POSIX Lexing with Derivatives of Regular Expressions

Fahad Ausaf[1], Roy Dyckhoff[2], and Christian Urban[3]

[1] King's College London
fahad.ausaf@icloud.com
[2] University of St Andrews
roy.dyckhoff@st-andrews.ac.uk
[3] King's College London
christian.urban@kcl.ac.uk

Abstract. Brzozowski introduced the notion of derivatives for regular expressions. They can be used for a very simple regular expression matching algorithm. Sulzmann and Lu cleverly extended this algorithm in order to deal with POSIX matching, which is the underlying disambiguation strategy for regular expressions needed in lexers. Their algorithm generates POSIX values which encode the information of *how* a regular expression matches a string—that is, which part of the string is matched by which part of the regular expression. In this paper we give our inductive definition of what a POSIX value is and show (*i*) that such a value is unique (for given regular expression and string being matched) and (*ii*) that Sulzmann and Lu's algorithm always generates such a value (provided that the regular expression matches the string). We show that (*iii*) our inductive definition of a POSIX value is equivalent to an alternative definition by Okui and Suzuki which identifies POSIX values as least elements according to an ordering of values. We also prove the correctness of Sulzmann's bitcoded version of the POSIX matching algorithm and extend the results to additional constructors for regular expressions.

Keywords: POSIX matching, Derivatives of Regular Expressions, Isabelle/HOL

theory *SizeBound*
imports *Lexer*
begin

⋆ This paper is a revised and expanded version of [**?**]. Compared with that paper we give a second definition for POSIX values introduced by Okui Suzuki [**?**,**?**] and prove that it is equivalent to our original one. This second definition is based on an ordering of values and very similar to, but not equivalent with, the definition given by Sulzmann and Lu [**?**]. The advantage of the definition based on the ordering is that it implements more directly the informal rules from the POSIX standard. We also prove Sulzmann & Lu's conjecture that their bitcoded version of the POSIX algorithm is correct. Furthermore we extend our results to additional constructors of regular expressions.

1 Bit-Encodings

datatype *bit* = *Z* | *S*

fun *code* :: *val* ⇒ *bit list*
where
  code Void = []
| *code* (*Char c*) = []
| *code* (*Left v*) = *Z* # (*code v*)
| *code* (*Right v*) = *S* # (*code v*)
| *code* (*Seq v1 v2*) = (*code v1*) @ (*code v2*)
| *code* (*Stars* []) = [*S*]
| *code* (*Stars* (*v* # *vs*)) = (*Z* # *code v*) @ *code* (*Stars vs*)

fun
  Stars___add :: *val* ⇒ *val* ⇒ *val*
where
  Stars___add v (*Stars vs*) = *Stars* (*v* # *vs*)

function
  decode′ :: *bit list* ⇒ *rexp* ⇒ (*val* ∗ *bit list*)
where
  decode′ ds ZERO = (*Void*, [])
| *decode′ ds ONE* = (*Void*, *ds*)
| *decode′ ds* (*CH d*) = (*Char d*, *ds*)
| *decode′* [] (*ALT r1 r2*) = (*Void*, [])
| *decode′* (*Z* # *ds*) (*ALT r1 r2*) = (*let* (*v*, *ds′*) = *decode′ ds r1 in* (*Left v*, *ds′*))
| *decode′* (*S* # *ds*) (*ALT r1 r2*) = (*let* (*v*, *ds′*) = *decode′ ds r2 in* (*Right v*, *ds′*))
| *decode′ ds* (*SEQ r1 r2*) = (*let* (*v1*, *ds′*) = *decode′ ds r1 in*
    let (*v2*, *ds′′*) = *decode′ ds′ r2 in* (*Seq v1 v2*, *ds′′*))
| *decode′* [] (*STAR r*) = (*Void*, [])
| *decode′* (*S* # *ds*) (*STAR r*) = (*Stars* [], *ds*)
| *decode′* (*Z* # *ds*) (*STAR r*) = (*let* (*v*, *ds′*) = *decode′ ds r in*
    let (*vs*, *ds′′*) = *decode′ ds′* (*STAR r*)
    in (*Stars___add v vs*, *ds′′*))
by *pat___completeness auto*

lemma *decode′___smaller*:
  assumes *decode′___dom* (*ds*, *r*)
  shows *length* (*snd* (*decode′ ds r*)) ≤ *length ds*
using *assms*
apply(*induct ds r*)
apply(*auto simp add*: *decode′.psimps split*: *prod.split*)
using *dual___order.trans* **apply** *blast*
by (*meson dual___order.trans le___SucI*)

termination decode′
apply(relation inv_image (measure(%cs. size cs) <*lex*> measure(%s. size s)) (%(ds,r). (r,ds)))
apply(auto dest!: decode′_smaller)
by (metis less_Suc_eq_le snd_conv)

definition
  decode :: bit list ⇒ rexp ⇒ val option
where
  decode ds r $\stackrel{def}{=}$ (let (v, ds′) = decode′ ds r
  in (if ds′ = [] then Some v else None))

