Disambiguation and Faithful Instantiation of Parametric Regular Expression Containment Coercions

Abstract

We consider the proofs-are-programs principle in the setting of containment among regular expressions. A proof of containment is a coercion function among parse trees. Based on the concept of Antimirov's partial derivatives, we develop a novel method to derive coercions out of containment proofs. Our approach naturally supports parametric coercions among parametric regular expressions where coercions satisfy the greedy left-most disambiguation strategy. We establish sufficient conditions which guarantee that parametric coercions retain the greedy left-most property under instantiation. The approach is implemented and provides for an important step to achieve for a satisfactory adoption of regular expression as types in the context of parametrically typed programming languages.

1. Introduction

We consider the proofs-are-programs principle, also known as Curry-Howard correspondence [14], in the setting of regular expressions. Regular expressions are interpreted as regular types [5, 13]. Values of a regular type serve as proofs of membership in the language denoted by the regular expression. In addition, proof values represent parse trees and explain which subexpressions match which substrings.

This view extends to the containment relation among regular expressions. Two regular expressions are in containment relation if there exists a transformation function mapping parse trees of one regular expression into parse trees of the other regular expression [9]. We refer to these transformation functions as *coercions*. In general, coercions may be ambiguous which is no surprise given that parsing of regular expressions is ambiguous. To adopt regular expressions as types in a programming language setting, it is paramount that they are deterministic and can be used predictable.

In this paper, we introduce a novel method to construct coercion functions which, unlike earlier works [9, 16], is shown to respect the *greedy left-most disambiguation* strategy. Our method naturally supports parametric coercions involving containment among parametric regular expressions. In general, parametric coercions are not faithful under instantiation. That is, for a specific instance the parametric coercion behaves differently compared to the monomorphic coercion derived out of the instantiated containment relation. Previous works [10, 20] imposes some severe conditions to guarantee faithfulness. We show how to relax these conditions.

In summary, we make the following main contributions:

- We show how to derive coercions out of regular expression containment proofs which respect the greedy left-most disambiguation strategy (Section 4.4).
- For a parametric coercion and a specific instance we identify necessary and sufficient conditions so that the parametric coercion behaves exactly like the coercion derived from the instantiated case (Section 4.5).
- The system to derive coercions has been fully implemented (Section 5).

The up-coming section givens an overview of the key ideas of our work. Section 3 covers some background material on regular expressions as types and partial derivatives. Related work is discussed in Section 6.

We generally assume that the reader is familiar with the basics of regular expressions, logic, types and lambda calculus.

2. Coercions, Disambiguation and Faithfulness

Regular Expressions as Types Checking that regular expression r matches word w boils down to a language membership test $w \in L(r)$. By applying the proofs-are-programs principle, we can rephrase the matching problem as a type inhabitation problem where we interpret regular expressions as types and terms represent proof of language membership [9]. For example, consider the statement

$$\vdash \mathit{Left}\ a : a + b$$

Proof term $\mathit{Left}\ a$ represents a compact representation of a parse tree, explaining that the left sub-pattern in $a + b$ matches word a.

Similarly, the proof of language containment can be represented as a coercion function among parse trees. For example, consider regular expressions a and $a+b$. The former expression is contained in the latter as shown by coercion $\lambda v.\mathit{Left}\ v$.

Containment via Partial Derivatives Our approach to derive coercions out of containment proofs is based on the use of partial derivatives [1, 2]. The idea is as follows. Regular expressions r are normalized to the form $(l_1, r_1) + ... + (l_n, r_n)$ where l are literals and r_i are alternatives among partial derivatives. Thus, a containment problem such as $r \leq r'$ can be broken into some 'smaller' pieces $r_i \leq r'_i$. For example, consider the following proof sketch for containment for $r \leq r'$:

$$\dfrac{\dfrac{\dfrac{\vdash r \leq r'}{\vdots}\ (\text{CoIn})}{\vdash r_1 \leq r'_1 ... \vdash r_n \leq r'_n}\ (\text{Lit}) \quad r = (l_1, r_1) + ... + (l_n, r_n) \quad r' = (l_1, r'_1) + ... + (l_n, r'_n)}{\vdash r \leq r'}$$

The proof is shown as a derivation tree starting with the to be proven statement at the bottom. Proof step (Lit) breaks the proof into 'smaller' pieces r_i and r_i' by removing leading literals. The partial derivative-based containment proof system applies coinduction to terminate a proof in case the to be proven statement $r \leq r'$ is rediscovered in some subproof. See the last proof step (CoIn).

Coercions as Injection and Projection Out of each containment proof $\vdash\ r \leq r'$ we compute two functions *inj* and *proj*. Function *inj* injects any of r's parse trees into a parse tree of r'. Function *proj* tries to project any of r' parse trees onto a parse tree of r. Their type signatures are as follows

$$\begin{array}{lcl} \mathit{inj}_{(r,r')} & :: & v_r \rightarrow v_r' \\ \mathit{proj}_{(r,r')} & :: & v_r' \rightarrow \mathit{Maybe}\ v_r \end{array}$$

The functions are indexed by types because the functions bodies will be generated specifically for the types involved in the containment proof. In the above, we write v_r to denote a proof term belonging to regular expression r.

Function *proj* is only a 'partial' function because not every parse tree of the larger expression can be projected onto a parse tree of the smaller expression. Hence, the 'maybe' result. For example, consider projection for $\vdash\ a \leq a + b$:

$$\begin{array}{l} \lambda.\ v\ \mathsf{case}\ v\ \mathsf{of} \\ \quad \mathit{Left}\ v_a\ \rightarrow\ \mathit{Nothing} \\ \quad \mathit{Right}\ v_b\ \rightarrow\ \mathit{Just}\ v_b \end{array}$$

The construction of coercions is guided by the shape of the containment proof. In case of a coinduction proof step (CoIn), we simply build a recursively defined coercion. The resulting coercion are terminating because coinduction is only applied once a leading literal has been removed.

In case of proof step (Lit), we simply reduce injection/projection among regular types r and r' to injection/projection among their partial derivatives. An important fact is that the set of partial derivatives of a regular expression and its descendants is finite and the elements can be computed systematically. Thus, we can define the following injection and projection functions among partial derivatives:

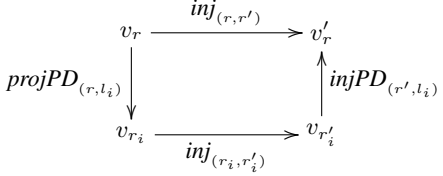
$$\begin{array}{lcl} \mathit{projPD}_{(r,l_i)} & :: & v_r \rightarrow \mathit{Maybe}\ (v_{l_i}, v_{r_i}) \\ \mathit{injPD}_{(r,l_i)} & :: & (v_{l_i}, v_{r_i}) \rightarrow v_r \end{array}$$

Function $\mathit{projPD}_{(r,l_i)}$ attempts to extract the partial derivatives component to literal l_i and function $\mathit{injPD}_{(r,l_i)}$ injects back the partial derivative component into its original expression.

In summary, our approach to derive the injection function *inj* is as follows:

1. Apply *projPD* to test which partial derivative component r_i can be extracted from r.
2. Apply *inj* on the extracted partial derivative components r_i which yields r_i' where either *inj* is a recursive call due to coinduction or a base case such as injecting an empty parse tree (representing the empty word) into another empty parse tree.
3. Apply *injPD* on r_i' to inject into original expression r'.

The following diagram summarizes our approach:



For example, consider the specific case $\vdash\ a + b \leq a + b$ where we encounter the following injection and projection functions. First, we give injection and projection among the resulting partial derivative components.

$$\begin{array}{l} \mathit{projPD}_{(a+b,a)}\ =\ \lambda\, v.\ \mathsf{case}\ v\ \mathsf{of} \\ \qquad\qquad\qquad\quad \mathit{Left}\ v\ \rightarrow\ (v, ()) \\ \qquad\qquad\qquad\quad \mathit{Right}\ v\ \rightarrow\ \mathit{Nothing} \end{array}$$

$$\begin{array}{l} \mathit{projPD}_{(a+b,b)}\ =\ \lambda\, v.\ \mathsf{case}\ v\ \mathsf{of} \\ \qquad\qquad\qquad\quad \mathit{Left}\ v\ \rightarrow\ \mathit{Nothing} \\ \qquad\qquad\qquad\quad \mathit{Right}\ v\ \rightarrow\ (v, ()) \end{array}$$

$$\begin{array}{l} \mathit{injPD}_{(a+b,a)}\ =\ \lambda\, (v, ()).\ \mathit{Left}\ v \\ \mathit{injPD}_{(a+b,b)}\ =\ \lambda\, (v, ()).\ \mathit{Right}\ v \end{array}$$

The containment proof for $\vdash\ a + b \leq a + b$ does not require recursion (coinduction) and simply reduces to the following base case.

$$\mathit{inj}_{(\epsilon,\epsilon)}\ =\ \lambda\, ().\ ()$$

Term $()$ is the proof term representation for ϵ.

The definition of *inj* for $\vdash\ a+b \leq a+b$ is then straightforward.

$$\begin{array}{l} \mathit{inj}_{(a+b,a+b)}\ = \\ \quad \lambda\, v.\ \mathsf{case}\ (\mathit{projPD}_{(a+b,a)}\ v,\ \mathit{projPD}_{(a+b,b)}\ v)\ \mathsf{of} \\ \qquad (\mathit{Just}\ (v, v'), \_)\ \rightarrow\ \mathit{injPD}_{(a+b,a)}\ (v,\ \mathit{inj}_{(\epsilon,\epsilon)}\ v') \\ \qquad (\mathit{Nothing},\ \mathit{Just}\ (v, v'))\ \rightarrow\ \mathit{injPD}_{(a+b,b)}\ (v,\ \mathit{inj}_{(\epsilon,\epsilon)}\ v') \end{array}$$

The case of projection works similarly and will be explained in detail later.

Disambiguation Matching may be ambiguous in the sense that there are several parse trees for the same word and regular expression. For example, consider word a and expression $a + a$ for which we find $\vdash\ \mathit{Left}\ a : a + a$ and $\vdash\ \mathit{Right}\ a : a + a$.

Similarly, we find that coercions connected to containment relations may be ambiguous. For example, expression a is contained in expression $a + a$ and there are several ways to inject a's parse tree into $a + a$. Namely, $\lambda v.\mathit{Left}\ a$ and $\lambda v.\mathit{Right}\ a$.

The standard approach to avoid ambiguities is to impose a disambiguation policy. Disambiguation is important and guarantees the deterministic and predictable use of regular expressions as types in a programming language setting [11]. Earlier work [9] on containment as coercions does not consider disambiguation.

Our approach based on partial derivatives for deriving coercions out of containment proofs naturally supports the greedy left-most disambiguation strategy [17] as found in Perl. We greedily take away leading literals and the normalized proof structure applies projection and injection on partial derivative components from left to right. For the above example, we compute the greedy left-most coercion $\lambda v.\mathit{Left}\ a$.

Parametric Coercions Our approach naturally supports parametric regular expressions and thus parametric coercions. Parametric variables such as α are simply treated like Skolem constants when computing partial derivatives.

For example, consider the provable statement $\vdash\ \alpha \leq \alpha + \beta$ where α and β are parametric variables. Our approach yields the following injection function $\lambda v.\mathit{Left}\ v$ via which we can inject any parse tree of α or any instance of it, into a parse tree of $\alpha + \beta$.

