \times 1 & = & 1 \times E \\
\Gamma & \vdash & \text{proj} & : & 1 \times E & = & E \\
\Gamma & \vdash & \text{distL} & : & E \times (F + G) & = & (E \times F) + (E \times G) \\
\Gamma & \vdash & \text{distR} & : & (E + F) \times G & = & (E \times G) + (F \times G) \\
\Gamma & \vdash & \text{fold} & : & 1 + E \times E^* & = & E^* \\
\Gamma & \vdash & \text{id} & : & E & = & E
\end{array}
$$

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

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

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

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

$$\frac{\Gamma, f : E \leq F \vdash c : E \leq F}{\Gamma \vdash \text{fix} f.c : E \leq F} \quad (*)$$

Figure 9: Declarative coinductive axiomatization of regular expression containment

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

Figure 10: Computational interpretation of coercions

Definition 5 (Size of proof)**.** *The size* $|v|$ *of a proof term is defined by:*

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

Note in particular the size of the unit value (): It is 1. If we set it to 0 we would have the *a priori* appealing property: If $\vdash v : E$ with $||v|| = s$ then $|v| = |s|$, where $|s|$ is the length of string s. In particular, each coercion, if it terminates, would preserve the size of its argument for the simple reason that the flattenings of its input and output are the same. The disadvantage is that under that size measure the component of a pair would not necessarily be properly smaller than the pair itself. Consider e.g. (v, w) and w. If $v = ()$ their sizes would be the same. Consequently, we would not be able to conclude from this size measure that coercion $\mathrm{fix} f.c$ terminates on (v, w) even if it calls itself recursively only with w. Intuitively, the unique fixed point axiomatization of Salomaa [Sal66] and the Brozowski-derivative based coinductive axiomatization of Grabmeyer [Gra05] can be thought of as adopting the size measure where () has size 0. To ensure that recursively defined coercions terminate they impose the additional constraint that the size of v must not be 0 in a recursive call as above: The regular expression E it comes from must not contain ϵ; that is, they require $o(E) = 1$.

Using our definition of size with $|()| = 1$ we get around having to impose this syntactic condition since we have the following strict inequalities:

Proposition 4. $|v| < |(v, w)|$ *and* $|w| < |(v, w)|$ *for all values* v, w.

Furthermore, $|v| \geq |s|$, where $s = ||v||$ and $|s|$ is the length of s. Intuitively, $|v| - |s|$ counts how often () occurs in v.

We can observe that proj *decreases* the size of its input by 1: $|\mathrm{proj}(v)| = |v| - 1$; and proj^{-1} increases it by 1: $|\mathrm{proj}^{-1}|(v) = |v| + 1$. All other primitive coercions c preserve the size of their inputs: $|c(v)| = |v|$.

In the body of a guarded coercion $\mathrm{fix} f.c$ an argument v of size n is processed by coercions that preserve or decrease n before calling f recursively since only size-nonincreasing coercions are applied. Furthermore each recursive call of f can only occur under a $\times$, which means it is passed a component whose size is necessarily strictly smaller than the size n of the original argument v passed to $\mathrm{fix} f.c$. As a consequence we get the following lemma:

Lemma 5. *Each guarded coercion is total.*

We can now formulate the side condition (*) in Figure 9 we shall use: $\boxed{\text{fix}f.c \text{ is guarded.}}$

Theorem 6 (Soundness)**.** *If* $\vdash c : E \leq F$ *then* $E \leq F$ *holds.*

Proof. (Sketch) By rule induction with Lemma 5 handling the case of the coinduction rule. Technically, we need to prove the obvious generalization to open coercions: For each derivable $\Gamma \vdash c : E \leq F$ we prove that $\Gamma \models c : E \leq F$ holds, which means that $\vdash \mathcal{F}[\![c]\!]^\rho(v) : F$ and $||\mathcal{F}[\![c]\!](v)|| = ||v||$ whenever $\vdash v : E$ and ρ is any mapping from coercion variables $f_i : E_i \leq F_i$ occurring freely in c to total functions f such that $||f(v)|| = ||v||$.

By Proposition 3 this shows that $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$ whenever there exists coercion c such that $\vdash c : E \leq F$. □

3.2 Completeness

We show now how to translate proofs in Kozen's axiomatization into the coercions of Figure 9, which implies the completeness of our system. Alternatively, a direct proof of completeness can be obtained by the coercion synthesis algorithm of Section 4.

The rules of Figure 1 correspond directly to rules in our system, and with a bit of care for Kleene-star also the equality rules work out straightforwardly.

Consider now the first rule of Figure 4: $E^* \leq 1 + E^* \times E$. First we prove $E^* \times E \leq E \times E^*$:

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

We are a bit informal here: We leave associativity steps implicit and write "by recursion" when applying the coinduction rule. For documentation we also write $=$ instead of $\leq$ whenever we apply a coercion that has an inverse.

The coercion c_0 representing the above derivation is as follows:

$$\begin{array}{l} \text{fix}f.(\text{fold}^{-1} \times \text{id}); \text{distR}; \\ \quad \text{swap} + (\text{assoc}^{-1}; \text{id} \times f); \\ \quad \text{distL}^{-1}; \text{id} \times \text{fold} \end{array}$$

Note that it is guarded and *parametric*: It actually proves the parametric statement $\forall X.X^* \times X \leq X \times X^*$. In particular, we can use the same c_0 for proving any substitution instance of $X^* \times X \leq X \times X^*$! Observe that

c_0 contains swap, but not proj^{-1}, and we do not need to worry about E containing the empty word or not.

Using c_0 we can define

$$c_1 = \text{fold}^{-1}; \text{id} + c_0 : E^* \leq 1 + E^* \times E.$$

This shows that $E^* \leq 1 + E^* \times E$ is derivable in our system.

The second rule in Figure 4 is $E^* \leq 1 + E \times E^*$. This follows immediately by the fold-coercion.

For the third rule, assume $f : E \times F \leq F$. We show that $E^* \times F \leq F$ is derivable.

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

Finally, for the fourth rule assume $g : E \times F \leq E$. We show that $E \times F^* \leq E$ is derivable.

$$\begin{array}{rcll} E \times F^* & \leq & E \times (1 + F \times F^*) & \text{by fold}^{-1} \\ & = & E \times 1 + E \times F \times F^* & \text{by distL} \\ & = & E + E \times F \times F^* & \text{by swap; proj} \\ & = & E + E \times F^* \times F & \text{previously proved} \\ & \leq & E + E \times F & \text{by recursion} \\ & \leq & E + E & \text{by } g \\ & = & E & \text{by untag} \end{array}$$

Theorem 7 (Completeness)**.** *If $E \leq F$ then there exists c such that $\vdash c : E \leq F$.*

Proof. Kozen has shown his axiomatization complete [Koz94]. The above codings show that each of his rules is derivable in our system. □

3.3 Parametric completeness

Let us extend regular expressions by adding variables that can be bound to arbitrary 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* $X_1, \ldots, X_i, \ldots$ and a over $\Sigma = \{a_1, \ldots, a_i, \ldots\}$. 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 without change to regular expressions with free variables. Without change it is not only complete, but *parametrically complete*:

Theorem 8 (Parametric completeness)**.** $\models \forall X_1, \ldots, X_m.E \leq F$ *if and only if* $\vdash E \leq F$.

Proof. Only if: By rule induction, our inference system (Figure 9) is closed under substitution.

If: Assume $\models \forall X_1, \ldots, X_n.E \leq F$. Consequently, $\models E\{X_1 \mapsto a_1, \ldots, X_n \mapsto a_n\} \leq F\{X_1 \mapsto a_1, \ldots, X_n \mapsto a_n\}$. By completeness we have $\vdash E\{X_1 \mapsto a_1, \ldots, X_n \mapsto a_n\} \leq F\{X_1 \mapsto a_1, \ldots, X_n \mapsto a_n\}$. By substitutivity of our rules we get $\vdash E \leq F$. □

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 synthesize a single coercion for it and use it for all instances of E.