lemma decode′_code_Stars:
  assumes ∀ v∈ set vs. ⊨ v : r ∧ (∀ x. decode′ (code v @ x) r = (v, x)) ∧ flat v ≠ []
  shows decode′ (code (Stars vs) @ ds) (STAR r) = (Stars vs, ds)
  using assms
  apply(induct vs)
  apply(auto)
  done

lemma decode′_code:
  assumes ⊨ v : r
  shows decode′ ((code v) @ ds) r = (v, ds)
using assms
  apply(induct v r arbitrary: ds)
  apply(auto)
  using decode′_code_Stars **by** blast

lemma decode_code:
  assumes ⊨ v : r
  shows decode (code v) r = Some v
  using assms **unfolding** decode_def
  by (smt append_Nil2 decode′_code old.prod.case)

2 Annotated Regular Expressions

datatype arexp =
  AZERO
| AONE bit list
| ACHAR bit list char
| ASEQ bit list arexp arexp
| AALTs bit list arexp list
| ASTAR bit list arexp

abbreviation
AALT bs r1 r2 $\stackrel{def}{=}$ *AALTs bs [r1, r2]*

fun *asize* :: *arexp* ⇒ *nat* **where**
asize AZERO = 1
| *asize (AONE cs) = 1*
| *asize (ACHAR cs c) = 1*
| *asize (AALTs cs rs) = Suc (sum_list (map asize rs))*
| *asize (ASEQ cs r1 r2) = Suc (asize r1 + asize r2)*
| *asize (ASTAR cs r) = Suc (asize r)*

fun
erase :: *arexp* ⇒ *rexp*
where
erase AZERO = ZERO
| *erase (AONE _) = ONE*
| *erase (ACHAR _ c) = CH c*
| *erase (AALTs _ []) = ZERO*
| *erase (AALTs _ [r]) = (erase r)*
| *erase (AALTs bs (r#rs)) = ALT (erase r) (erase (AALTs bs rs))*
| *erase (ASEQ _ r1 r2) = SEQ (erase r1) (erase r2)*
| *erase (ASTAR _ r) = STAR (erase r)*

fun *nonalt* :: *arexp* ⇒ *bool*
where
nonalt (AALTs bs2 rs) = False
| *nonalt r = True*

fun *good* :: *arexp* ⇒ *bool* **where**
good AZERO = False
| *good (AONE cs) = True*
| *good (ACHAR cs c) = True*
| *good (AALTs cs []) = False*
| *good (AALTs cs [r]) = False*
| *good (AALTs cs (r1#r2#rs))* = ($\forall r' \in$ *set (r1#r2#rs). good* $r' \wedge$ *nonalt* r')
| *good (ASEQ _ AZERO _) = False*
| *good (ASEQ _ (AONE _) _) = False*
| *good (ASEQ _ _ AZERO) = False*
| *good (ASEQ cs r1 r2) = (good r1* $\wedge$ *good r2)*
| *good (ASTAR cs r) = True*

```
fun fuse :: bit list ⇒ arexp ⇒ arexp where
  fuse bs AZERO = AZERO
| fuse bs (AONE cs) = AONE (bs @ cs)
| fuse bs (ACHAR cs c) = ACHAR (bs @ cs) c
| fuse bs (AALTs cs rs) = AALTs (bs @ cs) rs
| fuse bs (ASEQ cs r1 r2) = ASEQ (bs @ cs) r1 r2
| fuse bs (ASTAR cs r) = ASTAR (bs @ cs) r

lemma fuse_append:
  shows fuse (bs1 @ bs2) r = fuse bs1 (fuse bs2 r)
  apply(induct r)
  apply(auto)
  done


fun intern :: rexp ⇒ arexp where
  intern ZERO = AZERO
| intern ONE = AONE []
| intern (CH c) = ACHAR [] c
| intern (ALT r1 r2) = AALT [] (fuse [Z] (intern r1))
                               (fuse [S]  (intern r2))
| intern (SEQ r1 r2) = ASEQ [] (intern r1) (intern r2)
| intern (STAR r) = ASTAR [] (intern r)


fun retrieve :: arexp ⇒ val ⇒ bit list where
  retrieve (AONE bs) Void = bs
| retrieve (ACHAR bs c) (Char d) = bs
| retrieve (AALTs bs [r]) v = bs @ retrieve r v
| retrieve (AALTs bs (r#rs)) (Left v) = bs @ retrieve r v
| retrieve (AALTs bs (r#rs)) (Right v) = bs @ retrieve (AALTs [] rs) v
| retrieve (ASEQ bs r1 r2) (Seq v1 v2) = bs @ retrieve r1 v1 @ retrieve r2 v2
| retrieve (ASTAR bs r) (Stars []) = bs @ [S]
| retrieve (ASTAR bs r) (Stars (v#vs)) =
     bs @ [Z] @ retrieve r v @ retrieve (ASTAR [] r) (Stars vs)


fun
 bnullable :: arexp ⇒ bool
where
```