Faithful Instantiation In general, parametric coercions will not retain a specific disambiguation strategy, e.g. greedy left-most, under instantiation. For example, consider the coercion $\lambda v.v$ expressing the containment relation among $\alpha \leq \alpha$. Suppose, we instantiate α with (a^*, a^*). The greedy left-most coercion for the instantiated

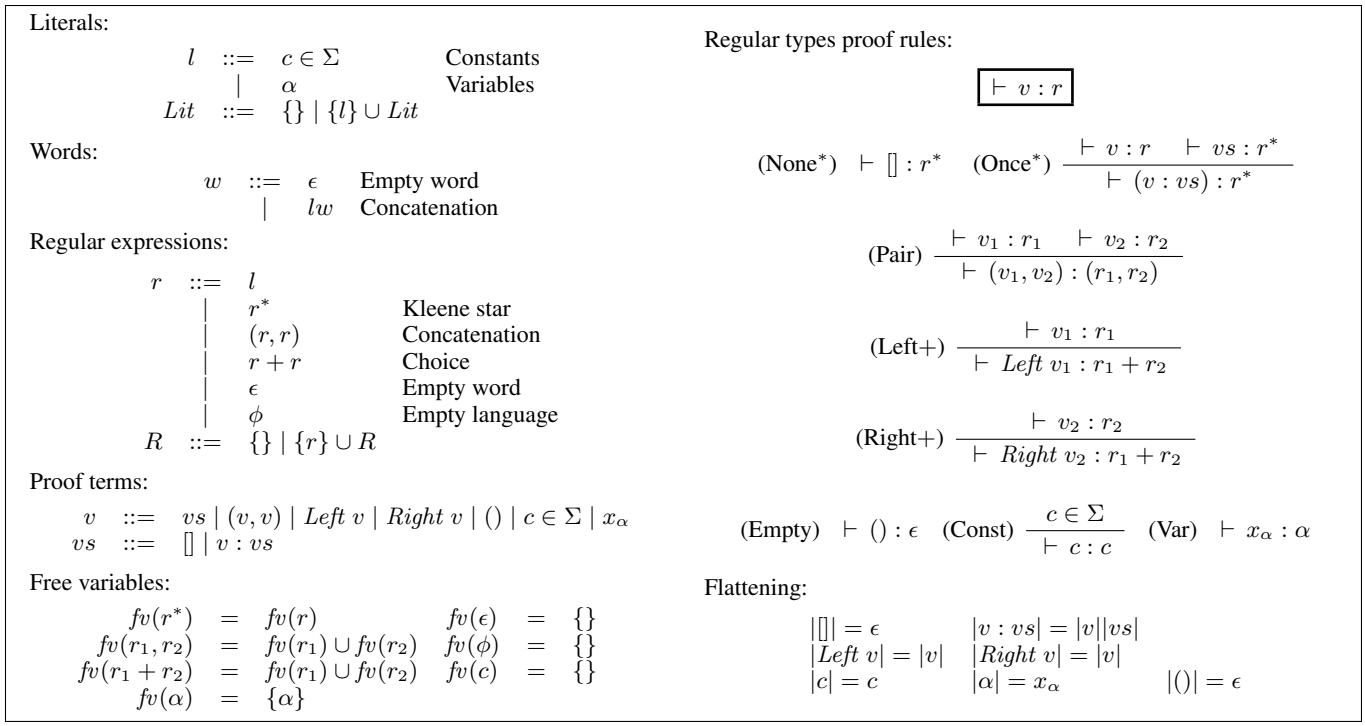


Figure 1: Regular Expressions and Proof Terms

case is $\lambda(v_1, v_2).(v_1 ++ v_2, [])$. Clearly, coercion $\lambda v.v$ does not satisfy the greedy left-most property for the instantiated case.

This issue has previously been discussed. The works in [10, 20] do not consider parametric containment coercions directly, rather, these works consider parametric programs operating on values of regular types. A program is only accepted if the program's meaning will not change under *all* instantiations. This is reminiscent of the approach to guarantee coherence [15] for type classes [8].

We propose a more relaxed 'ambiguity' check which takes into account the specific instantiation. After all, for *some* instantiation, a program's meaning may remain identical to the monomorphized program, but for others it may differ. Our idea is to check if for a specific instance ψ the parametric proof for $\vdash r \leq r'$ remains faithful w.r.t. the monomorphic proof $\vdash \psi(r) \leq \psi(r')$. We develop novel conditions which guarantee that instantiation is faithful.

Plan of Attack The up-coming section reviews some background material on regular expression types and Antimirov's partial derivative-based containment proof system. Then, in Section 4 we present our main technical contributions. Expressing containment as pairs of injection/projection function, disambiguation via greedy left-most coercions and faithful instantiation of parametric coercions. Section 5 briefly discusses our implementation.

3. Background

We first introduce some basic concepts about regular expressions as types.

3.1 Regular Expressions as Types

Figure 1 introduces proof terms v and regular expressions r. Proof terms represent parse trees which are represented via some standard data constructors such as lists, pairs, left/right injection into a disjoint sum etc. Constants c are taken from a finite alphabet Σ. Regular expressions may contain parametric variables α. For each variable we introduce a distinct proof term variable x_α.

We assume that $\mathcal{L}$ refers to the set of literals of the regular expressions used. Literals are either constants or variables. We will always consider only finitely many regular expressions. Hence, $\mathcal{L}$ is always finite.

Judgments $\vdash v : r$ relate proof terms and regular expressions. It is straightforward to see that $\vdash v : r$ is derivable if the word underlying v is contained in the language described by r. That is,

$$L(r) = \{\ |v| \ \mid \ \vdash v : r\ \}$$

where the flattening function $|\cdot|$ extracts the underlying word.

The following definitions will be required when discussing disambiguation of coercions.

DEFINITION 1 (Unambiguity). *We say that regular expression r is* unambiguous *iff for any v_1 and v_2 such that $|v_1| = |v_2|$, $\vdash v_1 : r$ and $\vdash v_2 : r$ we have that $v_1 = v_2$.*

DEFINITION 2 (Greedy Left-Most Parse Tree Ordering). *We define an ordering $v_1 >_r v_2$ among proof terms v_1 and v_2 where r is the underlying regular expression via the following rules:*

$$\frac{v_1 = v_1' \quad v_2 >_{r_2} v_2'}{(v_1, v_2) >_{(r_1, r_2)} (v_1', v_2')} \qquad \frac{v_1 >_{r_1} v_1'}{(v_1, v_2) >_{(r_1, r_2)} (v_1', v_2')}$$

$$Left\ v_1 >_{r_1 + r_2} Right\ v_2$$

$$\frac{v_2 >_{r_2} v_2'}{Right\ v_2 >_{r_1 + r_2} Right\ v_2'} \qquad \frac{v_1 >_{r_1} v_1'}{Left\ v_1 >_{r_1 + r_2} Left\ v_1'}$$

$$\frac{v_1 >_r v_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2} \qquad \frac{v_1 = v_2 \quad vs_1 >_{r^*} vs_2}{v_1 : vs_1 >_{r^*} v_2 : vs_2}$$

Let r be a regular expression and v_1 and v_2 proof terms such that $\vdash v_1 : r$ and $\vdash v_2 : r$. We define $v_1 \geq_r v_2$ iff either v_1 and

Containment proof environment:

$$\Gamma \quad ::= \quad \{\} \mid \{r_1 \leq r_2\} \cup \Gamma$$

Containment proof rules:

$$\text{(Lit1)} \quad \frac{\Gamma \vdash \mathsf{pd}(r_1, l) \leq \mathsf{pd}(r_2, l) \text{ for each } l \in \mathcal{L}}{\Gamma \vdash_{\text{lit}} r_1 \leq r_2}$$

$$\text{(Lit2)} \quad \frac{\begin{array}{c}\Gamma \cup \{r_1 \leq r_2\} \vdash_{\text{lit}} r_1 \leq r_2 \\ \text{if } \epsilon \in r_1 \text{ then } \epsilon \in r_2 \\ \neg(\phi \sim r_1) \quad \neg(\phi \sim r_2)\end{array}}{\Gamma \vdash r_1 \leq r_2}$$

$$\text{(Sub)} \quad \frac{\{\} \vdash r_1 \leq r_2 \quad \mathcal{L} = \Sigma \cup fv(r_1) \cup fv(r_2)}{\vdash r_1 \leq r_2}$$

$$\text{(Phi)} \quad \frac{\phi \sim r_1}{\Gamma \vdash r_1 \leq r_2} \qquad \text{(CoIn)} \quad \frac{(r_1 \leq r_2) \in \Gamma}{\Gamma \vdash r_1 \leq r_2}$$

Containment proof example: $a^* \leq (a^*, a^*)$

$$\begin{array}{lcl}\mathsf{pd}(a^*, a) & = & (\epsilon, a^*) \\ \mathsf{pd}((a^*, a^*), a) & = & ((\epsilon, a^*), a^*) + (\epsilon, a^*) \\ \mathsf{pd}((\epsilon, a^*), a) & = & (\epsilon, a^*) \\ \mathsf{pd}(((\epsilon, a^*), a^*) + (\epsilon, a^*), a) & = & ((\epsilon, a^*), a^*) + (\epsilon, a^*)\end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\epsilon, a^*) \leq ((\epsilon, a^*), a^*) + (\epsilon, a^*)}{\in} \text{(CoIn)}}{\{a^* \leq (a^*, a^*), (\epsilon, a^*) \leq ((\epsilon, a^*), a^*) + (\epsilon, a^*)\}}}{\vdash} \text{(Lit1)}}{(\epsilon, a^*) \leq ((\epsilon, a^*), a^*) + (\epsilon, a^*)}}{\{a^* \leq (a^*, a^*), (\epsilon, a^*) \leq ((\epsilon, a^*), a^*) + (\epsilon, a^*)\}}}{\vdash_{\text{lit}}} \text{(Lit2)}}{(\epsilon, a^*) \leq ((\epsilon, a^*), a^*) + (\epsilon, a^*)}}{\{a^* \leq (a^*, a^*)\} \vdash (\epsilon, a^*) \leq ((\epsilon, a^*), a^*) + (\epsilon, a^*)} \text{(Lit1)}$$

$$\frac{\{a^* \leq (a^*, a^*)\} \vdash_{\text{lit}} a^* \leq (a^*, a^*)}{\{\} \vdash a^* \leq (a^*, a^*)} \text{(Lit2)}$$

$$\frac{\{\} \vdash a^* \leq (a^*, a^*)}{\vdash a^* \leq (a^*, a^*)} \text{(Sub)}$$

Figure 2: Regular Expression Containment via Partial Derivatives

v_2 *are equal or* $v_1 >_r v_2$. *We say that* v_1 *is the* greedy left-most *proof term w.r.t.* r *iff* $\vdash v_1 : r$ *and* $v_1 \geq_r v_2$ *for any proof term* v_2 *where* $\vdash v_2 : r$ *and* $|v_1| = |v_2|$.

3.2 Containment via Partial Derivatives

We introduce the essential details of the containment algorithm based on partial derivatives [1, 2]. First, we consider the effective computation of partial derivatives.