Kozen's axiomatization [Koz94] is also parametrically complete, but neither Salomaa's [Sal66] nor Grabmeyer's [Gra05] 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 Grabmeyer's case E needs to be differentiated, the proof of which depends on the syntax of E. In comparison to Kozen's axiomatization [Koz94], which is parametrically complete, our system is coinductive and has more containment proofs. This makes proof (coercion) synthesis easier and admits finding computationally more *efficient* proofs when interpreted operationally as coercions.

4 Coercion synthesis

In the axiomatization of subtyping for regular expression types presented in the previous sections, it is not clear what strategy if any can be used to produce a coercion for a desired subtyping. In this section we will implement a strategy, always producing a coercion if possible, and otherwise finding a witness proving the subtyping is not fulfilled. We will do so by creating deterministic rules, that generate proof-terms in our existing coercion language. This allows coercion synthesis, since at most one rule is applicable at any time. The rules will be sound since we generate proof-terms that guarantee soundness, and the synthesis is complete since the derivation depth will be limited. The key instrument in this strategy is Brzozowski-derivatives of regular expressions [Brz64, Con71, Ant96] implemented below.

4.1 Differentiation

We call the process of finding the Brzozowski-derivatives of a regular expression for differentiation. The Brzozowski-derivatives are represented as regular expressions of the form

$$\begin{array}{ll} & D = 1 \\ \text{or} & D = Q_1 + (Q_2 + (\ldots + Q_n)\ldots) \\ \text{or} & D = 1 + (Q_1 + (Q_2 + (\ldots + Q_n)\ldots)) \\ \text{where} & Q ::= a_i \mid a_i \times E_i. \end{array}$$

For short we write $Q_1 + (Q_2 + (\ldots + Q_n)\ldots)$ as $\sum_{i=1}^{n} Q_i$.

The differentiation of E consists of generating a pair of coercions for $E \leq D$ and $D \leq E$ for some D. We require that proj^{-1} does not occur in the coercion of $E \leq D$, but may occur in the coercion of $D \leq E$ so as to ensure that all the coercions we construct are guarded.

If we only consider expressions of the form $D_1 + D_2 + \ldots + D_n$, differentiation can be performed simply by propagating the 1s to the left, using shuffle and retag, then implode the 1s to a single 1 using untag , and finally fixing the nesting of the +s using shuffle. This is implemented by the axiomatization of $\overset{\texttt{clean}}{=}$ in Figure 11.

If $E = \sum Q_i$ differentiation of $E \times F$ can be done simply by applying distR iteratively, and applying shuffle^{-1} to the necessary leaves. This is implemented by the axiomatization of $\overset{\texttt{dist}}{=}$ in Figure 12.

Using the $\overset{\texttt{clean}}{=}$ and $\overset{\texttt{dist}}{=}$ judgements, we can express the differentiation for all regular expressions denoted $\overset{\texttt{diff}}{=}$ in a straightforward fashion implemented in Figure 13. Notice that we have specified $E_1^\star \overset{\texttt{diff}}{\leq} D$ and $E_1^\star \overset{\texttt{diff}}{\geq} D$ separately when $E_1 \overset{\texttt{diff}}{=} 1 + \sum Q_i$, since $E_1 \overset{\texttt{diff}}{\leq} D$ cannot be reversed in this case[2].

Below we illustrate the implementation by differentiating the expression $1 + a^\star \times a$, where we find the differentiated expression to be $D = 1 + (a + Q)$ where $Q = a \times (a^\star \times a)$. We start by finding one of the sub-derivations concluding that $a^\star \overset{\texttt{diff}}{=} 1 + a \times a^\star$.

$$\mathcal{D}_1 = \dfrac{\texttt{dif-}\star_2 \dfrac{\texttt{dif-a}\ \dfrac{}{\vdash \text{id} : a \overset{\texttt{diff}}{=} a} \qquad \texttt{d-id}\ \dfrac{}{\vdash \text{id} : a \times a^\star \overset{\texttt{dist}}{=} a \times a^\star}}{\vdash \text{fold}^{-1}; (\text{id} + \text{id} \times \text{id}); \text{id} + \text{id} : a^\star \overset{\texttt{diff}}{=} 1 + a \times a^\star}}{}$$

We can now use $\mathcal{D}_1$ to find a sub-derivation concluding that $a^\star \times a \overset{\texttt{diff}}{=} a + Q$.

[2] $E_1^\star \overset{\texttt{diff}}{\geq} D$ cannot be reversed either because of its use of tagL

$$\texttt{c-11}\ \frac{}{\vdash \mathrm{untag} : 1 + 1 \overset{\texttt{clean}}{=} 1}$$

$$\texttt{c-12}\ \frac{}{\vdash \mathrm{shuffle}; \mathrm{untag} + \mathrm{id} : 1 + (1 + \sum Q_i) \overset{\texttt{clean}}{=} 1 + \sum Q_i}$$

$$\texttt{c-id}\ \frac{}{\vdash \mathrm{id} : D \overset{\texttt{clean}}{=} D}$$

$$\texttt{c-21}\ \frac{}{\vdash \mathrm{retag}; \mathrm{shuffle}; (\mathrm{retag}; \mathrm{untag}) + \mathrm{id} : (1 + \sum Q_i) + 1 \overset{\texttt{clean}}{=} 1 + \sum Q_i}$$

$$\texttt{c-22}\ \frac{\vdash c' : \sum Q_i + \sum Q'_i \overset{\texttt{clean}}{=} \sum Q''_i}{\vdash \begin{array}{l}\mathrm{shuffle}; \mathrm{retag} + \mathrm{id}; \mathrm{shuffle} + \mathrm{id}; \\ \mathrm{shuffle}^{-1}; (\mathrm{retag}; \mathrm{untag}) + c'\end{array} : (1 + \sum Q_i) + (1 + \sum Q'_i) \overset{\texttt{clean}}{=} 1 + \sum Q''_i}$$

$$\texttt{c-23}\ \frac{\vdash c' : \sum Q_i + \sum Q'_i \overset{\texttt{clean}}{=} \sum Q''_i}{\vdash \mathrm{shuffle}^{-1}; \mathrm{id} + c' : (1 + \sum Q_i) + \sum Q'_i \overset{\texttt{clean}}{=} 1 + \sum Q''_i}$$

$$\texttt{c-31}\ \frac{}{\vdash \mathrm{retag} : \sum Q_i + 1 \overset{\texttt{clean}}{=} 1 + \sum Q_i}$$

$$\texttt{c-32}\ \frac{\vdash c' : \sum Q_i + \sum Q'_i \overset{\texttt{clean}}{=} \sum Q''_i}{\vdash \mathrm{shuffle}; \mathrm{retag} + \mathrm{id}; \mathrm{shuffle}^{-1}; \mathrm{id} + c' : \sum Q_i + (1 + \sum Q'_i) \overset{\texttt{clean}}{=} 1 + \sum Q''_i}$$

$$\texttt{c-33}\ \frac{\vdash c' : \sum Q_i + \sum Q'_i \overset{\texttt{clean}}{=} \sum Q''_i}{\vdash \mathrm{shuffle}^{-1}; \mathrm{id} + c' : (Q_0 + \sum Q_i) + \sum Q'_i \overset{\texttt{clean}}{=} Q_0 + \sum Q''_i}$$

Figure 11: Definition of $\overset{\texttt{clean}}{=}$

$$\texttt{d-id}\ \frac{}{\vdash \mathrm{id} : a \times E \overset{\texttt{dist}}{=} a \times E}$$

$$\texttt{d-as}\ \frac{}{\vdash \mathrm{assoc}^{-1} : (a \times F) \times E \overset{\texttt{dist}}{=} a \times (F \times E)}$$

$$\texttt{d-rec}\ \frac{\vdash c_1 : Q_0 \times E \overset{\texttt{dist}}{=} Q'_0 \quad \vdash c_2 : (\sum Q_i) \times E \overset{\texttt{dist}}{=} \sum Q'_i}{\vdash \mathrm{distR}; c_1 + c_2 : (Q_0 + \sum Q_i) \times E \overset{\texttt{dist}}{=} Q'_0 + \sum Q'_i}$$