bnullable (*AZERO*) = *False*
| *bnullable* (*AONE bs*) = *True*
| *bnullable* (*ACHAR bs c*) = *False*
| *bnullable* (*AALTs bs rs*) = (∃ *r* ∈ *set rs*. *bnullable r*)
| *bnullable* (*ASEQ bs r1 r2*) = (*bnullable r1* ∧ *bnullable r2*)
| *bnullable* (*ASTAR bs r*) = *True*

fun
bmkeps :: *arexp* ⇒ *bit list*
where
bmkeps(*AONE bs*) = *bs*
| *bmkeps*(*ASEQ bs r1 r2*) = *bs* @ (*bmkeps r1*) @ (*bmkeps r2*)
| *bmkeps*(*AALTs bs* [*r*]) = *bs* @ (*bmkeps r*)
| *bmkeps*(*AALTs bs* (*r*#*rs*)) = (*if bnullable*(*r*) *then bs* @ (*bmkeps r*) *else* (*bmkeps* (*AALTs bs rs*)))
| *bmkeps*(*ASTAR bs r*) = *bs* @ [*S*]

fun
bder :: *char* ⇒ *arexp* ⇒ *arexp*
where
bder c (*AZERO*) = *AZERO*
| *bder c* (*AONE bs*) = *AZERO*
| *bder c* (*ACHAR bs d*) = (*if c* = *d then AONE bs else AZERO*)
| *bder c* (*AALTs bs rs*) = *AALTs bs* (*map* (*bder c*) *rs*)
| *bder c* (*ASEQ bs r1 r2*) =
(*if bnullable r1*
then AALT bs (*ASEQ* [] (*bder c r1*) *r2*) (*fuse* (*bmkeps r1*) (*bder c r2*))
else ASEQ bs (*bder c r1*) *r2*)
| *bder c* (*ASTAR bs r*) = *ASEQ bs* (*fuse* [*Z*] (*bder c r*)) (*ASTAR* [] *r*)

fun
bders :: *arexp* ⇒ *string* ⇒ *arexp*
where
bders r [] = *r*
| *bders r* (*c*#*s*) = *bders* (*bder c r*) *s*

lemma *bders___append*:
bders r (*s1* @ *s2*) = *bders* (*bders r s1*) *s2*
apply(*induct s1 arbitrary*: *r s2*)
apply(*simp___all*)
done