DEFINITION 3 (Partial Derivatives). *Let* r *be a regular expression and* l *be a literal. The definition of the set* $\mathcal{PD}(r, l)$ *of partial derivatives of* r *w.r.t.* l *is as follows:*

$$\begin{array}{l}\mathcal{PD}(\cdot,\cdot) :: r \rightarrow l \rightarrow R \\ \mathcal{PD}(r^*, l) = \{ (r', r^*) \mid r' \in \mathcal{PD}(r, l) \} \\ \mathcal{PD}((r_1, r_2), l) \\ \quad \mid \epsilon \in r_1 = \{ (r', r_2) \mid r' \in \mathcal{PD}(r_1, l) \} \cup \mathcal{PD}(r_2, l) \\ \quad \mid \mathsf{otherwise} = \{ (r', r_2) \mid r' \in \mathcal{PD}(r_1, l) \} \\ \mathcal{PD}(r_1 + r_2, l) = \mathcal{PD}(r_1, l) \cup \mathcal{PD}(r_2, l) \\ \mathcal{PD}(l', l) \\ \quad \mid l' == l = \{ \epsilon \} \\ \quad \mid \mathsf{otherwise} = \{ \} \\ \mathcal{PD}(\epsilon, l) = \{ \} \\ \mathcal{PD}(\phi, l) = \{ \}\end{array}$$

The above definition uses the helper function $\epsilon \in r$ to test if ϵ is contained in the language denoted by r. The definition of $\epsilon \in r$ is as follows:

$$\begin{array}{l}\epsilon \in \cdot :: r \rightarrow \mathit{Bool} \\ \epsilon \in r^* = \mathit{True} \\ \epsilon \in (r_1, r_2) \; \epsilon \in r_1 \wedge \epsilon \in r_2 \\ \epsilon \in (r_1 + r_2) = \epsilon \in r_1 \vee \epsilon \in r_2 \\ \epsilon \in l = \mathit{False} \\ \epsilon \in \epsilon = \mathit{True} \\ \epsilon \in \phi = \mathit{False}\end{array}$$

THEOREM 3.1 (Finite Set of Partial Derivatives [2]). *Let* r *be a regular expression and* $\mathcal{L}$ *be a set containing all literals in* r. *Then, we define* $\mathcal{PD}_{cl}(r, \mathcal{L})$, *the set of partial derivatives of a regular expression and its descendants, as follows:* $\mathcal{PD}_{cl}(r, \mathcal{L}) = \mathsf{fix}(\{r\})$ *where*

$$\begin{array}{l}\mathsf{fix}(R) = \\ \quad \mathsf{let}\ R' = \{ r' \mid \exists\, r \in R \wedge l \in \mathcal{L}.\, r' \in \mathcal{PD}(r, l) \} \\ \quad \mathsf{in\ if}\ R' \subseteq R\ \mathsf{then}\ R \\ \quad\quad \mathsf{else}\ \mathsf{fix}(R \cup R')\end{array}$$

The size of $\mathcal{PD}_{cl}(r, \mathcal{L})$ *is linear in the size of* r.

The partial derivative operation $\mathcal{PD}(r, l)$ effectively removes any leading l from r. In case of concatenation, we therefore remove l from both expressions in case the leading expression contains the empty string.

The approach sounds very similar to the derivative operation by Brzozowski [3]. The difference is that $\mathcal{PD}$ returns a set of expressions, e.g. $\mathcal{PD}(r, l) = \{r_1, ..., r_m\}$. Building the alternatives among these expressions corresponds exactly to the derivative operation. That is, $L(r_1 + ... + r_m) = \{w \mid lw \in L(r)\}$.

For example, consider the construction of the partial derivatives of (a^*, a^*) wrt. a:

$$\begin{array}{lcl}\mathcal{PD}((a^*, a^*), a) & = & \underbrace{\{((\epsilon, a^*), a^*)\}}_{\{(r', a^*) \mid r' \in \mathcal{PD}(a^*, a)\}} \cup \underbrace{\{(\epsilon, a^*)}_{\mathcal{PD}(a^*, a)} \\ \mathcal{PD}(a^*, a) & = & \underbrace{\{(\epsilon, a^*)\}}_{\{(r', a^*) \mid r' \in \mathcal{PD}(a, a)\}} \\ \mathcal{PD}(a, a) & = & \{\epsilon\}\end{array}$$

The important property of partial derivatives is that the set of partial derivatives of a regular expression and its descendants is finite. For example, $\mathcal{PD}((\epsilon, a^*), a) = \{(\epsilon, a^*)\}$.

Figure 2 contains the partial derivative-based proof system to decide containment among regular expressions. We also include a worked out example for proving $\vdash a^* \leq (a^*, a^*)$.

The proof rules make use of the following helper operations. Via operation $\mathsf{pd}(\cdot,\cdot)$ we turn the set of partial derivatives into a regular expression.

$$\begin{array}{l}\mathsf{pd}(\cdot,\cdot) :: r \rightarrow l \rightarrow r \\ \mathsf{pd}(r, l) = \mathcal{PD}(r, l) \downarrow_r \\ \\ \cdot \downarrow_r :: R \rightarrow r \\ \{\} \downarrow_r = \phi\end{array}$$

$\{r\}\downarrow_r = r$
$\{r_1, ..., r_n\}\downarrow_r = r_1 + ... + r_n$

where we assume that $+$ is right associative.

Operation $\phi \sim \cdot$ checks if a regular expression equals the empty language.

$\phi \sim \cdot :: r \rightarrow Bool$
$\phi \sim r^* = False$
$\phi \sim (r_1, r_2) = \phi \sim r_1 \vee \phi \sim r_2$
$\phi \sim (r_1 + r_2) = \phi \sim r_1 \wedge \phi \sim r_2$
$\phi \sim l = False$
$\phi \sim \epsilon = False$
$\phi \sim \phi = True$

Let us consider the derivation tree for proof for $a^* \leq (a^*, a^*)$ in Figure 2. In the first step (from the bottom), we initialize the containment proof environment. See rule (Sub). Next, we move the to be proven statement to the environment. See rule (Lit2) where the side conditions $\neg(\phi \sim r_1)$ and $\neg(\phi \sim r_2)$ guarantee that a proof is aborted for invalid containment statements such as $a^* \leq \phi$. Then, we apply (Lit1) and 'reduce' the containment problem to a containment problem among the partial derivatives. The sequence of applications of (Lit2) and (Lit1) is repeated one more time until we can finish the proof via the coinduction rule.

Rule (Phi) becomes necessary for cases such as $a^* \leq (a + b)^*$. In some subproof, we encounter $\phi \leq ((\epsilon, b), (a + b)^*)$ because of $\mathsf{pd}(a^*, b) = \phi$ and $\mathsf{pd}((a + b)^*, b) = ((\epsilon, b), (a + b)^*)$.

We conclude this section by highlighting a couple of subtle points which will show up in the up-coming section where we show who to build injection/projection functions based on partial derivatives.

The first point is that we slightly deviate from the original definition of partial derivatives in [2]. The original definition yields the set $\{a^*\}$ for a^* and a whereas our Definition 3 yields $\{(\epsilon, a^*)\}$. Our definition still enjoys all the good properties (see Theorem 3.1). We may generate slightly larger proof derivations as can been seen for the worked out example $\vdash a^* \leq a^* + a^*$. The important advantage of our definition is that the specification the injection/projection functions among partial derivatives becomes easier.

The second point is that the order of partial derivatives is important. That is, $\mathcal{PD}((a^*, a^*), a) = \{((\epsilon, a^*), a^*), (\epsilon, a^*)\}$ rather than $\mathcal{PD}((a^*, a^*), a) = \{(\epsilon, a^*), ((\epsilon, a^*), a^*)\}$. The exact reason will be explained when establishing the greedy left-most property for coercions in the up-coming section.

4. Containment as Injection and Projection

We discuss the construction of injection/projection coercions expressing containment proofs among regular expressions. For regular expressions r_1 and r_2 satisfying $\vdash r_1 \leq r_2$, we obtain these functions by calling $\mathit{inj}_{(\mathcal{L}, r_1, r_2)}$ and $\mathit{proj}_{(\mathcal{L}, r_1, r_2)}$ in Figure 3. Recall that $\mathcal{L}$ refers to the set of literals contained in r_1 and r_2. As discussed earlier, injection/projection among regular types is reduced to injection and projection among partial derivatives. See Figures 4 and 5.

First, we motivate our notation for the specification of injection/projection as type-indexed function families (Section 4.1), followed by a review of a few examples explaining how injection/projection works (Section 4.2). Then, we state some essential correctness and complexity results (Section 4.3). Next, we verify that our approach satisfies the greedy left-most disambiguation strategy (Section 4.4). Finally, we discuss faithful instantiation (Section 4.5).

4.1 Families of Type-indexed Functions with Contracts

Injection/projection functions are specified as families of type-indexed functions. Some readers might have expected a specification where proofs/coercions are attached to containment proof rules and their judgments. The reason for the given formulation is that the shape of containment proofs is completely fixed by the set of partial derivatives. Hence, we can give generic definitions of injection/projection. The definitions are of course tightly coupled to the proof structure of the partial derivative containment proof system. This will be discussed in the following subsection.

The point we make here is as follows. Index parameters attached to injection/projection functions are not dependent on proof terms. They play the roles of static types. As we will show, for concrete index parameters we obtain total functions operating at the level of proof terms. In these functions, all calls of *inj*/*proj* and its specialized versions, as well as all guarded clauses of the form $\mid guard = body$, have been resolved.

Concretely, the resulting functions can expressed as simply-typed lambda calculus expressions extended with structured data and recursion:

$e \quad ::= \quad v \mid p \mid f \mid \lambda v.e \mid e\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \overline{[p \rightarrow e]} \mid \mathsf{rec}\ f.e$
$p \quad ::= \quad x \mid (p, p) \mid \mathit{Left}\ p \mid \mathit{Right}\ p \mid \mathit{Just}\ v \mid \mathit{Nothing}$

We omit if-then-else which can be expressed via case expressions as well as recursive let definitions which can be expressed via rec.

Not all index parameters are sensible and result in some expression e. Therefore, injection/projection functions are specified as families of type-indexed functions *with contracts*. Contracts are formulated as type signatures of the form $C \Rightarrow t$ and state assumptions about inputs as well as guarantee about outputs.

For example, consider $\mathit{injE}_{(r', r, l)}$ in Figure 4. Suppose r' is an element in the set of partial derivatives of the regular expression r w.r.t. literal l, then $\mathit{injE}_{(r', r, l)}$ yields a function mapping a pair of proof terms to another proof term.

In case of $\mathit{injE}_{(r', (r_1, r_2), l)}$, we apply typeCase to analysis the shape of r_1. Based on the earlier failed guard $r' \in \mathcal{PD}(r_2, l) \wedge \epsilon \in r_1$ and the shape of partial derivatives, we can guarantee that the typeCase will not fail.

Function *injE* as well as the helper function $mkEps$ is called recursively. However, the size of the index arguments strictly increase and the contracts imposed on the index parameters are always satisfied.

To conclude, $\mathit{injE}_{(r', r, l)}$ yields a function of the form e. In fact, as stated by its type signature, we can give a more precise characterization: For each input pair of a literal value v_l and proof term $v_{r'}$ corresponding to a partial derivative, *injE* yields a proof term v_r corresponding to the original regular expression. Similar observations apply to all other injection/projection functions.