Figure 12: Definition of $\overset{\texttt{dist}}{=}$

$$\frac{}{\vdash \mathrm{id} : 1 \overset{\mathtt{diff}}{=} 1}\ \texttt{dif-1} \qquad \frac{}{\vdash \mathrm{id} : a \overset{\mathtt{diff}}{=} a}\ \texttt{dif-a}$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} D_1 \quad \vdash c_2 : E_2 \overset{\mathtt{diff}}{=} D_2 \quad \vdash c_3 : D_1 + D_2 \overset{\mathtt{clean}}{=} D}{\vdash (c_1 + c_2); c3 : E_1 + E_2 \overset{\mathtt{diff}}{=} D}\ \texttt{dif-+}$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} 1 \quad \vdash c_2 : E_2 \overset{\mathtt{diff}}{=} D_2}{\vdash c_1 \times c_2; \mathrm{proj} : E_1 \times E_2 \overset{\mathtt{diff}}{=} D_2}\ \texttt{dif-}\times_1$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} \sum Q_i \quad \vdash c_2 : (\sum Q_i) \times E_2 \overset{\mathtt{dist}}{=} \sum Q'_i}{\vdash c_1 \times \mathrm{id}; c_2 : E_1 \times E_2 \overset{\mathtt{diff}}{=} \sum Q'_i}\ \texttt{dif-}\times_2$$

$$\frac{\begin{array}{c}\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} 1 + \sum Q_i \quad \vdash c_2 : E_2 \overset{\mathtt{diff}}{=} D_2 \\ \vdash c_3 : (\sum Q_i) \times E_2 \overset{\mathtt{dist}}{=} \sum Q'_i \quad \vdash c_4 : D_2 + \sum Q'_i \overset{\mathtt{clean}}{=} D\end{array}}{\vdash c_1 \times \mathrm{id}; \mathrm{distR}; (\mathrm{proj}; c_2) + c_3; c_4 : E_1 \times E_2 \overset{\mathtt{diff}}{=} D}\ \texttt{dif-}\times_3$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} 1}{\vdash \mathrm{fix}\ f : E_1^\star = 1.\mathrm{fold}^{-1}; \mathrm{id} + (c_1 \times f); \mathrm{id} + \mathrm{proj}; \mathrm{untag} : E_1^\star \overset{\mathtt{diff}}{\leq} 1}\ \texttt{dif-}\star_{1\leq}$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} 1}{\vdash \mathrm{tagL}\,; \mathrm{fold}\ : E_1^\star \overset{\mathtt{diff}}{\geq} 1}\ \texttt{dif-}\star_{1\geq}$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} \sum Q_i \quad \vdash c_2 : (\sum Q_i) \times E_1^\star \overset{\mathtt{dist}}{=} \sum Q'_i}{\vdash \mathrm{fold}^{-1}; (\mathrm{id} + c_1 \times \mathrm{id}); \mathrm{id} + c_2 : E_1^\star \overset{\mathtt{diff}}{=} 1 + \sum Q'_i}\ \texttt{dif-}\star_2$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} 1 + \sum Q_i \quad \vdash c_2 : (\sum Q_i) \times E_1^\star \overset{\mathtt{dist}}{=} \sum Q'_i}{\vdash \begin{array}{l}\mathrm{fix}\ f : E_1^\star \overset{\mathtt{diff}}{\leq} 1 + \sum Q'_i. \\ \mathrm{fold}^{-1}; \mathrm{id} + c_1 \times \mathrm{id}; \mathrm{id} + \mathrm{distR}; \\ \mathrm{id} + (\mathrm{id} \times f + c_2); \mathrm{id} + (\mathrm{proj} + \mathrm{id}); \\ \mathrm{id} + \mathrm{retag}; \mathrm{shuffle}; \mathrm{untag}\end{array} : E_1^\star \overset{\mathtt{diff}}{\leq} 1 + \sum Q'_i}\ \texttt{dif-}\star_{3\leq}$$

$$\frac{\vdash c_1 : E_1 \overset{\mathtt{diff}}{=} 1 + \sum Q_i \quad \vdash c_2 : (\sum Q_i) \times E_1^\star \overset{\mathtt{dist}}{=} \sum Q'_i}{\vdash \mathrm{id} + (c_2^{-1}; (\mathrm{tagL}\,; \mathrm{retag}; c_1^{-1}) \times \mathrm{id}); \mathrm{fold}\ : E_1^\star \overset{\mathtt{diff}}{\geq} 1 + \sum Q'_i}\ \texttt{dif-}\star_{3\geq}$$

Figure 13: Definition of $\overset{\mathtt{diff}}{=}$

$$\dfrac{}{\vdash \mathrm{id} : 1 \overset{\mathtt{ewp}}{\leq} 1}\ \mathtt{ewp\text{-}1} \qquad \dfrac{}{\vdash \mathrm{tagL}\,;\mathrm{fold} : 1 \overset{\mathtt{ewp}}{\leq} E^\star}\ \mathtt{ewp\text{-}\star}$$

$$\dfrac{\vdash c_1 : 1 \overset{\mathtt{ewp}}{\leq} E \quad \vdash c_2 : 1 \overset{\mathtt{ewp}}{\leq} F}{\vdash \mathrm{proj}^{-1}; c_1 \times c_2 : 1 \overset{\mathtt{ewp}}{\leq} E \times F}\ \mathtt{ewp\text{-}\times}$$

$$\dfrac{\vdash c_1 : 1 \overset{\mathtt{ewp}}{\leq} E}{\vdash c_1; \mathrm{tagL} : 1 \overset{\mathtt{ewp}}{\leq} E + F}\ \mathtt{ewp\text{-}+_1} \qquad \dfrac{\vdash c_1 : 1 \overset{\mathtt{ewp}}{\leq} F}{\vdash c_1; \mathrm{tagR} : 1 \overset{\mathtt{ewp}}{\leq} E + F}\ \mathtt{ewp\text{-}+_2}$$

Figure 14: Big step semantics of the empty word property

$$\mathcal{D}_2 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathrm{id} : 1 \overset{\mathtt{diff}}{=} 1}\ \mathtt{dif\text{-}1} \quad \mathcal{D}_1, \quad \dfrac{}{\vdash \mathrm{id} : a \overset{\mathtt{diff}}{=} a}\ \mathtt{dif\text{-}a}\ , \quad \dfrac{}{\vdash \mathrm{assoc}^{-1} : (a \times a^\star) \times a \overset{\mathtt{dist}}{=} Q}\ \mathtt{d\text{-}as}\ , \quad \dfrac{}{\vdash \mathrm{id} : a + Q \overset{\mathtt{clean}}{=} a + Q}\ \mathtt{c\text{-}id}}{\vdash \begin{array}{l}(\mathrm{fold}^{-1}; (\mathrm{id} + \mathrm{id} \times \mathrm{id}); \mathrm{id} + \mathrm{id}) \times \mathrm{id}; \\ \mathrm{distR}; (\mathrm{proj}; \mathrm{id}) + \mathrm{assoc}^{-1}; \mathrm{id}\end{array} : a^\star \times a \overset{\mathtt{diff}}{=} a + Q}\ \mathtt{dif\text{-}\times_3}$$

We can now use $\mathcal{D}_2$ to express the desired derivation.

$$\dfrac{\dfrac{}{\vdash \mathrm{id} : 1 \overset{\mathtt{diff}}{=} 1}\ \mathtt{dif\text{-}1} \quad \mathcal{D}_2 \quad \dfrac{}{\vdash \mathrm{id} : D \overset{\mathtt{clean}}{=} D}\ \mathtt{c\text{-}id}}{\vdash \mathrm{id} + \left(\begin{array}{l}(\mathrm{fold}^{-1}; (\mathrm{id} + \mathrm{id} \times \mathrm{id}); \mathrm{id} + \mathrm{id}) \times \mathrm{id}; \\ \mathrm{distR}; (\mathrm{proj}; \mathrm{id}) + \mathrm{assoc}^{-1}; \mathrm{id}\end{array} \right); \mathrm{id} : 1 + a^\star \times a \overset{\mathtt{diff}}{=} D}\ \mathtt{dif\text{-}+}$$