lemma *bnullable___correctness*:

```
  shows nullable (erase r) = bnullable r
  apply(induct r rule: erase.induct)
  apply(simp_all)
  done

lemma erase_fuse:
  shows erase (fuse bs r) = erase r
  apply(induct r rule: erase.induct)
  apply(simp_all)
  done

thm Posix.induct

lemma erase_intern [simp]:
  shows erase (intern r) = r
  apply(induct r)
  apply(simp_all add: erase_fuse)
  done

lemma erase_bder [simp]:
  shows erase (bder a r) = der a (erase r)
  apply(induct r rule: erase.induct)
  apply(simp_all add: erase_fuse bnullable_correctness)
  done

lemma erase_bders [simp]:
  shows erase (bders r s) = ders s (erase r)
  apply(induct s arbitrary: r )
  apply(simp_all)
  done

lemma retrieve_encode_STARS:
  assumes ∀ v∈ set vs. ⊨ v : r ∧ code v = retrieve (intern r) v
  shows code (Stars vs) = retrieve (ASTAR [] (intern r)) (Stars vs)
  using assms
  apply(induct vs)
  apply(simp_all)
  done


lemma retrieve_fuse2:
  assumes ⊨ v : (erase r)
  shows retrieve (fuse bs r) v = bs @ retrieve r v
  using assms
  apply(induct r arbitrary: v bs)
```

```
      apply(auto elim: Prf_elims)[4]
  defer
 using retrieve_encode_STARS
  apply(auto elim!: Prf_elims)[1]
  apply(case_tac vs)
   apply(simp)
  apply(simp)

 apply(simp)
 apply(case_tac x2a)
  apply(simp)
  apply(auto elim!: Prf_elims)[1]
 apply(simp)
  apply(case_tac list)
  apply(simp)
 apply(auto)
 apply(auto elim!: Prf_elims)[1]
 done

lemma retrieve_fuse:
 assumes ⊨ v : r
 shows retrieve (fuse bs (intern r)) v = bs @ retrieve (intern r) v
 using assms
 by (simp_all add: retrieve_fuse2)


lemma retrieve_code:
 assumes ⊨ v : r
 shows code v = retrieve (intern r) v
 using assms
 apply(induct v r )
 apply(simp_all add: retrieve_fuse retrieve_encode_STARS)
 done


lemma bnullable_Hdbmkeps_Hd:
 assumes bnullable a
 shows  bmkeps (AALTs bs (a # rs)) = bs @ (bmkeps a)
 using assms
 by (metis bmkeps.simps(3) bmkeps.simps(4) list.exhaust)

lemma r1:
 assumes ¬ bnullable a bnullable (AALTs bs rs)
 shows  bmkeps (AALTs bs (a # rs)) = bmkeps (AALTs bs rs)
 using assms
```

```
  apply(induct rs)
   apply(auto)
  done

lemma r2:
  assumes x ∈ set rs bnullable x
  shows bnullable (AALTs bs rs)
  using assms
  apply(induct rs)
   apply(auto)
  done

lemma  r3:
  assumes ¬ bnullable r
          ∃ x ∈ set rs. bnullable x
  shows retrieve (AALTs bs rs) (mkeps (erase (AALTs bs rs))) =
        retrieve (AALTs bs (r # rs)) (mkeps (erase (AALTs bs (r # rs))))
  using assms
  apply(induct rs arbitrary: r bs)
   apply(auto)[1]
  apply(auto)
  using bnullable_correctness apply blast
   apply(auto simp add: bnullable_correctness mkeps_nullable retrieve_fuse2)
   apply(subst retrieve_fuse2[symmetric])
 apply (smt bnullable.simps(4) bnullable_correctness erase.simps(5) erase.simps(6)
insert_iff list.exhaust list.set(2) mkeps.simps(3) mkeps_nullable)
   apply(simp)
  apply(case_tac bnullable a)
 apply (smt append_Nil2 bnullable.simps(4) bnullable_correctness erase.simps(5)
erase.simps(6) fuse.simps(4) insert_iff list.exhaust list.set(2) mkeps.simps(3)
mkeps_nullable retrieve_fuse2)
  apply(drule_tac x=a in meta_spec)
  apply(drule_tac x=bs in meta_spec)
  apply(drule meta_mp)
   apply(simp)
  apply(drule meta_mp)
   apply(auto)
  apply(subst retrieve_fuse2[symmetric])
  apply(case_tac rs)
    apply(simp)
   apply(auto)[1]
      apply (simp add: bnullable_correctness)
  apply (metis append_Nil2 bnullable_correctness erase_fuse fuse.simps(4)
list.set_intros(1) mkeps.simps(3) mkeps_nullable nullable.simps(4) r2)
    apply (simp add: bnullable_correctness)
```

```
 apply (metis append_Nil2 bnullable_correctness erase.simps(6) erase_fuse
fuse.simps(4) list.set_intros(2) mkeps.simps(3) mkeps_nullable r2)
  apply(simp)
  done


lemma t:
  assumes ∀ r ∈ set rs. nullable (erase r) ⟶ bmkeps r = retrieve r (mkeps
(erase r))
          nullable (erase (AALTs bs rs))
 shows  bmkeps (AALTs bs rs) = retrieve (AALTs bs rs) (mkeps (erase (AALTs
bs rs)))
  using assms
  apply(induct rs arbitrary: bs)
   apply(simp)
  apply(auto simp add: bnullable_correctness)
   apply(case_tac rs)
     apply(auto simp add: bnullable_correctness)[2]
   apply(subst r1)
     apply(simp)
    apply(rule r2)
     apply(assumption)
    apply(simp)
   apply(drule_tac x=bs in meta_spec)
   apply(drule meta_mp)
    apply(auto)[1]
   prefer 2
  apply(case_tac bnullable a)
    apply(subst bnullable_Hdbmkeps_Hd)
     apply blast
    apply(subgoal_tac nullable (erase a))
  prefer 2
  using bnullable_correctness apply blast
  apply (metis (no_types, lifting) erase.simps(5) erase.simps(6) list.exhaust
mkeps.simps(3) retrieve.simps(3) retrieve.simps(4))
  apply(subst r1)
     apply(simp)
  using r2 apply blast
  apply(drule_tac x=bs in meta_spec)
   apply(drule meta_mp)
    apply(auto)[1]
   apply(simp)
  using r3 apply blast
  apply(auto)
  using r3 by blast
```

```
lemma bmkeps_retrieve:
  assumes nullable (erase r)