LEMMA 4.1. *Functions* $\mathit{inj}_{(Lit, r_1, r_2)}$, $\mathit{proj}_{Lit, r_1, r_2)}$, $\mathit{injE}_{(r', r, l)}$, $mkEps_r$, $\mathit{projPD}_{(R, r, l)}$, $\mathit{projE}_{(r', r, l)}$ *and* $isEps_r$ *yield lambda expressions* e *assuming that the contracts imposed on their index parameters are satisfied.*

The above result is straightforward. The only case that deserves a closer observation is *inj* for the following case:

$\mathit{inj}_{(\{l\} \cup Lit, r_1, r_2)} \mid \phi \sim r_1 = \bot$

The above deals with the base case (Phi) of the containment proof system in Figure 2. Clearly, there exists no proof value for ϕ. Hence, this case will never be executed at run-time.

Another special case is $\mathit{injPD}_{(R, r, l)}$ for

$\mathit{injPD}_{(\{\}, r, l)} = \bot$

$$
\begin{array}{l}
\mathit{inj}_{(\mathit{Lit},r_1,r_2)} :: (\mathit{Lit} \subseteq \mathcal{L} \wedge \vdash r_1 \leq r_2 \wedge \vdash v_1 : r_1 \wedge \vdash v_2 : r_2) \Rightarrow v_1 \rightarrow v_2 \\
\mathit{inj}_{(\{l\} \cup \mathit{Lit},r_1,r_2)} \quad \mid \phi \sim r_1 = \bot \\
\qquad\qquad\qquad\qquad \mid \mathsf{otherwise} = \lambda\, v. \quad \mathsf{if}\ \mathit{isEps}_{r_1}\ v\ \mathsf{then}\ \mathit{mkEps}_{r_2} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else\ case}\ (\mathit{projPD}_{(\mathcal{PD}(r_1,l),r_1,l)}\ v)\ \mathsf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ (v_l,\ v) \rightarrow \mathit{injPD}_{(\mathcal{PD}(r_2,l),r_2,l)}\ (v_l,\ \mathit{inj}_{(\mathcal{L},\mathsf{pd}(r_1,l),\,\mathsf{pd}(r_2,l))}\ v) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{inj}_{(\mathit{Lit},r_1,r_2)}\ v \\
\mathit{proj}_{\mathit{Lit},r_1,r_2)} :: (\mathit{Lit} \subseteq \mathcal{L} \wedge \vdash r_1 \leq r_2 \wedge \vdash v_1 : r_1 \wedge \vdash v_2 : r_2) \Rightarrow v_2 \rightarrow \mathit{Maybe}\ v_1 \\
\mathit{proj}_{(\{l\} \cup \mathit{Lit},r_1,r_2)} \\
\quad \mid \phi \sim r_1 = \lambda\, v.\ \mathit{Nothing} \\
\quad \mid \epsilon \in r_1 = \lambda\, v.\ \mathsf{if}\ \mathit{isEps}_{r_2}\ v\ \mathsf{then}\ \mathit{Just}\ \mathit{mkEps}_{r_1} \\
\qquad\qquad\qquad \mathsf{else\ case}\ (\mathit{projPD}_{(\mathcal{PD}(r_2,l),r_2,l)}\ v)\ \mathsf{of} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ (v_l, v) \rightarrow \mathsf{case}\ (\mathit{proj}_{(\mathcal{L},\mathsf{pd}(r_1,l),\mathsf{pd}(r_2,l))}\ v)\ \mathsf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ v' \rightarrow \mathit{Just}\ (\mathit{injPD}_{(\mathcal{PD}(r_1,l),r_1,l)}\ (v_l, v')) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{proj}_{(\mathit{Lit},r_1,r_2)} \\
\quad \mid \mathsf{otherwise} = \lambda\, v.\ \mathsf{if}\ \mathit{isEps}_{r_2}\ v\ \mathsf{then}\ \mathit{Nothing} \\
\qquad\qquad\qquad\quad \mathsf{else\ case}\ (\mathit{projPD}_{(\mathcal{PD}(r_2,l),r_2,l)}\ v)\ \mathsf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{Just}\ (v_l, v) \rightarrow \mathsf{case}\ (\mathit{proj}_{(\mathcal{L},\mathsf{pd}(r_1,l),\mathsf{pd}(r_2,l))}\ v)\ \mathsf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ v' \rightarrow \mathit{Just}\ (\mathit{injPD}_{(\mathcal{PD}(r_1,l),r_1,l)}\ (v_l, v')) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{proj}_{(\mathit{Lit},r_1,r_2)}
\end{array}
$$

Figure 3: Injection and projection among regular types

Injection of sum of partial derivatives:

$$
\begin{array}{l}
\mathit{injPD}_{(R,r,l)} :: (R \subseteq \mathcal{PD}(r,l) \wedge \vdash v_l : l \wedge \\
\qquad\qquad \vdash v : \mathsf{pd}(r,l) \wedge \vdash v_r : r) \Rightarrow (v_l, v) \rightarrow v_r \\
\mathit{injPD}_{(\{\},r,l)} = \bot \\
\mathit{injPD}_{(\{r'\},r,l)} = \lambda\, (v_l, v).\ \mathit{injE}_{(r',r,l)}\ (v_l, v) \\
\mathit{injPD}_{(\{r_1,\ldots,r_n\},r,l)} = \lambda\, (v_l, v). \\
\quad \mathsf{case}\ v\ \mathsf{of} \\
\qquad \mathit{Left}\ v \rightarrow \mathit{injE}_{(r_1,r,l)}\ (v_l, v) \\
\qquad \mathit{Right}\ v \rightarrow \mathit{injPD}_{(\{r_2,\ldots,r_n\},r,l)}\ (v_l, v)
\end{array}
$$

$$
\begin{array}{l}
\mathit{mkEps}_r :: \epsilon \in r \Rightarrow v \\
\mathit{mkEps}_{r^*} = [] \\
\mathit{mkEps}_{(r_1,r_2)} = (\mathit{mkEps}_{r_1},\ \mathit{mkEps}_{r_2}) \\
\mathit{mkEps}_{(r_1+r_2)} \\
\quad \mid \epsilon \in r_1 = \mathit{Left}\ (\mathit{mkEps}_{r_1}) \\
\quad \mid \epsilon \in r_2 = \mathit{Right}\ (\mathit{mkEps}_{r_2}) \\
\mathit{mkEps}_{\epsilon} = ()
\end{array}
$$

Injection of partial derivative:

$$
\begin{array}{l}
\mathit{injE}_{(r',r,l)} :: (r' \in \mathcal{PD}(r,l) \wedge \vdash v_l : l \wedge \\
\qquad\qquad \vdash v_{r'} : r' \wedge \vdash v_r : r) \Rightarrow (v_l, v_{r'}) \rightarrow v_r \\
\mathit{injE}_{((r',r''),r^*,l)} = \lambda\, (v_l, (v_1, vs)).\ (\mathit{injE}_{(r',r,l)}\ (v_l, v_1)) : vs \\
\mathit{injE}_{(r',(r_1,r_2),l)} \\
\quad \mid r' \in \mathcal{PD}(r_2,l) \wedge \epsilon \in r_1 = \\
\qquad \lambda\, (v_l, v').\ (\mathit{mkEps}_{r_1},\ \mathit{injE}_{(r',r_2,l)}\ (v_l, v')) \\
\quad \mid \mathsf{otherwise} = \\
\qquad \mathsf{typeCase}\ r'\ \mathsf{of} \\
\qquad\quad (r_1', r_2') \rightarrow \lambda\, (v_l, (v_1, v_2)).\ (\mathit{injE}_{(r_1',r_1,l)}\ (v_l, v_1),\ v_2) \\
\mathit{injE}_{(r',(r_1+r_2),l)} \\
\quad \mid r' \in \mathcal{PD}(r_1,l) = \lambda\, (v_l, v').\ \mathit{Left}\ (\mathit{injE}_{(r',r_1,l)}\ (v_l, v')) \\
\quad \mid r' \in \mathcal{PD}(r_2,l) = \lambda\, (v_l, v').\ \mathit{Right}\ (\mathit{injE}_{(r',r_2,l)}\ (v_l, v')) \\
\mathit{injE}_{(\epsilon,l,l)} = \lambda\, (v_l, ()).\ v_l
\end{array}
$$

Figure 4: Injection of partial derivatives

The above case arises in case of for containment proofs such as $\vdash \phi \leq r$. The associated injection function is represented by $\mathit{inj}_{(\mathcal{L},\phi,r)}$. In the body of $\mathit{inj}_{(\mathcal{L},\phi,r)}$, we find a call $\mathit{injPD}_{(\{\},\_,\_)}$. Of course, this call will never be executed but needs to be resolved.

LEMMA 4.2. *Function $\mathit{injPD}_{(R,r,l)}$ yields a lambda expression e for any set R of partial derivatives of r w.r.t. l where all calls to injPD, mkEps and injE have been resolved. Assuming that R is non-empty, e is total*

Another observation is that during the resolution of calls $\mathit{inj}_{(\mathit{Lit},r_1,r_2)}$ we may encounter the same call again. This is due to the co-inductive nature of the regular expression containment proof system. In this case, we simply generate a recursive lambda expression.

We summarize the results.

THEOREM 4.3 (Coercions are Well-Typed Lambda Expressions). *Let r_1 and r_2 be two regular expressions such that $\vdash r_1 \leq r_2$. Then, we obtain lambda expressions i and p representing total injection and projection functions.*

Next, we take a closer look at the definitions of injection and projection.

4.2 Projection and Injection via Shuffling of Literals

Our goal is to understand the inner-workings of the injection and projection functions. The gist of the idea in case of *inj* is as follows:

1. *projPD* extracts a literal
2. *inj* is applied on the remaining part
3. *injPD* puts back the literal on the resulting part

Projection onto sum of partial derivatives:

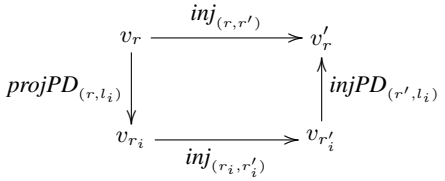
$$
\begin{array}{l}
\mathit{projPD}_{(R,r,l)} :: (R \subseteq \mathcal{PD}(r,l) \wedge \vdash v : r \wedge \vdash v' : \mathsf{pd}(r,l) \wedge \\
\qquad \vdash v_l : l) \Rightarrow v \rightarrow \mathit{Maybe}\,(v_l, v') \\
\mathit{projPD}_{(\{\},r,l)} = \lambda\, v.\, \mathit{Nothing} \\
\mathit{projPD}_{(\{r'\},r,l)} = \lambda\, v.\, \mathit{projE}_{(r',r,l)}\ v \\
\mathit{projPD}_{(\{r_1,\ldots,r_n\},r,l)} = \lambda\, v.\ \mathsf{case}\ (\mathit{projE}_{(r_1,r,l)}\ v)\ \mathsf{of} \\
\qquad \mathit{Just}\,(v_l, v') \rightarrow \mathit{Just}\,(v_l,\ \mathit{Left}\ v') \\
\qquad \mathit{Nothing} \rightarrow \mathsf{case}\ (\mathit{projPD}_{(\{r_2,\ldots,r_n\},r,l)}\ v)\ \mathsf{of} \\
\qquad\qquad \mathit{Just}\,(v_l, v') \rightarrow \mathit{Just}\,(v_l,\ \mathit{Right}\ v') \\
\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\mathit{isEps}_r :: (\vdash v : r) \Rightarrow v \rightarrow \mathit{Bool} \\
\mathit{isEps}_{r^*}\ [] = \mathit{True} \\
\mathit{isEps}_{r^*}\ (v : vs) = \mathit{False} \\
\mathit{isEps}_{(r_1,r_2)}\ (v_1, v_2) = \\
\quad (\mathit{isEps}_{r_1}\ v_1)\ \&\&\ (\mathit{isEps}_{r_2}\ v_2) \\
\mathit{isEps}_{r_1+r_2}\ (\mathit{Left}\ v) = \mathit{isEps}_{r_1}\ v \\
\mathit{isEps}_{r_1+r_2}\ (\mathit{Right}\ v) = \mathit{isEps}_{r_2}\ v \\
\mathit{isEps}_{l\ \_} = \mathit{False} \\
\mathit{isEps}_{\epsilon}\ () = \mathit{True}
\end{array}
$$