The found coercion can be simplified to $\mathrm{id} + (\mathrm{fold}^{-1} \times \mathrm{id}; \mathrm{distR}; \mathrm{proj} + \mathrm{shuffle})$ by removing identity steps such ad $\mathrm{id} + \mathrm{id} \times \mathrm{id}$. Now we can verify that the found proof-term actually is a coercion of $1 + a^\star \times a = 1 + (a + a \times (a^\star \times a))$.

$$\begin{array}{rcll} 1 + a^\star \times a & = & 1 + (1 + a \times a^\star) \times a & \text{by } \mathrm{fold}^{-1} \\ & = & 1 + (1 \times a + (a \times a^\star) \times a) & \text{by distR} \\ & = & 1 + (a + (a \times a^\star) \times a) & \text{by proj} \\ & = & 1 + (a + a \times (a^\star \times a)) & \text{by } \mathrm{assoc}^{-1} \end{array}$$

Now we have implemented differentiation as $E \overset{\mathtt{diff}}{=} D$. It is straightforward to see that it is sound, complete and allows efficient proof-searching. Finally we have seen by example that it works.

4.2 Empty Word Property

The next step is to implement the empty word property. This basically says that if $\epsilon \in \mathcal{L}[\![E]\!]$ then we must be able to find a derivation of $1 \leq E$. This is implemented by the axiomatization of $\overset{\mathtt{ewp}}{\leq}$ in Figure 14.

4.3 Strategy

There are probably many complete search strategies, but we will present one that is based on Grabmeyers finitary coinduction principle [Gra05] using the above implementation of Brzozowski-derivatives [Brz64].

If we given E, F and Γ want to construct a coercion for $\Gamma \vdash c : E \leq F$, then the strategy is to differentiate both sides, to obtain coercions of $E \leq D_E$ and $D_F \leq F$. D_E will by definition be a sum of elements of the form $a_i \times E_i$ and a_i. Then we recursively find coercions

$$\Gamma, f_{E,F} : E \leq F \vdash c_i : E_i \leq \sum_{a_i \times F_j \text{ in } D_F} F_j$$

for each element of the form $a_i \times E_i$ in D_E, and return

$$\vdash c_i : 1 \overset{\texttt{ewp}}{\leq} \sum_{a_i \times F_j \text{ in } D_F} F_j$$

directly for each element of the form a_i in D_E. The types on the right hand sides are found by collecting all the elements that start with the given symbol, and then *discard* the symbol from each element. If a_i is in D_F then we rewrite it to $a_i \times 1$ and therefore 1 will be added to the sum. The rules implementing this are given in Figure 15 and 17.

If the recursion at some point returns to a previous *state* $E' \leq F'$ we can simply return $\Gamma'_1, f_{E',F'} : E' \leq F', \Gamma'_1 \vdash f_{E',F'} : E' \leq F'$. The problem is that this naive proof search does not always terminate, because the subterms can grow in size for each iteration.

An example of this problem is to consider $F = a^\star \times a^{\star^\star}$. If we differentiate F we get $1 + (a \times (a^\star \times a^{\star^\star}) + a \times (a^\star \times a^{\star^\star})) = 1 + (a \times F + a \times F)$, so

$$\sum_{a \times F_j \text{ in } D_F} F_j = F + F.$$

Therefore the first time we differentiate we get two elements in the sum, the second time we differentiate we get 4 elements and so on, and this of course never terminates.

It is a result of Brzozowski [Brz64], that we can fix this problem by eliminating multiple occurrences of the same element in the right sum and thus ensure the amount of derivatives for any regular expression is finite, proving that this strategy always terminates.

It is a cumbersome but trivial task to find coercions that collects and sorts all the elements with a given starting-symbol (Figure 15), eliminates multiple occurrences (Figure 16), collects all the elements starting with a given symbol in a single leading element (Figure 17), and finally sequences the above steps, and copies the resulting element for future use (Figure 18).

$$\frac{}{\vdash \mathrm{id} : b \times E \underline{\underline{\mathrm{sort}}}_a b \times E} \quad \frac{}{\vdash \mathrm{proj}^{-1}; \mathrm{swap}^{-1} : b \underline{\underline{\mathrm{sort}}}_a b \times 1}$$

$$\frac{\vdash c_1 : Q \underline{\underline{\mathrm{sort}}}_a b \times E \quad \vdash c_2 : F \underline{\underline{\mathrm{sort}}}_a \sum a_i \times F_i}{\vdash c_1 + c_2 : Q + F \underline{\underline{\mathrm{sort}}}_a b \times E + \sum a_i \times F_i}(a_1 \neq a \vee (b = a \wedge E \underline{\underline{\lesssim}}^{\mathrm{lex}} F_1))$$

$$\frac{\begin{array}{c}\vdash c_1 : Q \underline{\underline{\mathrm{sort}}}_a b \times E \quad \vdash c_2 : \sum Q_i \underline{\underline{\mathrm{sort}}}_a a \times F_0 + \sum Q'_i \\ \vdash c_3 : b \times E + \sum Q'_i \underline{\underline{\mathrm{sort}}}_a \sum Q''_i \end{array} \quad (b \neq a \vee F_0 \underline{\underline{\lesssim}}^{\mathrm{lex}} E)}{\vdash c_1 + c_2; \mathrm{shuffle}; \mathrm{retag} + \mathrm{id}; \mathrm{shuffle}^{-1}; \mathrm{id} + c_3 : Q + \sum Q_i \underline{\underline{\mathrm{sort}}}_a a \times F_0 + \sum Q''_i}$$

$$\frac{\vdash c_1 : Q \underline{\underline{\mathrm{sort}}}_a b \times E \quad \vdash c_2 : \sum Q_i \underline{\underline{\mathrm{sort}}}_a a \times F}{\vdash c_1 + c_2; \mathrm{retag} : Q + \sum Q_i \underline{\underline{\mathrm{sort}}}_a a \times F + b \times E}(b \neq a \vee F \underline{\underline{\lesssim}}^{\mathrm{lex}} E)$$

Figure 15: Sorting the elements of a sum with start symbol a.

$$\frac{}{\vdash \mathrm{id} : b \times E \underline{\underline{\mathrm{elim}}}_a b \times E} \quad \frac{}{\vdash \mathrm{id} : \sum a_i \times E_i \underline{\underline{\mathrm{elim}}}_a \sum a_i \times E_i}(a_1 \neq a)$$

$$\frac{\vdash c_1 : \sum Q_i \underline{\underline{\mathrm{elim}}}_a \sum a_i \times F_i}{\vdash \mathrm{id} + c_1 : a \times E + \sum Q_i \underline{\underline{\mathrm{elim}}}_a a \times E + \sum a_i \times F_i}(a_1 \neq a \vee E \not\equiv F'_1)$$

$$\frac{\vdash c_1 : \sum Q_i \underline{\underline{\mathrm{elim}}}_a a \times E + \sum Q'_i}{\vdash \mathrm{id} + c_1; \mathrm{shuffle}; \mathrm{untag} + \mathrm{id} : a \times E + \sum Q_i \underline{\underline{\mathrm{elim}}}_a a \times E + \sum Q'_i}$$