  shows bmkeps r = retrieve r (mkeps (erase r))
  using assms
  apply(induct r)
         apply(simp)
        apply(simp)
       apply(simp)
    apply(simp)
   defer
   apply(simp)
  apply(rule t)
   apply(auto)
  done

lemma bder_retrieve:
  assumes ⊨ v : der c (erase r)
  shows retrieve (bder c r) v = retrieve r (injval (erase r) c v)
  using assms
  apply(induct r arbitrary: v rule: erase.induct)
         apply(simp)
         apply(erule Prf_elims)
        apply(simp)
        apply(erule Prf_elims)
        apply(simp)
      apply(case_tac c = ca)
       apply(simp)
       apply(erule Prf_elims)
       apply(simp)
      apply(simp)
       apply(erule Prf_elims)
  apply(simp)
      apply(erule Prf_elims)
     apply(simp)
    apply(simp)
  apply(rename_tac r1 r2 rs v)
    apply(erule Prf_elims)
     apply(simp)
    apply(simp)
    apply(case_tac rs)
     apply(simp)
    apply(simp)
  apply (smt Prf_elims(3) injval.simps(2) injval.simps(3) retrieve.simps(4)
retrieve.simps(5) same_append_eq)
```

```
   apply(simp)
   apply(case_tac nullable (erase r1))
    apply(simp)
  apply(erule Prf_elims)
     apply(subgoal_tac bnullable r1)
  prefer 2
  using bnullable_correctness apply blast
    apply(simp)
     apply(erule Prf_elims)
     apply(simp)
   apply(subgoal_tac bnullable r1)
  prefer 2
  using bnullable_correctness apply blast
    apply(simp)
    apply(simp add: retrieve_fuse2)
    apply(simp add: bmkeps_retrieve)
   apply(simp)
   apply(erule Prf_elims)
   apply(simp)
  using bnullable_correctness apply blast
  apply(rename_tac bs r v)
  apply(simp)
  apply(erule Prf_elims)
     apply(clarify)
  apply(erule Prf_elims)
  apply(clarify)
  apply(subst injval.simps)
  apply(simp del: retrieve.simps)
  apply(subst retrieve.simps)
  apply(subst retrieve.simps)
  apply(simp)
  apply(simp add: retrieve_fuse2)
  done


lemma MAIN_decode:
  assumes ⊨ v : ders s r
  shows Some (flex r id s v) = decode (retrieve (bders (intern r) s) v) r
  using assms
proof (induct s arbitrary: v rule: rev_induct)
  case Nil
  have ⊨ v : ders [] r by fact
  then have ⊨ v : r by simp
  then have Some v = decode (retrieve (intern r) v) r
```

```
    using decode_code retrieve_code by auto
  then show Some (flex r id [] v) = decode (retrieve (bders (intern r) []) v) r
    by simp
next
  case (snoc c s v)
  have IH: ⋀v. ⊨ v : ders s r ⟹
     Some (flex r id s v) = decode (retrieve (bders (intern r) s) v) r by fact
  have asm: ⊨ v : ders (s @ [c]) r by fact
  then have asm2: ⊨ injval (ders s r) c v : ders s r
    by (simp add: Prf_injval ders_append)
  have Some (flex r id (s @ [c]) v) = Some (flex r id s (injval (ders s r) c v))
    by (simp add: flex_append)
  also have ... = decode (retrieve (bders (intern r) s) (injval (ders s r) c v)) r
    using asm2 IH by simp
  also have ... = decode (retrieve (bder c (bders (intern r) s)) v) r
    using asm by (simp_all add: bder_retrieve ders_append)
  finally show Some (flex r id (s @ [c]) v) =
                 decode (retrieve (bders (intern r) (s @ [c])) v) r by (simp add:
bders_append)
qed


definition blex where
 blex a s ≝ if bnullable (bders a s) then Some (bmkeps (bders a s)) else None



definition blexer where
 blexer r s ≝ if bnullable (bders (intern r) s) then
                decode (bmkeps (bders (intern r) s)) r else None

lemma blexer_correctness:
  shows blexer r s = lexer r s
proof −
  { define bds where bds ≝ bders (intern r) s
    define ds  where ds ≝ ders s r
    assume asm: nullable ds
    have era: erase bds = ds
      unfolding ds_def bds_def by simp
    have mke: ⊨ mkeps ds : ds
      using asm by (simp add: mkeps_nullable)
    have decode (bmkeps bds) r = decode (retrieve bds (mkeps ds)) r
      using bmkeps_retrieve
      using asm era by (simp add: bmkeps_retrieve)
    also have ... =  Some (flex r id s (mkeps ds))
```

```
      using mke by (simp___all add: MAIN___decode ds___def bds___def)
    finally have decode (bmkeps bds) r = Some (flex r id s (mkeps ds))
      unfolding bds___def ds___def .
  }
  then show blexer r s = lexer r s
    unfolding blexer___def lexer___flex
    apply(subst bnullable___correctness[symmetric])
    apply(simp)
    done
qed
```

```
fun distinctBy :: 'a list ⇒ ('a ⇒ 'b) ⇒ 'b set ⇒ 'a list
  where
  distinctBy [] f acc = []
| distinctBy (x#xs) f acc =
     (if (f x) ∈ acc then distinctBy xs f acc
      else x # (distinctBy xs f ({f x} ∪ acc)))
```

```
fun flts :: arexp list ⇒ arexp list
  where
  flts [] = []
| flts (AZERO # rs) = flts rs
| flts ((AALTs bs  rs1) # rs) = (map (fuse bs) rs1) @ flts rs
| flts (r1 # rs) = r1 # flts rs
```

```
fun li :: bit list ⇒ arexp list ⇒ arexp
  where
  li ___ [] = AZERO
| li bs [a] = fuse bs a
| li bs as = AALTs bs as
```

```
fun bsimp___ASEQ :: bit list ⇒ arexp ⇒ arexp ⇒ arexp
  where
  bsimp___ASEQ ___ AZERO ___ = AZERO
| bsimp___ASEQ ___ ___ AZERO = AZERO
```