Projection onto partial derivative:

$$
\begin{array}{l}
\mathit{projE}_{(r',r,l)} :: (r' \in \mathcal{PD}(r,l) \wedge \vdash v_l : l \wedge \\
\qquad \vdash v_{r'} : r' \wedge \vdash v_r : r) \\
\qquad \Rightarrow v_r \rightarrow \mathit{Maybe}\,(v_l, v') \\
\mathit{projE}_{((r',r''),r^*,\,l)} = \\
\quad \lambda.v\ \mathsf{case}\ v\ \mathsf{of} \\
\qquad [] \rightarrow \mathit{Nothing} \\
\qquad (v : vs) \rightarrow \mathsf{case}\ (\mathit{projE}_{(r',r,l)}\ v)\ \mathsf{of} \\
\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad \mathit{Just}\,(v_l, v') \rightarrow \mathit{Just}\,(v_l, (v', vs)) \\
\mathit{projE}_{(r',(r_1,r_2),l)} \\
\quad \mid r' \in \mathcal{PD}(r_2, l) \wedge \epsilon \in r_1 = \\
\qquad \lambda\,(v_1, v_2).\ \mathsf{if}\ \mathit{isEps}_{r_1}\ v_1 \\
\qquad\qquad \mathsf{then}\ \mathit{projE}_{(r',r_2,\,l)}\ v_2 \\
\qquad\qquad \mathsf{else}\ \mathit{Nothing} \\
\quad \mid \mathsf{otherwise} = \\
\qquad \mathsf{typeCase}\ r'\ \mathsf{of} \\
\qquad\quad (r_1', r_2') \rightarrow \lambda\,(v_1, v_2). \\
\qquad\qquad\quad \mathsf{case}\ (\mathit{projE}_{(r_1',r_1,l)}\ v_1)\ \mathsf{of} \\
\qquad\qquad\quad \mathit{Just}\,(v_l, v_1') \rightarrow \mathit{Just}\,(v_l, (v_1', v_2)) \\
\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\mathit{projE}_{(r',(r_1+r_2),l)} \\
\quad \mid r' \in \mathcal{PD}(r_1, l) \wedge r' \in \mathcal{PD}(r_2, l) = \\
\qquad \lambda\, v.\ \mathsf{case}\ v\ \mathsf{of}\ \mathit{Left}\ v \rightarrow \mathit{projE}_{(r',r_1,l)}\ v \\
\qquad\qquad\qquad\qquad \mathit{Right}\ v \rightarrow \mathit{projE}_{(r',r_2,l)}\ v \\
\quad \mid r' \in \mathcal{PD}(r_1, l) = \\
\qquad \lambda\, v.\ \mathsf{case}\ v\ \mathsf{of}\ \mathit{Left}\ v \rightarrow \mathit{projE}_{(r',r_1,l)}\ v \\
\qquad\qquad\qquad\qquad \mathit{Right}\ v \rightarrow \mathit{Nothing} \\
\quad \mid r' \in \mathcal{PD}(r_2, l) = \\
\qquad \lambda\, v.\ \mathsf{case}\ v\ \mathsf{of}\ \mathit{Left}\ v \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Right}\ v \rightarrow \mathit{projE}_{(r',r_2,l)}\ v \\
\mathit{projE}_{(\epsilon,l,l)} = \lambda\, v_l.\ \mathit{Just}\,(v_l, ())
\end{array}
$$

Figure 5: Projection onto partial derivatives

Recall the diagram from the earlier Section 2:



Like before in Section 2, we omit the $\mathcal{PD}$ parameter for *injPD* and *projPD* and write the shorter form $\mathit{projPD}_{(r,l)}$ instead of $\mathit{projPD}_{(\mathcal{PD}(r,l),r,l)}$. For simplicity, we will even write just *projPD*. The index parameters are always defined by the context.

The above process repeats itself in case of the second call of *inj*. Hence, effectively we perform the following sequence of operations:

1. Apply a sequence of *projPD* calls to extract *all* literals
2. Generate an empty proof term.
3. Apply a sequence of *injPD* calls to put back *all* literals

Function *proj* differs from *inj* in that if some literal extraction fails or an empty proof term can not be generated, the *Nothing* value is returned.

Let's consider the concrete example $\vdash (a^*, a^*) \leq (a^*, a^*)$ This example is a slight generalization of the example in Figure 2. The condensed proof derivation tree is as follows:

$$\dfrac{\dfrac{\vdash ((\epsilon, a^*), a^*) + (\epsilon, a^*) \leq ((\epsilon, a^*), a^*) + (\epsilon, a^*)}{\vdash ((a^*, a^*), a^*) + (\epsilon, a^*) \leq ((a^*, a^*), a^*) + (\epsilon, a^*)}}{\vdash (a^*, a^*) \leq (a^*, a^*)}$$

Let's consider the workings of the injection function applied to some concrete proof term $([], [aa])$. We attach proof terms to regular expressions in intermediate steps. We leave out "right-hand sides" because for the moment we are only interested in the sequence of *projPD* calls until we hit a base case.

$$\dfrac{\dfrac{\vdash \mathit{Right}\ ((), []) : ((\epsilon, a^*), a^*) + (\epsilon, a^*) \leq \ldots}{\vdash \mathit{Right}\ ((), [a]) : ((\epsilon, a^*), a^*) + (\epsilon, a^*) \leq \ldots}}{\vdash ([], [aa]) : (a^*, a^*) \leq \ldots}$$

After extracting literal a twice from $([], [aa])$ via *projPD*, we reach $\mathit{Right}\ ((), [])$. That is,

$$
\begin{array}{lcl}
\mathit{projPD}\ ([], [aa]) & \Longrightarrow & (a, \mathit{Right}\ ((), [a])) \\
\mathit{projPD}\ \mathit{Right}\ ((), [a]) & \Longrightarrow & (a, \mathit{Right}\ ((), []))
\end{array}
$$

At this point of the proof, one of the base cases of *inj* applies. See Figure 3 where we reach the case that the given proof value represents the empty string. Instantiated for our example, we find

$$\mathit{if}\ \mathit{isEps}_{r_2}\ (\mathit{Right}\ ((), []))\ \mathit{then}\ \mathit{mkEps}_{r_2}$$

where $r_2 = ((\epsilon, a^*), a^*) + (\epsilon, a^*)$. Condition $\mathit{isEps}_{r_2}\ (\mathit{Right}\ ((), []))$ clearly holds. The call mkEps_{r_2} computes an empty proof term which is here $\mathit{Left}\ (((), []), [])$.

What remains is to apply the sequence of *injPD* calls to put back the extracted literals. That is,

$$\begin{array}{lcl} \mathit{injPD}\ (a, \mathit{Left}\ (((), []), [])) & \Longrightarrow & \mathit{Left}\ (((), [a]), []) \\ \mathit{injPD}\ (a, \mathit{Left}\ (((), [a]), [])) & \Longrightarrow & ([a, a], []) \end{array}$$

This results in the following proof steps

$$\dfrac{\dfrac{\vdash \ldots \leq \mathit{Left}\ (((), []), []) : ((\epsilon, a^*), a^*) + (\epsilon, a^*)}{\vdash \ldots \leq \mathit{Left}\ (((), [a]), []) : ((\epsilon, a^*), a^*) + (\epsilon, a^*)}}{\vdash \ldots \leq ([a, a], []) : (a^*, a^*)}$$

As we can see, the *inj* function for $\vdash (a^*, a^*) \leq (a^*, a^*)$ applied to $([], [aa])$ yields $([a, a], [])$.

4.3 Correctness and Complexity

Based on the intuitions gained, we formally verify correctness of injection and projection as well as establishing some time complexity results. As observed, everything boils down to applications of *injPD*, *projPD* and $mkEps$. Hence, we require some essential correctness results for these functions.

Correctness for $mkEps$ is straightforward.

LEMMA 4.4 (Empty Word Proof Term). *Let r be a regular expression such that $\epsilon \in r$. Then, $\vdash mkEps_r : r$ and $|mkEps_r| = \epsilon$.*

Next, we consider *injPD*. Observe that *injPD* applies the appropriate helper *injE* by traversing over v. Hence, we first consider the correctness result for *injE* which verifies that *injE*'s contract specification can be satisfied and no literals during the injection are lost.

LEMMA 4.5 (*injE* Correctness). *Let r', r be regular expressions, l a literal, v_l, v' proof terms such that $r' \in \mathcal{PD}(r, l)$, $\vdash v_l : l$ and $\vdash v' : r'$. Then, $\mathit{injE}_{(r',r,l)}\ (v_l, v')$ yields some proof v such that $\vdash v : r$ and $|v| = v_l|v'|$.*

PROOF. (Sketch) We perform induction over r.

For example, consider case r^*. Elements in $\mathcal{PD}(r^*, l)$ are always concatenations of expressions: $\mathcal{PD}(r^*, l) = \{(r', r^*) | r' \in \mathcal{PD}(r, l)\}$. Hence, $v' = (v'', v''')$. [1] Thus, we can directly apply *injE* again on the first component v''. By induction we obtain the desired result and can thus establish the induction step.

As another case, consider $r_1 + r_2$. Recall the definition of $\mathcal{PD}$:

$$\begin{array}{l} \mathcal{PD}((r_1,\ r_2), l) \\ \quad \mid \epsilon \in r_1 = \{\ (r', r_2) \mid r' \in \mathcal{PD}(r_1, l)\ \} \cup \mathcal{PD}(r_2, l) \\ \quad \mid \mathsf{otherwise} = \{\ (r', r_2) \mid r' \in \mathcal{PD}(r_1, l)\ \} \end{array}$$

If $r' \in \mathcal{PD}(r_2, l)$ and $\epsilon \in r_1$ we definitely know that proof term v' is missing the first component. By induction, the call $\mathit{injE}_{(r',r_2,l)}(v_l, v')$ yields v'' such that $\vdash v'' : r_2$ and $|v''| = v_l|v'|$. We set $v = (mkEps_{r_1}, v'')$ and can thus establish the induction step.

The other cases can be proven similarly. □

We state correctness of *injPD* which follows straightforwardly from *injE*.

LEMMA 4.6 (*injPD* Correctness). *Let r be a regular expression, l a literal, v_l, v proof terms such that $\vdash v_l : l$ and $\vdash v : \mathsf{pd}(r, l)$. Then, $\mathit{injPD}_{(\mathcal{PD}(r,l),r,l)}\ (v_l, v)$ yields some proof v' such that $\vdash v' : r$ and $|v'| = v_l|v|$.*

We consider *projPD*. Observe that *projPD* iterates over the set of partial derivatives applying the helper *projE* until we either reach a successful case, or all cases yield $Nothing$. Hence, we first consider *projE*.