Figure 16: Eliminating multiple occurances in a sorted sum

$$\frac{\vdash c_1 : \sum Q_i \underline{\underline{\mathrm{join}}}_a a \times F}{\vdash \mathrm{id} + c_1; \mathrm{distL}^{-1} : a \times E + \sum Q_i \underline{\underline{\mathrm{join}}}_a a \times (E + F)}$$

$$\frac{}{\vdash \mathrm{id} : \sum_{i=1}^{n} a_i \times E_i \underline{\underline{\mathrm{join}}}_a \sum_{i=1}^{n} a_i \times E_i}(a_1 \neq a \vee n = 1)$$

$$\frac{\vdash c_1 : \sum Q_i \underline{\underline{\mathrm{join}}}_a \sum a_i \times F_i}{\vdash \mathrm{id} + c_1 : a \times E + \sum Q_i \underline{\underline{\mathrm{join}}}_a a \times E + \sum a_i \times F_i}(a_1 \neq a)$$

$$\frac{\vdash c_1 : \sum Q_i \underline{\underline{\mathrm{join}}}_a a \times F + \sum Q'_i}{\vdash \mathrm{id} + c_1; \mathrm{shuffle}; \mathrm{distL}^{-1} + \mathrm{id} : a \times E + \sum Q_i \underline{\underline{\mathrm{join}}}_a a \times (E + F) + \sum Q'_i}$$

Figure 17: Joining all leading elements starting with a in a sum

$$\frac{\vdash c_1 : \sum Q_i \underline{=}^{\mathrm{sort}_a} \sum Q'_i \quad \vdash c_2 : \sum Q'_i \underline{=}^{\mathrm{elim}_a} \sum Q''_i \quad \vdash c_3 : \sum Q''_i \underline{=}^{\mathrm{join}_a} a \times E + \sum Q'''_i}{\vdash c_1; c_2; c_3; \mathrm{tagL} + \mathrm{id}; \mathrm{shuffle}^{-1} : \sum Q_i \underline{=}^{\mathrm{fact}_a} a \times E + (a \times E + \sum Q'''_i)}$$

$$\frac{\vdash c_1 : \sum Q_i \underline{=}^{\mathrm{sort}_a} \sum Q'_i \quad \vdash c_2 : \sum Q'_i \underline{=}^{\mathrm{elim}_a} \sum Q''_i \quad \vdash c_3 : \sum Q''_i \underline{=}^{\mathrm{join}_a} a \times E}{\vdash c_1; c_2; c_3; \mathrm{tagL} : \sum Q_i \underline{=}^{\mathrm{fact}_a} a \times E + a \times E}$$

Figure 18: Factoring a in a sum

Now the final coercion can be constructed using the rules in Figure 19. Note that since the recursion c_i is always used in the context $\mathrm{id} \times c_i$ and since c_F^{-1}, c_{fact}^{-1} and c_{ewp} never occur before the recursion, then proj^{-1} will never occur before any recursion variable and therefore the resulting proof-term is guarded.

If we use the coercion synthesis on $a^\star \leq 1 + a^\star \times a$ we get the proof-term

$$\begin{aligned}
&\mathrm{fix}\ f : a^\star \leq 1 + a^\star \times a. && (1)\\
&\quad (\mathrm{fold}^{-1}; (\mathrm{id} + \mathrm{id} \times \mathrm{id}); \mathrm{id} + \mathrm{id}); && (2)\\
&\quad \mathrm{id} + (\mathrm{fix}\ g : a \times a^\star \leq a + a \times (a^\star \times). && (3)\\
&\qquad\qquad (\mathrm{id} \times \mathrm{id}; \mathrm{id}); \mathrm{id} \times f; \mathrm{tagL}\,; && (4)\\
&\qquad\qquad (\mathrm{untag}\,; (\mathrm{distL}; \mathrm{id} + \mathrm{id}); (\mathrm{id} + \mathrm{id}); ((\mathrm{swap}; \mathrm{proj}) + \mathrm{id})); && (5)\\
&\qquad\qquad (\mathrm{id}; \mathrm{id} + (\mathrm{id}; \mathrm{id} \times \mathrm{id}))); && (6)\\
&\quad (\mathrm{id} + (\mathrm{id}; (\mathrm{proj}^{-1}; \mathrm{id}) + \mathrm{assoc}; \mathrm{distR}^{-1}; (\mathrm{fold}\,; (\mathrm{id} + \mathrm{id} \times \mathrm{id}); \mathrm{id} + \mathrm{id}) \times \mathrm{id})) && (7)
\end{aligned}$$

Notice that (2) is the differentiation of $a^\star = 1 + a \times a^\star$, and (7) is the inverse of the differentiation $1 + a^\star \times a = 1 + (a + a \times (a^\star \times a))$.

By removing identity steps, and removing fix g since g is never used, we get the simplified coercion

$$\begin{aligned}
&\mathrm{fix}\ f : a^\star \leq 1 + a^\star \times a.\\
&\quad \mathrm{fold}^{-1};\\
&\quad \mathrm{id} + (\mathrm{id} \times f; \mathrm{tagL}\,; \mathrm{untag}\,; \mathrm{distL}; (\mathrm{swap}; \mathrm{proj}) + \mathrm{id});\\
&\quad \mathrm{id} + (\mathrm{proj}^{-1} + \mathrm{assoc}; \mathrm{distR}^{-1}; \mathrm{fold}\ \times \mathrm{id}).
\end{aligned}$$

Now we can verify that the coercion found actually translates from $a^\star$ to $1 + a^\star \times a$.

$$\texttt{syn-hyp}\ \frac{}{\Gamma, f : E \le F \vdash_{\text{syn}} f : E \le F}$$

$$\texttt{syn-id}\ \frac{\vdash c_E : E \overset{\text{diff}}{=} 1 \quad \vdash c_F : F \overset{\text{diff}}{=} 1}{\Gamma \vdash_{\text{syn}} c_E; c_F^{-1} : E \le F}$$

$$\texttt{syn-11}\ \frac{\vdash c_E : E \overset{\text{diff}}{=} 1 \quad \vdash c_F : F \overset{\text{diff}}{=} 1 + \sum Q'_i}{\Gamma \vdash_{\text{syn}} c_E; \text{tagL}\,; c_F^{-1} : E \le F}$$

$$\texttt{syn-E}\ \frac{\begin{array}{c}\vdash c_E : E \overset{\text{diff}}{=} a \quad \vdash c_F : F \overset{\text{diff}}{=} \sum Q'_i \quad \vdash c_1 : 1 \overset{\text{ewp}}{\le} F_0 \\ \vdash c_{\text{fact}} : \sum Q'_i \overset{\text{fact}_a}{=} a \times F_0 + \sum Q''_i\end{array}}{\Gamma \vdash_{\text{syn}} c_E; \text{proj}^{-1}; \text{swap}; \text{id} \times c_1; \text{tagL}\,; c_{\text{fact}}^{-1}; c_F^{-1} : E \le F}$$

$$\texttt{syn-R}\ \frac{\begin{array}{c}\vdash c_E : E \overset{\text{diff}}{=} a \times E_0 \quad \vdash c_F : F \overset{\text{diff}}{=} \sum Q'_i \\ \Gamma, f : E \le F \vdash_{\text{syn}} c_1 : E_0 \le F_0 \\ \vdash c_{\text{fact}} : \sum Q'_i \overset{\text{fact}_a}{=} a \times F_0 + \sum Q''_i\end{array}}{\Gamma \vdash_{\text{syn}} \text{fix} f.c_E; \text{id} \times c_1; \text{tagL}\,; c_{\text{fact}}^{-1}; c_F^{-1} : E \le F}$$

$$\texttt{syn-11R}\ \frac{\begin{array}{c}\vdash c_E : E \overset{\text{diff}}{=} 1 + \sum Q_i \quad \vdash c_F : F \overset{\text{diff}}{=} 1 + \sum Q'_i \\ \Gamma, f : E \le F \vdash_{\text{syn}} c_1 : \sum Q_i \le \sum Q'_i\end{array}}{\Gamma \vdash_{\text{syn}} \text{fix} f.c_E; \text{id} + c_1; c_F^{-1} : E \le F}$$