| *bsimp___ASEQ bs1 (AONE bs2) r2 = fuse (bs1 @ bs2) r2*
| *bsimp___ASEQ bs1 r1 r2 = ASEQ bs1 r1 r2*

fun *bsimp___AALTs* :: *bit list ⇒ arexp list ⇒ arexp*
where
bsimp___AALTs ___ [] = AZERO
| *bsimp___AALTs bs1 [r] = fuse bs1 r*
| *bsimp___AALTs bs1 rs = AALTs bs1 rs*

fun *bsimp* :: *arexp ⇒ arexp*
where
bsimp (ASEQ bs1 r1 r2) = bsimp___ASEQ bs1 (bsimp r1) (bsimp r2)
| *bsimp (AALTs bs1 rs) = bsimp___AALTs bs1 (distinctBy (flts (map bsimp rs)) erase {})*
| *bsimp r = r*

fun
bders___simp :: *arexp ⇒ string ⇒ arexp*
where
bders___simp r [] = r
| *bders___simp r (c # s) = bders___simp (bsimp (bder c r)) s*

definition *blexer___simp* **where**
blexer___simp r s $\stackrel{def}{=}$ *if bnullable (bders___simp (intern r) s) then decode (bmkeps (bders___simp (intern r) s)) r else None*

export-code *bders___simp* **in** *Scala* **module-name** *Example*

lemma *bders___simp___append*:
shows *bders___simp r (s1 @ s2) = bders___simp (bders___simp r s1) s2*
apply(*induct s1 arbitrary*: *r s2*)
apply(*simp*)
apply(*simp*)
done

lemma *L___bsimp___ASEQ*:
L (*SEQ* (*erase r1*) (*erase r2*)) = *L* (*erase* (*bsimp___ASEQ bs r1 r2*))
apply(*induct bs r1 r2 rule*: *bsimp___ASEQ.induct*)
apply(*simp___all*)
by (*metis erase___fuse fuse.simps*(*4*))

lemma *L___bsimp___AALTs*:
L (*erase* (*AALTs bs rs*)) = *L* (*erase* (*bsimp___AALTs bs rs*))
apply(*induct bs rs rule*: *bsimp___AALTs.induct*)
apply(*simp___all add*: *erase___fuse*)
done

lemma *L___erase___AALTs*:
shows *L* (*erase* (*AALTs bs rs*)) = ⋃ (*L* ‘ *erase* ‘ (*set rs*))
apply(*induct rs*)
apply(*simp*)
apply(*simp*)
apply(*case___tac rs*)
apply(*simp*)
apply(*simp*)
done

lemma *L___erase___flts*:
shows ⋃ (*L* ‘ *erase* ‘ (*set* (*flts rs*))) = ⋃ (*L* ‘ *erase* ‘ (*set rs*))
apply(*induct rs rule*: *flts.induct*)
apply(*simp___all*)
apply(*auto*)
using *L___erase___AALTs erase___fuse* **apply** *auto*[*1*]
by (*simp add*: *L___erase___AALTs erase___fuse*)

lemma *L___erase___dB___acc*:
shows (⋃(*L* ‘ *acc*) ∪ (⋃ (*L* ‘ *erase* ‘ (*set* (*distinctBy rs erase acc*)))))
= ⋃(*L* ‘ *acc*) ∪ ⋃ (*L* ‘ *erase* ‘ (*set rs*))
apply(*induction rs arbitrary*: *acc*)
apply *simp*
apply *simp*
by (*smt* (*z3*) *SUP___absorb UN___insert sup___assoc sup___commute*)

lemma *L___erase___dB*:
shows (⋃ (*L* ‘ *erase* ‘ (*set* (*distinctBy rs erase* {})))) = ⋃ (*L* ‘ *erase* ‘ (*set rs*))
by (*metis L___erase___dB___acc Un___commute Union___image___empty*)

lemma *L___bsimp___erase*:

```
  shows L (erase r) = L (erase (bsimp r))
  apply(induct r)
  apply(simp)
  apply(simp)
  apply(simp)
  apply(auto simp add: Sequ_def)[1]
  apply(subst L_bsimp_ASEQ[symmetric])
  apply(auto simp add: Sequ_def)[1]
  apply(subst (asm)  L_bsimp_ASEQ[symmetric])
  apply(auto simp add: Sequ_def)[1]
   apply(simp)
   apply(subst L_bsimp_AALTs[symmetric])
   defer
   apply(simp)
  apply(subst (2)L_erase_AALTs)
  apply(subst L_erase_dB)
  apply(subst L_erase_flts)
  apply(auto)
   apply (simp add: L_erase_AALTs)
  using L_erase_AALTs by blast

lemma bsimp_ASEQ0:
  shows bsimp_ASEQ bs r1 AZERO = AZERO
  apply(induct r1)
  apply(auto)
  done



lemma bsimp_ASEQ1:
  assumes r1 ≠ AZERO r2 ≠ AZERO ∀ bs. r1 ≠ AONE bs
  shows bsimp_ASEQ bs r1 r2 = ASEQ bs r1 r2
  using assms
  apply(induct bs r1 r2 rule: bsimp_ASEQ.induct)
  apply(auto)
  done

lemma bsimp_ASEQ2:
  shows bsimp_ASEQ bs (AONE bs1) r2 = fuse (bs @ bs1) r2
  apply(induct r2)
  apply(auto)
  done


lemma L_bders_simp:
```