[1] Recall the "first point" at the end of Section 3.2.

LEMMA 4.7 (*projE* Correctness). *Let r be a regular expression, l a literal, v_l, v proof terms and w a word such that $\vdash v_l : l$, $\vdash v : r$ and $|v| = v_l w$. Then, there exists $r' \in \mathcal{PD}(r, l)$ such that $\mathit{projE}_{(r',r,l)}\ v$ yields $\mathit{Just}\ (v'_l, v')$ for some v'_l, v' where $\vdash v'_l : l$, $\vdash v' : r'$ and $|v'| = w$.*

PROOF. (Sketch)

We perform induction over r. For example, consider the case $r_1 + r_2$. That is, $\vdash v : r_1 + r_2$. Suppose $v = \mathit{Left}\ v_1$. Hence, $\vdash v_1 : r_2$. By induction, we find $r'_1 \in \mathcal{PD}(r_1, l)$ such that $projE_{(r'_1,r_1,l)}\ v_1$ yields $\mathit{Just}\ (v'_l, v')$ where $\vdash v'_l : l$, $\vdash v' : r'_1$ and $|v'| = w$.

Application for this case exactly results into the call $projE_{(r'_1,r_1,l)}\ v_1$. Note that $r'_1 \in \mathcal{PD}(r_1 + r_2, l)$. Hence, we can establish the induction step.

The other cases can be proven similarly. □

The correctness result for *projPD* follows straightforwardly from *projE*.

LEMMA 4.8 (*projPD* Correctness). *Let r be a regular expression, l a literal, v_l, v proof terms and w a word such that $\vdash v_l : l$, $\vdash v : r$ and $|v| = v_l w$. Then, $\mathit{projPD}_{(\mathcal{PD}(r,l),r,l)}\ v$ yields $\mathit{Just}\ (v'_l, v')$ for some v'_l and v' such that $\vdash v'_l : l$, $\vdash v' : \mathsf{pd}(r, l)$ and $|v'| = w$.*

Based on the above correctness lemmas, correctness of *inj* and *proj* follows straightforwardly.

THEOREM 4.9 (Injection Correctness). *Let r_1 and r_2 be two regular expressions such that $\vdash r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let v be a proof term such that $\vdash v : r_1$. Then, $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v$ yields v' where $\vdash v' : r_2$ and $|v| = |v'|$.*

THEOREM 4.10 (Projection Correctness). *Let r_1 and r_2 be two regular expressions such that $\vdash r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let v_2 be a proof term such that $\vdash v_2 : r_2$. (1) Suppose there exists a proof term v_1 such that $\vdash v_1 : r_1$ and $|v_1| = |v_2|$. Then $\mathit{proj}_{(\mathcal{L},r_1,r_2)}\ v$ yields Just v where $\vdash v : r_1$ and $|v| = |v_2|$. (2) Suppose for all proof terms terms v_1 where $\vdash v_1 : r_1$ we have that $|v_1| \neq |v_2|$. Then $\mathit{proj}_{(\mathcal{L},r_1,r_2)}\ v$ yields $Nothing$.*

Note that the above results support injection and projection involving parametric regular expressions. The result below states that coercions involving parametric regular expressions are indeed parametric.

PROPOSITION 4.11 (Parametric Injection Correctness). *Let r_1 and r_2 be two regular expressions such that $\vdash r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let ψ be a substitution such that all parametric literals in $\mathcal{L}$ are mapped to monomorphic regular expressions. Let v be a proof term such that $\vdash v : \psi(r_1)$. Then $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v$ yields v' where $\vdash v' : \psi(r_2)$.*

Let v' such that $\vdash v' : r_1$ where for some mapping ρ we have that $\rho(v') = v$. Then, $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v = \rho(\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v')$.

A similar result holds for projection.

We conclude this subsection by stating the time complexity of injection and projection.

THEOREM 4.12 (Linear Time Complexity of Injection and Projection). *Let r_1 and r_2 be two regular expressions such that $\vdash r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let v be such that $\vdash v : r_1$. Then, the time complexity of computing $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v$ and $\mathit{proj}_{(\mathcal{L},r_1,r_2)}\ v$ is linear in the size of $|v|$.*

PROOF. (Sketch) The number of partial derivatives is bound by the size of r_1 and r_2. Hence, assumed to be a constant. The time complexities of function $isEps$, $mkEps$, *injE* and *projE* is dependent on the size of regular expressions involved. Hence, assumed to be a constant. Calls of *injPD* and *projPD* result in a constant number of calls to *injE* and *proj*E.

Suppose n is the size of $|v|$. Then, the actual cost factor are the n calls of *projPD* to extract all literals and the n calls of *injPD* to put back all literals. Hence, we obtain time complexity linear in the size of $|v|$. □

4.4 Greedy Left-Most

Our goal is to verify that injection and projection functions always produce greedy left-most parse trees. As we have seen, proof terms are constructed by putting back all extracted literals via functions *injPD* and *injE* starting with an empty proof term built by $mkEps_r$ for $\epsilon \in r$.

Our first observation is that $mkEps_r$ always yields the greedy left-most proof terms. Formally, we find the following result.

LEMMA 4.13 (Greedy Left-Most Empty String). *Let r be a regular expression such that $\epsilon \in r$. Then, $\vdash\ mkEps_r : r$ and and $mkEps_r$ is greedy left-most among all empty proof terms for r.*

So, will repeated applications of *injE* result in a greedy left-most proof term? Yes as we will show. Does *injE* itself always produce greedy left-most proof terms? No. Does *injE* itself preserve the greedy left-most property? No. The property we rely on to obtain the greedy left-most proof is a bit more subtle. The observation is that we apply *injE* always on partial derivatives which are "left-most". By left-most we mean the following.

Suppose r_0 is the initial regular expression on the right-hand side of a successful containment proof. Suppose we consider putting back the sequence of literals $l_1 ... l_n$. Then, we will repeatedly apply $inj_{(r_i, r_{i-1}, l_i)}$ on (l_i, v_i) where $r_i \in \mathcal{PD}(r_{i-1}, l_i)$ and $\vdash\ v_i : r_i$. The sequence of partial derivatives and their proof terms is as follows:

$$\vdash\ v_0 : r_0 \overset{l_1}{\leftarrow} \vdash\ v_1 : r_1 ... \vdash\ v_i : r_i ... \overset{l_n}{\leftarrow} \vdash\ v_n : r_n$$

where $\epsilon \in r_n$ and $v_n = mkEps_{r_n}$.

Each combination $\vdash\ v_i : r_i$ is left-most in the sense that there exists no r_i' and v_i' such that $\mathcal{PD}(r_{i-1}, l_{i-1}) = \{..., r_i', ..., r_i, ...\}$, $\vdash\ v_i' : r_i'$ and $|v_i'| = |v_i|$. This property is guaranteed by the structure of the containment proof.

Hence, we claim that *injE* preserves the greedy left-most ordering not in general. Only for proof terms which result from left-most successful partial derivatives.

For example, consider $a^* + (a^*, a^*)$ where

$$\begin{array}{lcl} \mathcal{PD}(a^* + (a^*, a^*), a) & = & \mathcal{PD}(a^*, a) \cup \mathcal{PD}((a^*, a^*) \\ & = & \{(\epsilon, a^*)\} \cup \{((\epsilon, a^*), a^*), (\epsilon, a^*)\} \\ & = & \{(\epsilon, a^*), ((\epsilon, a^*), a^*)\} \end{array}$$

Partial derivative (ϵ, a^*) appears twice but we favor $(\epsilon, a^*) \in \mathcal{PD}(a^*, a)$.

Consider term $(((), [a]), [])$ which is the greedy left-most proof w.r.t. $((\epsilon, a^*), a^*)$. However expression $((\epsilon, a^*), a^*)$ is here not the left-most successful partial derivative. This would be (ϵ, a). Indeed, $inj_{(((\epsilon, a^*), a^*), a^* + (a^*, a^*), a)}\ (a, (((), [a]), []))$ yields proof *Right* $([a, a], [])$ which is clearly not greedy left-most. The greedy left-most proof term is *Left* $[a, a]$ which results from $inj_{((\epsilon, a^*), a^* + (a^*, a^*), a)}\ (a, ((), [a]))$.

The above observation can be generalized into the following result.

LEMMA 4.14 (*injE* Greedy Left-Most Preservation). *Let r, r' be two regular expressions, and l a literal such that $r' \in \mathcal{PD}(r, l)$. Let v, v' and v_l be proof terms such that $\vdash\ v' : r'$ and $\vdash\ v_l : l$ where v' is the greedy left-most proof term w.r.t. r'. There exists no $r'' \in \mathcal{PD}(r, l)$ and v'' such that $\mathcal{PD}(r, l) = \{..., r'', ..., r', ...\}$ and $\vdash\ v'' : r''$ where $|v''| = |v'|$. Then, we have that $injPD_{(r', r, l)}\ (v_l, v')$ is the greedy left-most proof term w.r.t. r.*

PROOF. (Sketch) As can been seen from the definition of *injE*, in all but two cases we strictly apply *injE* to the left-most component. Thus, we can immediately argue that for these cases *injE* preserves the greedy left-most property.

Among the two exceptional cases, we first consider case $r_1 + r_2$ where $r' \in \mathcal{PD}(r_2, l)$ and $r' \notin \mathcal{PD}(r_1, l)$. As a result we find *Right* v_2 where v_2 results from the sub-call $injE_{(r', r_2, l)}\ (v_l, v')$.

Suppose that *Right* v_2 is not greedy left-most. Then, there must exist v_1 such that *Left* v_1 is the greedy left-most w.r.t. $r_1 + r_2$. However, then there must exist $r'' \in \mathcal{PD}(r, l)$ and v'' such that $\mathcal{PD}(r, l) = \{..., r'', ..., r', ...\}$ and $\vdash\ v'' : r''$ where $|v''| = |v'|$. This contradicts the assumption. Hence, *Right* v_2 is greedy left-most.

We consider the second exceptional case. Case (r_1, r_2) for $r' \in \mathcal{PD}(r_2, l) \wedge \epsilon \in r_1$ where for argument (v_l, v') we obtain the proof term $(mkEps_{r_1}, injE_{(r', r_2, l)}(v_l, v))$. Suppose this proof is not greedy left-most. As before we can argue that then there must exist $r'' \in \mathcal{PD}(r, l)$ and v'' such that $\mathcal{PD}(r, l) = \{..., r'', ..., r', ...\}$ and $\vdash\ v'' : r''$ where $|v''| = |v'|$. Yet again we have reached a contradiction which completes the proof. □

The greedy left-most property of injection and projection follows then directly from Lemmas 4.13 and 4.14 and the observation we made about left-most successful partial derivatives.

THEOREM 4.15 (Greedy Left-Most Injection). *Let r_1 and r_2 be two regular expressions such that $\vdash\ r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let v be a proof term such that $\vdash\ v : r_1$. Then, we have that $\vdash\ inj_{(\mathcal{L}, r_1, r_2)}\ v : r_2$ where $inj_{(\mathcal{L}, r_1, r_2)}\ v$ is the greedy left-most proof term.*

The case of projection is similar.