$$\texttt{syn-01R}\ \frac{\begin{array}{c}\vdash c_E : E \overset{\text{diff}}{=} \sum Q_i \quad \vdash c_F : F \overset{\text{diff}}{=} 1 + \sum Q'_i \\ \Gamma, f : E \le F \vdash_{\text{syn}} c_1 : \sum Q_i \le \sum Q'_i\end{array}}{\Gamma \vdash_{\text{syn}} \text{fix} f.c_E; c_1; \text{tagR}\,; c_F^{-1} : E \le F}$$

$$\texttt{syn-ER}\ \frac{\begin{array}{c}\vdash c_E : E \overset{\text{diff}}{=} a + \sum Q_i \quad \vdash c_F : F \overset{\text{diff}}{=} \sum Q'_i \quad \vdash c_1 : 1 \overset{\text{ewp}}{\le} F_0 \\ \Gamma, f : E \le F \vdash_{\text{syn}} c_2 : \sum Q_i \le \sum Q''_i \\ \vdash c_{\text{fact}} : \sum Q'_i \overset{\text{fact}_a}{=} a \times F_0 + \sum Q''_i\end{array}}{\Gamma \vdash_{\text{syn}} \text{fix} f.c_E; (\text{proj}^{-1}; \text{swap}^{-1}; \text{id} \times c_1) + c_2; c_{\text{fact}}^{-1}; c_F^{-1} : E \le F}$$

$$\texttt{syn-RR}\ \frac{\begin{array}{c}\vdash c_E : E \overset{\text{diff}}{=} a \times E_0 + \sum Q_i \quad \vdash c_F : F \overset{\text{diff}}{=} \sum Q'_i \\ \Gamma, f : E \le F \vdash_{\text{syn}} c_1 : E_0 \le F_0 \\ \Gamma, f : E \le F \vdash_{\text{syn}} c_2 : \sum Q_i \le \sum Q''_i \\ \vdash c_{\text{fact}} : \sum Q'_i \overset{\text{fact}_a}{=} a \times F_0 + \sum Q''_i\end{array}}{\Gamma \vdash_{\text{syn}} \text{fix} f.c_E; \text{id} \times c_1 + c_2; c_{\text{fact}}^{-1}; c_F^{-1} : E \le F}$$

Figure 19: Implementation of coercion synthesis

$$
\begin{array}{lcll}
a^\star & = & 1 + a \times a^\star & \text{by fold}^{-1} \\
& = & 1 + a \times (1 + a^\star \times a) & \text{by recursion } (f) \\
& = & 1 + (a \times (1 + a^\star \times a) + a \times (1 + a^\star \times a)) & \text{by tagL} \\
& = & 1 + a \times (1 + a^\star \times a) & \text{by untag} \\
& = & 1 + (a \times 1 + a \times (a^\star \times a)) & \text{by distL} \\
& = & 1 + (1 \times a + a \times (a^\star \times a)) & \text{by swap} \\
& = & 1 + (a + a \times (a^\star \times a)) & \text{by proj} \\
& = & 1 + (1 \times a + a \times (a^\star \times a)) & \text{by proj}^{-1} \\
& = & 1 + (1 \times a + (a \times a^\star) \times a) & \text{by assoc} \\
& = & 1 + (1 + a \times a^\star) \times a & \text{by distR}^{-1} \\
& = & 1 + a^\star \times a & \text{by fold}
\end{array}
$$

We have now seen that the found proof-term does translate between the desired types, but we can also see, that even after removing identity steps, and unused recursion definitions, it is still possible to improve the found coercion. For example it is possible to remove the tagL and untag step, since they cancel each other out. Similarly it is possible to remove the proj and proj^{-1} steps as they also cancel each other. The described method always gives a translation between the two types, but there is no guarantee that it will be the most computationally efficient translation. By following the size of data, it is possible to show that the running time of $c(v)$ will be in $\mathcal{O}(|v|)$. So unless there is a constant-time translation, it will only be possible to optimize the found translation by a constant factor.

Now we have seen that if $\models E \leq F$ then the strategy produces a coercion with type $E \leq F$. Consider now if $\models E \not\leq F$. This means that there is a string s which inhabits E but not F, we will call such a string a *witness*. We will now describe three states the coercion synthesis can reach, where it is possible to find a witness.

Case 1: The differentiated form of E' is $1 + \sum a_i \times E'_i$ and the differentiated form of F' is $\sum Q_i$. It is now obvious that $E' \not\leq F'$ since the empty string is a *witness*. We can now find a *witness* of $E \not\leq F$ simply by sequencing the a_i's we have *discarded* to reach this state.

Case 2: The differentiated form of E' has a as a summand, and the differentiated form of F' does not have a or $a \times F''$ where $1 \in \mathcal{L}[\![F'']\!]$ as a summand. It is now obvious that $E' \not\leq F'$ since the "a" is a *witness*. We can now find a *witness* of $E \not\leq F$ simply by prefixing the a_i's we have *discarded* to reach this state.

Case 3: The differentiated form of E' has $a \times E''$ as a summand, and the differentiated form of F' does not have a or $a \times F''$ as a summand. It is now obvious that $E' \not\leq F'$ since E' has an element that starts with an a, and F' does not. We can find an element s'' of E'' in the same way as the empty word search ($\overset{\text{ewp}}{\leq}$). We can now find a witness s' of $E' \leq F'$ by prefixing a to

s''. Finally we can find a *witness* of $E \not\leq F$ simply by prefixing the a_i's we have *discarded* to reach this state to s'.

It is a cumbersum but straightforward task to show that unless one of the above cases are fulfilled the proof synthesis can proceed. Since the proof search is sound and terminates, we can conclude that if $\models E \not\leq F$ then the search will reach one of the above states, and in each case we can find a witness.

We have now implemented the proof synthesis as $\vdash_{\text{syn}} E \leq F$ in a sound and complete way, yielding a proof-term $c : E \leq F$ if possible, and a witness $s \in \mathcal{L}[\![E]\!] \backslash \mathcal{L}[\![F]\!]$ otherwise.

4.4 Disambiguation principle

The strategy for coercion searching is forced to make some decissions, determining how values are encoded in the destination type. This is necessary when the destination type is ambiguous. Consider for example a proof-term c for the containment $\vdash c : a \leq a + a$. The only proof-value inhabiting a is a, while both $\operatorname{inl} a$ and $\operatorname{inr} a$ inhabits $a + a$. This means the functional interpretation of c can translate a to either $\operatorname{inl} a$ or $\operatorname{inr} a$. Both possibilities are realizable, for example tagL will choose $\operatorname{inl} a$ while tagR will choose $\operatorname{inr} a$. The way the coercions produced by a search strategy chooses between ambiguous representations can be called the disambiguation principle of that strategy. tagL is the only axiom increasing the ambiguity, which means the disambiguation principle is decided by the places a strategy produces tagL and untag coercions[3].

There are two different types of cases where the presented strategy produces tagL or untag coercions.

The first type occurs in the rules `ewp-`$\star$ used to prove $1 \overset{\text{ewp}}{\leq} E^\star$, `dif-`$\star_{3\geq}$ and `dif-`$\star_1$ applied on $E_1^\star$ when $E_1 \overset{\text{diff}}{=} 1 + \sum Q_i$ or $E_1 \overset{\text{diff}}{=} 1$, which means that E_1 can represent the empty string. In this case any proof-value v inhabiting $E_1^\star$, represents the same string as $\operatorname{fold} \operatorname{inr} (v', v)$ where v' is a proof-value producing the empty string for E_1. This alternative proof-value corresponds to producing the empty string for E_1 before producing the original string for $E_1^\star$. The choice made in these rules is to produce the string directly.

The second type occurs in $\overset{\text{elim}_a}{=}$, $\overset{\text{ewp}}{\leq}$ and other places including the final coercion synthesis rules. This type decides wether the left expression or the right expression in $E_1 + E_2$ is used. The choice made in these rules is to always use the left expression, which means that we use the leftmost possible branch. One exception is the rule `ewp-`$+_2$, since this allows the right expression to be used, but since `ewp-`$+_1$ allows the left expression to be used it is possible to make a coersion that uses the left expression whenever possible.