shows *L* (*erase* (*bders___simp r s*)) = *L* (*erase* (*bders r s*))
apply(*induct s arbitrary*: *r rule*: *rev___induct*)
apply(*simp*)
apply(*simp*)
apply(*simp add*: *ders___append*)
apply(*simp add*: *bders___simp___append*)
apply(*simp add*: *L___bsimp___erase*[*symmetric*])
by (*simp add*: *der___correctness*)

lemma *b2*:
assumes *bnullable r*
shows *bmkeps* (*fuse bs r*) = *bs* @ *bmkeps r*
by (*simp add*: *assms bmkeps___retrieve bnullable___correctness erase___fuse mkeps___nullable retrieve___fuse2*)

lemma *b4*:
shows *bnullable* (*bders___simp r s*) = *bnullable* (*bders r s*)
by (*metis L___bders___simp bnullable___correctness lexer.simps*(*1*) *lexer___correct___None option.distinct*(*1*))

lemma *qq1*:
assumes ∃ *r* ∈ *set rs*. *bnullable r*
shows *bmkeps* (*AALTs bs* (*rs* @ *rs1*)) = *bmkeps* (*AALTs bs rs*)
using *assms*
apply(*induct rs arbitrary*: *rs1 bs*)
apply(*simp*)
apply(*simp*)
by (*metis Nil___is___append___conv bmkeps.simps*(*4*) *neq___Nil___conv bnullable___Hdbmkeps___Hd split___list___last*)

lemma *qq2*:
assumes ∀ *r* ∈ *set rs*. ¬ *bnullable r* ∃ *r* ∈ *set rs1*. *bnullable r*
shows *bmkeps* (*AALTs bs* (*rs* @ *rs1*)) = *bmkeps* (*AALTs bs rs1*)
using *assms*
apply(*induct rs arbitrary*: *rs1 bs*)
apply(*simp*)
apply(*simp*)
by (*metis append___assoc in___set___conv___decomp r1 r2*)

lemma *qq3*:
shows *bnullable* (*AALTs bs rs*) = (∃ *r* ∈ *set rs*. *bnullable r*)
apply(*induct rs arbitrary*: *bs*)

```
  apply(simp)
  apply(simp)
  done


fun nonnested :: arexp ⇒ bool
  where
  nonnested (AALTs bs2 []) = True
| nonnested (AALTs bs2 ((AALTs bs1 rs1) # rs2)) = False
| nonnested (AALTs bs2 (r # rs2)) = nonnested (AALTs bs2 rs2)
| nonnested r = True


lemma k0:
  shows flts (r # rs1) = flts [r] @ flts rs1
  apply(induct r arbitrary: rs1)
   apply(auto)
  done

lemma k00:
  shows flts (rs1 @ rs2) = flts rs1 @ flts rs2
  apply(induct rs1 arbitrary: rs2)
   apply(auto)
  by (metis append.assoc k0)

lemma k0a:
  shows flts [AALTs bs rs] = map (fuse bs) rs
  apply(simp)
  done


lemma bsimp_AALTs_qq:
  assumes 1 < length rs
  shows bsimp_AALTs bs rs = AALTs bs rs
  using assms
  apply(case_tac rs)
```

```
 apply(simp)
apply(case_tac list)
 apply(simp_all)
done
```

```
lemma bbbbs1:
  shows nonalt r ∨ (∃ bs rs. r = AALTs bs rs)
  using nonalt.elims(3) by auto
```

```
lemma flts_append:
  flts (xs1 @ xs2) = flts xs1 @ flts xs2
  apply(induct xs1 arbitrary: xs2 rule: rev_induct)
   apply(auto)
  apply(case_tac xs)
   apply(auto)
   apply(case_tac x)
        apply(auto)
  apply(case_tac x)
        apply(auto)
  done
```

```
fun nonazero :: arexp ⇒ bool
  where
  nonazero AZERO = False
| nonazero r = True
```

```
lemma flts_single1:
  assumes nonalt r nonazero r
  shows flts [r] = [r]
  using assms
  apply(induct r)
  apply(auto)
  done
```

```
lemma q3a:
  assumes ∃ r ∈ set rs. bnullable r
```

shows *bmkeps* (*AALTs bs* (*map* (*fuse bs1*) *rs*)) = *bmkeps* (*AALTs* (*bs*@*bs1*) *rs*)
using *assms*
apply(*induct rs arbitrary*: *bs bs1*)
apply(*simp*)
apply(*simp*)
apply(*auto*)
apply (*metis append___assoc b2 bnullable___correctness erase___fuse bnullable___Hdbmkeps___Hd*)
apply(*case___tac bnullable a*)
apply (*metis append.assoc b2 bnullable___correctness erase___fuse bnullable___Hdbmkeps___Hd*)
apply(*case___tac rs*)
apply(*simp*)
apply(*simp*)
apply(*auto*)[*1*]
apply (*metis bnullable___correctness erase___fuse*)+
done