THEOREM 4.16 (Greedy Left-Most Projection). *Let r_1 and r_2 be two regular expressions such that $\vdash\ r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let v_2 be a proof term such that $\vdash\ v_2 : r_2$. (1) If there exists a proof term v_1 such that $\vdash\ v_1 : r_1$ and $|v_1| = |v_2|$, then $proj_{(\mathcal{L}, r_1, r_2)}\ v$ evaluates to Just v where $\vdash\ v : r_1$ and v is the greedy left-most proof term. (2) If for all proof terms terms v_1 where $\vdash\ v_1 : r_1$ we have that $|v_1| \neq |v_2|$, then $proj_{(\mathcal{L}, r_1, r_2)}\ v$ evaluates to Nothing.*

4.5 Faithful Instantiation

We consider the issue of applying parametric coercions in a specialized context. We wish that the parametric coercion behaves as the specialized coercions obtained for this specific context. This is what we refer to as faithfulness. Its formal definition is as follows.

DEFINITION 4 (Faithful Instance). *Let r_1 and r_2 be two regular expressions such that $\vdash\ r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let ψ be a substitution such that all parametric literals in $\mathcal{L}$ are mapped to monomorphic regular expressions. We write $\psi(\mathcal{L})$ to denote the set of literals in $\psi(r_1)$ and $\psi(r_2)$.*

We say that ψ is faithful w.r.t. $\vdash\ r_1 \leq r_2$ iff for all v such that $\vdash\ v : \psi(r_1)$ we have that

1. *$inj_{(\mathcal{L}, r_1, r_2)}\ v$ and $inj_{(\psi(\mathcal{L}), \psi(r_1), \psi(r_2))}\ v$ yield the same, and*
2. *$proj_{(\mathcal{L}, r_1, r_2)}\ v$ and $proj_{(\psi(\mathcal{L}), \psi(r_1), \psi(r_2))}\ v$ yield the same.*

In general, parametric coercions are not faithful. For example, consider $\vdash\ \alpha \leq a + \alpha$ where a is a constant and α a parametric

variable. The resulting (simplified) injection and projection functions are:

$$\mathit{inj} \;=\; \lambda\, v.\, \mathit{Right}\; v$$

$$\begin{array}{rl} \mathit{proj} \;=\; \lambda\, v.\, \mathsf{case}\; v\; \mathsf{of} & \\ & \mathit{Right}\; v' \;\rightarrow\; v' \\ & \mathit{Left}\; v' \;\rightarrow\; \mathit{Nothing} \end{array}$$

It is easy to see that for instance $\psi = [\alpha \mapsto a]$ the parametric injection as well as parametric projection function differ from their monmorphized counterparts derived from $\vdash\; a \leq a + a$.

$$\mathit{inj}_{\psi} \;=\; \lambda\, v.\, \mathit{Left}\; v$$

$$\begin{array}{rl} \mathit{proj}_{\psi} \;=\; \lambda\, v.\, \mathsf{case}\; v\; \mathsf{of} & \\ & \mathit{Right}\; v' \;\rightarrow\; v' \\ & \mathit{Left}\; v' \;\rightarrow\; v' \end{array}$$

It is possible that only one among the pair of parametric injection/projection functions is unfaithful. For example, consider the variant $\vdash\; \alpha \leq \alpha + a$ and substitution $\psi_2 = [\alpha \rightarrow b]$ which results in the parametric cases

$$\mathit{inj}' \;=\; \lambda\, v.\, \mathit{Left}\; v$$

$$\begin{array}{rl} \mathit{proj}' \;=\; \lambda\, v.\, \mathsf{case}\; v\; \mathsf{of} & \\ & \mathit{Left}\; v' \;\rightarrow\; v' \\ & \mathit{Right}\; v' \;\rightarrow\; \mathit{Nothing} \end{array}$$

and the monomorphic cases

$$\mathit{inj}'_{\psi_2} \;=\; \lambda\, v.\, \mathit{Left}\; v$$

$$\begin{array}{rl} \mathit{proj}'_{\psi_2} \;=\; \lambda\, v.\, \mathsf{case}\; v\; \mathsf{of} & \\ & \mathit{Left}\; v' \;\rightarrow\; v' \\ & \mathit{Right}\; v' \;\rightarrow\; v' \end{array}$$

Functions inj' and inj'_{ψ_2} still agree whereas proj' and proj'_{ψ_2} differ.

Our goal is to identify sufficient conditions which guarantee that parametric injection/projection functions, respectively at least one of them, remain faithful w.r.t. some specific instance. A sufficient condition is unambiguity of the instantiated regular expressions.

THEOREM 4.17 (Unambiguity yields Faithfulness). *Let r_1 and r_2 be two regular expressions such that $\vdash\; r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. Let ψ be a substitution such that all parametric literals in $\mathcal{L}$ are mapped to monomorphic regular expressions where $\psi(r_1)$ and $\psi(r_2)$ are unambiguous. Then, ψ is faithful w.r.t. $\vdash\; r_1 \leq r_2$.*

PROOF. (Sketch) We consider the case of injection. Suppose we have v such that $\vdash\; v : \psi(r_1)$. By correctness of injection (Theorem 4.9) and parametric instantiation (Proposition 4.11) we find that $\vdash\; \mathit{inj}_{(\mathcal{L},r_1,r_2)}\; v : \psi(r_2)$ and $\vdash\; \mathit{inj}_{(\psi(\mathcal{L}),\psi(r_1),\psi(r_2))}\; v : \psi(r_2)$. Clearly, $|\mathit{inj}_{(\mathcal{L},r_1,r_2)}\; v| = |\mathit{inj}_{(\psi(\mathcal{L}),\psi(r_1),\psi(r_2))}\; v|$. By unambiguity of $\psi(r_2)$, it holds that

$$\mathit{inj}_{(\mathcal{L},r_1,r_2)}\; v = \mathit{inj}_{(\psi(\mathcal{L}),\psi(r_1),\psi(r_2))}\; v$$

The other case can be proven similarly. □

From the above we can conclude that $\psi(r_2)$ unambiguous is relevant for faithfulness of injection whereas $\psi(r_1)$ unambiguous is relevant for projection. Note that unambiguity of the parametric expressions is not sufficient to guarantee unambiguity of the instantiation. For example, consider $\alpha \;\leq\; \alpha + \beta$ and instance $\psi = [\alpha \mapsto a, \beta \mapsto a]$.

Unambiguity (of instantiation) is a fairly strong condition. Typically we will encounter ambiguous expressions. After all, we apply greedy left-most to provide for a predictable coercion semantics. So, the next question we address is as follows. What are sufficient conditions which guarantee that a parametric greedy left-most coercion still behaves like a greedy left-most coercion under instantiation? We identify two conditions. Before diving into the formal details, we provide some motivation.

The first condition concerns instantiation of variables α. Recall that variables α are treated like Skolem constants. Hence, monomorphic values of type $\psi(\alpha)$ will not be touched by injection/projection functions generated from the parametric proof. Recall Proposition 4.11. Therefore, the substitutes for α must always be unambiguous as we will shortly see.

The second condition ensures that the structure of the containment proof remains "stable" under instantiation. Recall that injection/projection boils down to calls of *projPD*, $projE$, *injPD* and *injE*. The order and kind in which these functions are called is determined by the containment proof structure. So, we require a condition which ensures that the structure of the monomorphic proof does not change in any essential way from the structure of the parametric proof.

As we know, the containment proof structure is determined by the set of partial derivatives. For example, consider

$$r = (l_1, r_{1_1} + r_{1_2}) + (l_2, r_{2_1} + r_{2_2})$$

where $\mathcal{PD}(r, l_1) = \{r_{1_1}, r_{1_2}\}$ and $\mathcal{PD}(r, l_2) = \{r_{2_1}, r_{2_2}\}$.

The monomorphic coercion will most likely differ from the parametric coercions if for example there is an overlap between $\psi((l_1, r_{1_1} + r_{1_2}))$ and $\psi((l_2, r_{2_1} + r_{2_2}))$. Another possible source for a difference in behavior is if there exists some overlap among the set of partial derivatives. For example, among $\psi(r_{1_1})$ and $\psi(r_{1_2})$. Intuitively, if none of the two above overlaps arise under instantiation, we can guarantee that the monomorphic proof structure can be traced back to the parametric proof structure. Otherwise, the proof structures must differ which results in different behavior of the parametric and monomorphic coercion.

For example, consider $\vdash\; \alpha + \beta \;\leq\; \alpha + \beta$ for $\psi = [\alpha \mapsto a, \beta \mapsto a]$. We find that $\psi(\alpha)$ and $\psi(\beta)$ are overlapping. Indeed, faithfulness does not hold here.

For example, consider $(\alpha^*, \alpha) \;\leq\; (\alpha^*, \alpha)$ for $\psi = [\alpha \mapsto a^*]$. We have that $\mathcal{PD}((\alpha^*, \alpha), \alpha) = \{((\epsilon, \alpha^*), \alpha), \epsilon\}$. We encounter here an overlap within the set of partial derivatives. That is, $\psi(((\epsilon, \alpha^*), \alpha))$ and $\psi(\epsilon)$ overlap because ϵ is in the language denoted by $\psi(((\epsilon, \alpha^*), \alpha))$. Yet again, faithfulness does not hold here. The parametric injection is $\lambda(vs, v).(vs, v)$. On the other hand, the (simplified) monomorphic version is:
$\lambda(vs, v).(vs, v).([\mathit{concat}\; vs \;\texttt{++}\; v], [])$.

Thus motivated we give the full formalization.

DEFINITION 5 (Non-Overlap under Instantiation). *Let r be a regular expression and ψ be a substitution where $\mathcal{L}$ denotes the set of literals in r. We say that r's partial derivatives are non-overlapping w.r.t. ψ iff the following two conditions hold:*

1. $\forall l, l' \in \mathcal{L}.\forall r' \in \mathcal{PD}_{cl}(r, \mathcal{L}).\forall r_1 \in \mathcal{PD}(r', l), r_2 \in \mathcal{PD}(r, l')$ *such that $l \neq l'$ we have that $L(\psi(l, r_1)) \cap L(\psi(l', r_2)) = \emptyset$.*
2. $\forall l \in \mathcal{L}.\forall r' \in \mathcal{PD}_{cl}(r, \mathcal{L}).\forall r_1, r_2 \in \mathcal{PD}(r', l)$ *such that $r_1 \neq r_2$ we have that $L(\psi(r_1)) \cap L(\psi(r_2)) = \emptyset$.*

DEFINITION 6 (Unambiguous Variable Instantiation). *Let $\mathcal{L}$ be the set of literals. We say that ψ is a unambiguous variable instantiation iff $\psi(\alpha)$ is unambiguous for each $\alpha \in \mathcal{L}$.*

The above examples show that both conditions in Definition 5 are necessary. We also want to convince ourselves that the unambiguous variable instantiation is necessary. In fact, we can verify that faithfulness is always destroyed once this condition is violated.