[3]Since tagL is used as the inverse of untag

We have now covered the places where the disambiguation principle is decided. It always uses as few unfolds as possible, by trying `ewp-`$+_1$ before `ewp-`$+_1$ the given proof synthesis always uses the leftmost possible branch.

5 Compact bit representations of syntax trees

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

5.1 Bit coded strings

Intuitively, a *bit coding* of a syntax 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 string $s = abaab$ as an element of $H_1 = (a + b)^*$. It has the unique syntax tree

$$\begin{array}{rcl} p_s & = & \mathrm{fold}\,(\mathrm{inr}\,(\mathrm{inl}\,a,\\ & & \mathrm{fold}\,(\mathrm{inr}\,(\mathrm{inr}\,b,\\ & & \mathrm{fold}\,(\mathrm{inr}\,(\mathrm{inl}\,a,\\ & & \mathrm{fold}\,(\mathrm{inr}\,(\mathrm{inl}\,a,\\ & & \mathrm{fold}\,(\mathrm{inr}\,(\mathrm{inr}\,b,\\ & & \mathrm{fold}\,(\mathrm{inl}\,())))))))))) \end{array}$$

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

By defining

$$\begin{array}{rcl} \mathrm{cons}\,(x, y) & = & \mathrm{fold}\,(\mathrm{inr}\,(x, y)) \\ \mathrm{nil} & = & \mathrm{fold}\,(\mathrm{inl}\,()) \end{array}$$

we get the more familiar-looking

$$p_s = \mathrm{cons}\,(\mathrm{inl}\,a, \mathrm{cons}\,(\mathrm{inr}\,b, \mathrm{cons}\,(\mathrm{inl}\,a, \mathrm{cons}\,(\mathrm{inl}\,a, \mathrm{cons}\,(\mathrm{inr}\,b, \mathrm{nil}\,)))))$$

with list constructors cons and nil. A graphical representation of p_s can be seen in Figure 20. Note that cons is a short hand for a fold-node with an inr-node as its sole child; likewise, nil is short-hand for a fold-node with an inl-node as its sole child.

The notation can be further simplified by employing bracket notation:

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

This reflects that the syntax tree of a string under a Kleene-star expression always is a list.

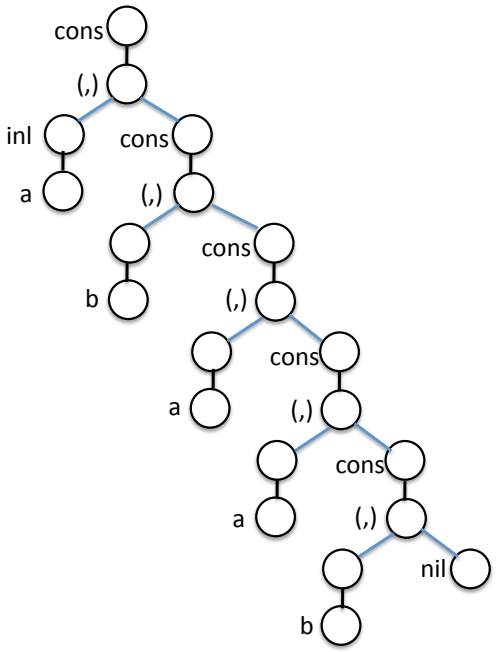

Figure 20: Syntax tree of $abaab$ under $H_1 = (a + b)^*$

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

Figure 21: Type-directed encoding function from syntax trees (proof values) to bit sequences

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

Figure 22: Type-directed decoding function from bit sequences to syntax trees (proof values)

Figures 21 and 22 define regular-expression directed coding and decoding functions code and decode from syntax trees to their *(canonical) bit codings* and back. Informally, the bit coding of a syntax 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 syntax tree p as a bit coding of the underlying string $||p||$. 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 syntax 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 different from b_s: it 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.

5.2 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 each return a bit-coded string, but under different regular expressions, and

we need to ensure that the result of the conditional is a bit coding in a common regular expression that contains the two.

Let us consider s again and how to transform b_s into b'_s. As we have seen in Section 3.2, 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 proof value $\vdash p : E^*$ representing a syntax tree for string $s' = ||p||$ to a syntax 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 21 and 22 to compute a function $\hat{c_1}$ operating on bit codings:

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

Instead of converting to and from proof values we can define a *bit coding coercion* by providing a computational interpretation of coercions that operates directly on bit coded strings. See Figure 23. It uses a type-directed function split shown in Figure 25 for splitting off the prefix of a bit sequence coding the first component of a pair.

Consider for example the coercion $\vdash c_0 : E^* \times E$ to $E \times E^*$ from Section 3.2. By interpreting c_0 according to Figure 23 we arrive at the following highly efficient function g_E, which transforms bit codings of proof values of $E^* \times E$ into corresponding bit representations for $E \times E^*$.

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

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 (syntax trees) of strings in E^* and transforms them to bit coddings 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 proof value.

5.3 Tail-recursive μ-types

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.

$$
\begin{array}{lcl}
\mathrm{retag}(0d) & = & 1d \\
\mathrm{retag}(1d) & = & 0d \\
\mathrm{retag}^{-1} & = & \mathrm{retag} \\
\mathrm{tagL}\,(d) & = & 0d \\
\mathrm{untag}\,(bd) & = & d \\
\mathrm{shuffle}(0d) & = & 00d \\
\mathrm{shuffle}(10d) & = & 01d \\
\mathrm{shuffle}(11d) & = & 1d \\
\mathrm{shuffle}^{-1}(00d) & = & 0d \\
\mathrm{shuffle}^{-1}(01d) & = & 10d \\
\mathrm{shuffle}^{-1}(1d) & = & 11d \\
\mathrm{swap}(d) & = & d \\
\mathrm{swap}^{-1} & = & \mathrm{swap} \\
\mathrm{proj}(d) & = & d \\
\mathrm{proj}^{-1}(d) & = & d \\
\mathrm{assoc}(d) & = & d \\
\mathrm{assoc}^{-1}(d) & = & d \\
\mathrm{distL}(d : E \times (F + G)) & & \\
\qquad & = & \mathbf{let}\,(d_1, bd_2) = \mathrm{split}(d : E) \\
\qquad & & \mathbf{in}\,bd_1d_2 \\
\mathrm{distL}^{-1}(bd : (E \times F) + (E \times G)) & & \\
\qquad & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
\qquad & & \mathbf{in}\,d_1bd_2 \\
\mathrm{distR}(d) & = & d \\
\mathrm{distR}^{-1}(d) & = & d \\
\mathrm{fold}\,(d) & = & d \\
\mathrm{fold}^{-1}(d) & = & d \\
(c + c')(0d) & = & 0\ c(d) \\
(c + c')(1d') & = & 1\ c'(d')) \\
(c \times c')(d : E \times F) & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
\qquad & & \mathbf{in}\,c(d_1)\ c'(d_2) \\
(c; c')(d) & = & c'(c(d)) \\
\mathrm{id}(d) & = & d \\
(\mathrm{fix}\ f.c)(d) & = & c[\mathrm{fix}\ f.c/f](d)
\end{array}
$$

Figure 23: Coercions operating on typed bit sequence representations instead of proof 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 24: 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 25: Inhabitation rule for μ

Consider now the size of the bit coding from Section 5.1 of a string under regular expression E_Σ^* where E_Σ is a sum type holding of 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 $E_\Sigma^\star$. Since $E_\Sigma^\star$ is unfolded to $1 + E_\Sigma \times E_\Sigma^\star$ the representation of the empty string requires 1 bit, while the representation of other strings is 1 bit plus the 8 bits for representing the first character, plus the bits to represent the rest of the string for the type $E_\Sigma^\star$. Thus $9 \cdot n + 1$ 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 $\star$ constructor to define recursive types, and a regular expression $E^\star$ always unfolds to $1 + E \times E^\star$.

We now generalize the recursion to *tail-recursive μ-types* in order to obtain more compact bit codings.

Consider the language of expressions $\mathrm{UnReg}_\Sigma^\mu$ 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 unguarded by 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 $\mathrm{Reg}^{\mu}_{\Sigma}$ as the closed, tail-recursive expressions from $\mathrm{UnReg}^{\mu}_{\Sigma}$.

We need to define inhabitation for the new expressions, but we can reuse the rules from Figure 8, except we need to rewrite the fold-rule to match $\mu X.\alpha$. This rule is given in Figure 24 using capture-avoiding substitution. We can now prove that $\mathrm{Reg}^{\mu}_{\Sigma}$ expresses exactly the same languages as Reg_{Σ}:

Theorem 9 (Conservativity of tail-recursive μ-types over regular types)**.**

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

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

Proof. (Sketch)

1. The first statement is proved by encoding $E^{\star}$ as $\mu X.1 + \alpha \times X$ where α is the encoding of E.

2. The second statement is proved by first rewriting μ statements to the form $\mu X.(\sum \alpha_i \times X + \sum \beta_i)$, where X is not free in any α_i or β_i. Now the language equality is true for the regular expression $E_1^{\star} \times E_2$ where E_1 is an encoding of $\sum \alpha_i$ and E_2 is an encoding of $\sum \beta_i$.

□

Even though $\mathrm{Reg}^{\mu}_{\Sigma}$ expresses exactly the same languages as Reg_{Σ}, the new expressions allow us to define $(a{+}b{+}c)^{\star}$ as $\mu X.(a{\times}X{+}b{\times}X){+}(c{\times}X{+}1)$ 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 Σ_{μ} will use exactly eight bits per 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 Σ_{μ}.

It may not seem very impressive to have reinvented the Latin1 representation this way, but the benefit occurs when we no longer consider all Latin1 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 representation of `'abcbcba'` under regular expression $a \times b \times c \times b \times c \times b \times a$ uses *zero* bits since its proof value contains inl /inr -choices for a sum type must be made in its coding.

We end this section with two examples showing the number of bits used to represent a string for different types, and the datasize of a real world example with Latin1 and the described bit representation before and after bzip [Sew] compression.

Example 2 (Expressiveness vs. datasize)**.**

Type	*Representation*	*Size*
Latin1	*abcbcba*	*64*
$\Sigma^\star$	*1a1b1c1b1c1b1a0*	*64*
$((a+b)+(c+d))^\star$	*1001011101011101011000*	*22*
$((a+b)+c)^\star$	*100101111011111011000*	*20*
$a \times (b+c)^\star \times a$	*10111011100*	*11*
$a \times b \times c \times b \times c \times b \times a$		*0*

The following is a more real world example. We consider the data size before and after compression.

Example 3 (Sizes for XML record collection string)**.** *Consider the following regular expression corresponding to a regular XML schema ($\times$ 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 the following

	Uncompressed	*Compressed*
Latin-1	*32760*	*7248*
bit representations	*11187*	**6654*

**) Where the selection bits are not compressed.*

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, but there is still an* 8% *reduction in the space used.*

6 Related and future work

Salomaa provided the first sound and complete axiomatizations for regular expression equivalence [Sal66] based on a unique fixed point rule, with Krob [Kro90], Pratt [Pra90] (for extended regular expressions) and Kozen [Koz94] providing alternatives.

Recently coinductive axiomatizations based on finitary cases of Rutten's coinduction principle [Rut98] for simulation relations have become popular: Grabmeyer for regular expressions [Gra05], and Chen and Pucella [CP04] and Kozen [Koz08] for Kleene Algebra with Test. They are based on Brzozwoski derivatives [Brz64, Ant96] and allow automata constructions and pairwise regular expression rewriting [Gin67, Ant95, LS04] to be understood as proof search. In these systems, the coinduction rule may only be applied to expressions in Brzozowski normal form. Ours is the first axiomatization of regular expression containment that eliminates such syntactic restrictions and gives a Curry-Howard style computational interpretation of containment proofs as functions (coercions) operating on regular expressions read as *regular types*, which are constructed from unit, singleton, product, sum and list types. Like Kozen's (noncoinductive) axiomatization, but unlike Salomaa's (noncoinductive) and Grabmeyer's (coinductive) it is parametrically complete: If $E[X] \leq F[X]$, then there is a parametric coercion $c : E[X] \leq F[X]$ showing this.

Coinductive axiomatizations with a Curry-Howard style computational interpretation have been introduced by Brandt and Henglein [BH98] for recursive type equivalence and subtyping, expounded on by Gapeyev Levin Pierce[GLP02], as an alternative to Amadio and Cardelli's axiomatization [AC93], which uses the unique fixed point principle. In this fashion classical unification closure can be understood as proof search for a type isomorphism, and Kozen, Palsberg and Schwartzbach's product automaton construction [KPS95] as search for the coercion embedding a subtype into another type. Cosmo, Pottier, and Remy [DCPR05] provide a coinductive characterization of recursive subtyping with associative-commutative products, but it is not a proper *axiomatization* since it appeals directly to bisimilarity.[4] Recursive type isomorphisms have also been studied by Abadi and Fiore [AF96, Fio96, Fio04] and have been used for stub generation [ABCCR99].

Our regular-expression specific bit representation of (syntax trees of) strings corresponds to the oracle-based coding of proofs in proof-carrying code [NR01]. Our direct compilation of containment proofs to bit manipulation functions appears to be novel, however. It should be noted that this form of compaction is orthogonal to (string) data compression. Indeed combining both may yield significantly shorter bit strings than either technique

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

by itself.

Our techniques appear to be applicable to *regular expression types* [HVP05, HFC05, SL07, Nie08] and other nonregular extensions, notably context-free languages (completeness is out of the question, of course); this still needs to be investigated further, however. 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 parsing and more. We hope that this may lead the way to putting logic and computer science into a new generation of regular expression processing.

References

[ABCCR99] Joshua S. Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *ICDCS*, pages 393–402, 1999.

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, September 1993.

[AF96] Martín Abadi and Marcelo 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.

[Ant95] Valentin 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.

[Ant96] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[BH98] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines*. Printed in GB by William Clowes & Sons Ltd, 1971.

[CP04] Hubie Chen and Riccardo Pucella. A coalgebraic approach to kleene algebra with tests. *Theor. Comput. Sci.*, 327(1-2):23–44, 2004.

[DCPR05] Roberto Di Cosmo, Francois Pottier, and Didier Remy. Subtyping recursive types modulo associative commutative products. In *Proc. Seventh International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, 2005.

[FC04] 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.

[Fio96] Marcelo 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.

[Fio04] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. *SIGPLAN Not.*, 39(1):77–88, 2004.

[Fri97] Jeffrey Friedl. *Master Regular Expressions—Powerful Techniques for Perl and Other Tools.* O'Reilly, 1997.

[Gin67] A. Ginzburg. A procedure for checking equality of regular expressions. *J. ACM*, 14(2):355–362, 1967.

[GLP02] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.

[Gra05] Clemens Grabmeyer. 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.

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for xml. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 50–62. ACM, 2005.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[HVP05] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[Koz94] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, May 1994.

[Koz08] Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical report, Computing and Information Science, Cornell University, March 2008.

[KPS95] Dexter Kozen, Jens Palsberg, and Michael 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.

[Kro90] Daniel Krob. A complete system of b-rational identities. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 60–73. Springer, 1990.

[LS04] Kenny Zhuo Ming Lu and Martin 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.

[Nie08] Lasse Nielsen. A coinductive axiomatization of XML subtyping. Graduate term project report, DIKU, University of Copenhagen, 2008.

[NR01] George C. Necula and Shree Prakash Rahul. Oracle-based checking of untrusted software. In *POPL*, pages 142–154, 2001.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[Pra90] Vaughan 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.

[Rut98] Jan J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[Sew] Julian Seward. Bzip.

[SL07] Martin Sulzmann and Kenny Zhuo Ming Lu. Xhaskell - adding regular expression types to haskell. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2007.

[Van06] Stijn Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006.