LEMMA 4.18 (Necessity of Unambiguous Variable Instantiation). *Let r_1 and r_2 be two regular expressions such that $\vdash\; r_1 \leq r_2$ where $\mathcal{L}$ denotes the set of literals in r_1 and r_2. We assume that r_1 does not denote the empty language ϕ. Let ψ be a substitution such that all parametric literals in $\mathcal{L}$ are mapped to monomorphic regular expressions and there exists $\alpha \in \mathcal{L}$ such that $\psi(\alpha)$ is ambiguous and $\alpha \in fv(r_1)$ and $\alpha \in fv(r_2)$. Then, ψ is not faithful w.r.t. $\vdash\; r_1 \leq r_2$.*

PROOF. We show that injection is not faithful. By assumption $\psi(\alpha)$ is ambiguous. Hence, there exists $\vdash\ v_1 : \psi(\alpha)$, $\vdash\ v_2 : \psi(\alpha)$ where $v_1 \neq v_2$ and $|v_1| = |v_2|$ (1).

Expression r_1 does not denote the empty language. Hence, there exists v such that $\vdash\ v : r_1$. From v and (1) we can derive v_1' and v_2' such that $\vdash\ v_1' : \psi(r_1)$ and $\vdash\ v_2' : \psi(r_1)$. Proof term v_1' is derived from v by replacing x_α with v_1 and v_2' is derived from v by replacing x_α with v_2. That is, there exists mappings ρ_1 and ρ_2 such that $\rho_1(x_\alpha) = v_1$, $\rho_1(v) = v_1'$, $\rho_2(x_\alpha) = v_2$ and $\rho_2(v) = v_2'$.

Next, we argue that either

$$\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v_1' \neq \mathit{inj}_{(\psi(\mathcal{L}),\psi(r_1),\psi(r_2))}\ v_1'$$

or

$$\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v_2' \neq \mathit{inj}_{(\psi(\mathcal{L}),\psi(r_1),\psi(r_2))}\ v_2'$$

We refer to this property as (2).

The reason is as follows. From Proposition 4.11 we obtain that $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v_1' = \rho_1(\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v)$ and $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v_2' = \rho_2(\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v)$. Clearly, $\rho_1(\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v) \neq \rho_2(\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v)$ because $\alpha \in \mathit{fv}(r_2)$ and from (1) we have that $|\rho_1(\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v)| = |\rho_2(\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v)|$.

Suppose $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v_1'$ is not greedy left-most. Then, we immediately find $\mathit{inj}_{(\mathcal{L},r_1,r_2)}\ v_1' \neq \mathit{inj}_{(\psi(\mathcal{L}),\psi(r_1),\psi(r_2))}\ v_1'$.

Hence, (2) holds and thus we have shown that ψ is not faithful w.r.t. $\vdash\ r_1 \leq r_2$. □

The conditions stated by Definitions 5 and 6 are not only necessary. They are also sufficient. We take a look at some of the proof details.

The first sub-condition in Definition 5 ensures that the behavior of *projPD* in the parametric case is still valid if applied to a monomorphic value and matches the behavior of the monomorphic case.

The second sub-condition ensures that the behavior of *projE* is still valid and consistent with the monomorphic case. Consider the specific situation $(l_1, r_{1_1} + r_{1_2})$. Suppose in the parametric case *projE* succeeds on r_{1_2} but fails (with $Nothing$) on r_{1_2}. Then, the second sub-condition guarantees that in the monomorphic case *projE* will not succeed on some partial derivative r' where $r' \in \mathcal{PD}(c, \psi(r_{1_1}))$ where we assume that $\psi(l_1) = c$. Simply, because there is no 'overlap' among r_{1_1} and r_{1_2}.

The condition in Definition 6 guarantees that the parametric injection functions *injPD* and *injE* are faithful.

In conclusion, we arrive at the following proposition.

PROPOSITION 4.19. *Unambiguous variable instantiation and non-overlapping of instantiation are sufficient to guarantee faithfulness.*

5. Implementation

We have fully implemented the coercion system (Figures 3, 4 and 5) in Template Haskell [18]. The main steps include:

- Turn the contracts into assertions.
- Turn the application of type-indexed functions into splices.
- Lift the body of the type-indexed functions into the quasi-quotation.
- Explicitly keep tracks of the coinduction context.

Thus, Template Haskell resolves at compile-time all type-indexed functions and we obtain 'simple' coercion functions.

Figure 6 shows the Template Haskell implementation of $inj_{(Lit,r1,r2)}$ from Figure 3. We use the Haskell library `Data.Map` to implement the coinduction context ctx. It maps the regular expression containment relations that we have seen so far to the quasi-quoted function names. At location `(1)`, we apply the coinduction hypothesis to return the quasi-quoted function name from the context $ctxt$ indexed by (r_1, r_2). At location `(2)`, we create a fresh quasi-quoted function name which will be bound to the coercion function. We also extend the co-inductive context by inserting the quasi-quoted function name under the index of (r_1, r_2).

In future work, we plan to expand our Template Haskell encoding of the coercion system into a full-scale language extension which integrates regular expression types into a parametrically typed programming language. Thus lifting some of the limitations of earlier works [4, 19].

For example, the work in [4] describes a merger of OCaml and XDuce. A significant limitation is that OCaml and XDuce components are strictly separated. Hence, it is not possible to directly use OCaml polymorphism in XDuce programs.

The work [19] combines regular expression types and parametric polymorphism but does neither address disambiguation nor the issue of faithful instantiation. Hence, while the system in [19] is sound, its semantic is not predictable for the user.

6. Related Work and Conclusion

Lu and Sulzmann [16] appear to be the first to show how to derive coercion functions out of regular expression containment proofs. Like our work, theirs is based on the concept of partial derivatives [2]. Their algorithm effectively gives a Curry-Howard interpretation of the original partial derivative-based containment algorithm described in [1]. Our algorithm significantly differs in that we view containment coercions as pairs of type-indexed families of injection/projection functions. We also provide concise correctness results and verify that coercions satisfy the greedy left-most property.

Henglein and Nielsen [9] give a characterization of containment coercions based on a novel co-inductive regular expression containment proof system. Their main interest is in characterizing *all* possible containment proofs. In contrast, our *specific* focus is on greedy left-most proofs which can be naturally extracted out of the partially derivative-based containment proof system. In particular, we give an in-depth discussion of issues concerning the use of parametric coercions. Neither [16] nor [9] cover this important issue.

There exists a number of alternative formulations of regular expression containment proof system such as e.g. [7]. We refer to [9] for an overview. From a practical point of view, we believe it is important to enforce a disambiguation strategy such as e.g. greedy left-most. We consider it as some worthwhile future work to investigate if via some other proof method we can obtain "better", e.g. smaller, greedy left-most proofs, or can support alternative disambiguation strategies.

Our second main focus is on the faithful instantiation of parametric coercions. As far as we know, this important issue has so far not been studied at all in the context of containment proof systems. In the context of typed XML processing languages [11], similar issues arise due to the use of semantic subtyping [6] and pattern matching [12]. In our terminology, semantic subtyping corresponds to injection and pattern matching to projection.

For example, Hosoya, Frisch and Castagna [10] develop an extension of XDuce [11] with parametric polymorphism. Their key idea is to interpret type variables as "markings" to (sub)values. Markings have no semantic meaning. Run-time values carry markings as well and the type system guarantees that the markings in the output must be traced back to the markings in the input. This is somewhat reminiscent of Reynold's parametricity theory and allows them to adopt the existing XDuce semantics without having to rely on run-time type checking.

As already mentioned in the earlier Section 2, this form of check effectively tries to guarantee that the program's meaning remains faithful under *all* instantiations. Vouillon [20] could slightly relax

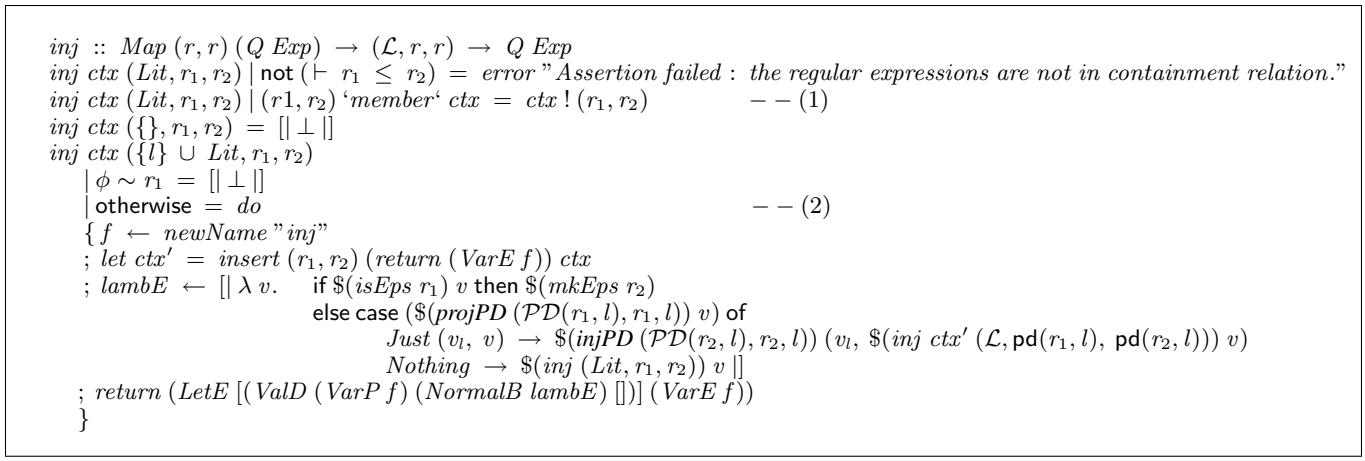


Figure 6: Injection function in Template Haskell

the conditions in [10]. Ultimately, enforcing faithfulness for all instantiations is often overly restrictive. The novel approach which we pursue is to develop sufficient and necessary conditions which establish faithfulness for a specific instance.

To conclude. We have provided a novel method to derive coercions out of containment proofs. We have given an in-depth discussion of the issues of disambiguation and faithful instantiation including concise formal results.

References

[1] Valentin M. Antimirov. Rewriting regular inequalities. In *Proc. of FCT'95*, volume 965 of *LNCS*, pages 116–125. Springer-Verlag, 1995.

[2] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

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

[4] A. Frisch. OCaml + XDuce. In *Proc. of ICFP'06*, pages 192–200. ACM Press, 2006.

[5] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Proc. of ICALP'04*, pages 618– 629. Spinger-Verlag, 2004.

[6] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Proc. of LICS'02*, pages 137–146. IEEE Computer Society, 2002.

[7] Clemens Grabmayer. Using proofs by coinduction to find "traditional" proofs. In *Proc. of CALCO'05*, pages 175–193. Springer-Verlag, 2005.

[8] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[9] Fritz Henglein and Lasse Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. In *Proc. of POPL'11*, pages 385–398. ACM, 2011.

[10] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for xml. *ACM Trans. Program. Lang. Syst.*, 32(1):2:1–2:56, 2009.

[11] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proc. of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997, pages 226–244, 2000.

[12] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *Proc. of POPL'01*, pages 67–80. ACM Press, 2001.

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

[14] William A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[15] Mark P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, Department of Computer Science, September 1993.

[16] Kenny Zhuo Ming Lu and Martin Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004.

[17] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

[18] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.

[19] Martin Sulzmann and Kenny Zhuo Ming Lu. XHaskell - adding regular expression types to Haskell. In *Proc. of IFL'07*, volume 5083 of *LNCS*, pages 75–92. Springer, 2007.

[20] Jérôme Vouillon. Polymorphic regular tree types and patterns. In *Proc. of POPL'06*, pages 103–114. ACM Press, 2006.