lemma *qq4*:
assumes ∃ *x*∈ *set list*. *bnullable x*
shows ∃ *x*∈ *set* (*flts list*). *bnullable x*
using *assms*
apply(*induct list rule*: *flts.induct*)
apply(*auto*)
by (*metis UnCI bnullable___correctness erase___fuse imageI*)

lemma *qs3*:
assumes ∃ *r* ∈ *set rs*. *bnullable r*
shows *bmkeps* (*AALTs bs rs*) = *bmkeps* (*AALTs bs* (*flts rs*))
using *assms*
apply(*induct rs arbitrary*: *bs taking*: *size rule*: *measure___induct*)
apply(*case___tac x*)
apply(*simp*)
apply(*simp*)
apply(*case___tac a*)
apply(*simp*)
apply (*simp add*: *r1*)
apply(*simp*)
apply (*simp add*: *bnullable___Hdbmkeps___Hd*)
apply(*simp*)
apply(*case___tac flts list*)
apply(*simp*)
apply (*metis L___erase___AALTs L___erase___flts L___flat___Prf1 L___flat___Prf2 Prf___elims*(*1*) *bnullable___correctness erase.simps*(*4*) *mkeps___nullable r2*)

```
    apply(simp)
    apply (simp add: r1)
   prefer 3
   apply(simp)
   apply (simp add: bnullable___Hdbmkeps___Hd)
  prefer 2
  apply(simp)
 apply(case___tac ∃ x∈ set x52. bnullable x)
 apply(case___tac list)
   apply(simp)
   apply (metis b2 fuse.simps(4) q3a r2)
  apply(erule disjE)
   apply(subst qq1)
    apply(auto)[1]
    apply (metis bnullable___correctness erase___fuse)
   apply(simp)
    apply (metis b2 fuse.simps(4) q3a r2)
   apply(simp)
   apply(auto)[1]
    apply(subst qq1)
     apply (metis bnullable___correctness erase___fuse image___eqI set___map)
    apply (metis b2 fuse.simps(4) q3a r2)
 apply(subst qq1)
     apply (metis bnullable___correctness erase___fuse image___eqI set___map)
   apply (metis b2 fuse.simps(4) q3a r2)
  apply(simp)
  apply(subst qq2)
    apply (metis bnullable___correctness erase___fuse imageE set___map)
 prefer 2
 apply(case___tac list)
    apply(simp)
   apply(simp)
  apply (simp add: qq4)
 apply(simp)
 apply(auto)
  apply(case___tac list)
   apply(simp)
  apply(simp)
  apply (simp add: bnullable___Hdbmkeps___Hd)
 apply(case___tac bnullable (ASEQ x41 x42 x43))
  apply(case___tac list)
   apply(simp)
  apply(simp)
  apply (simp add: bnullable___Hdbmkeps___Hd)
 apply(simp)
```

using *qq4 r1 r2* **by** *auto*

lemma *bder_fuse*:
shows *bder c (fuse bs a) = fuse bs (bder c a)*
apply(*induct a arbitrary*: *bs c*)
apply(*simp_all*)
done

fun *flts2* :: *char* ⇒ *arexp list* ⇒ *arexp list*
where
flts2 _ [] = []
| *flts2 c (AZERO # rs) = flts2 c rs*
| *flts2 c (AONE _ # rs) = flts2 c rs*
| *flts2 c (ACHAR bs d # rs) = (if c = d then (ACHAR bs d # flts2 c rs) else flts2 c rs)*
| *flts2 c ((AALTs bs rs1) # rs) = (map (fuse bs) rs1) @ flts2 c rs*
| *flts2 c (ASEQ bs r1 r2 # rs) = (if (bnullable(r1) ∧ r2 = AZERO) then flts2 c rs else ASEQ bs r1 r2 # flts2 c rs)*
| *flts2 c (r1 # rs) = r1 # flts2 c rs*

lemma *WQ1*:
assumes *s ∈ L (der c r)*
shows *s ∈ der c r → mkeps (ders s (der c r))*
using *assms*
oops

lemma *bder___bsimp___AALTs*:
shows *bder c* (*bsimp___AALTs bs rs*) = *bsimp___AALTs bs* (*map* (*bder c*) *rs*)
apply(*induct bs rs rule*: *bsimp___AALTs.induct*)
apply(*simp*)
apply(*simp*)
apply (*simp add*: *bder___fuse*)
apply(*simp*)
done

lemma
assumes *asize* (*bsimp a*) = *asize a* *a* = *AALTs bs* [*AALTs bs2* [], *AZERO*, *AONE bs3*]
shows *bsimp a* = *a*
using *assms*
apply(*simp*)
oops

inductive *rrewrite*:: *arexp* ⇒ *arexp* ⇒ *bool* (___ ⇝ ___ [*99*, *99*] *99*)
where
ASEQ bs AZERO r2 ⇝ *AZERO*
| *ASEQ bs r1 AZERO* ⇝ *AZERO*
| *ASEQ bs* (*AONE bs1*) *r* ⇝ *fuse* (*bs*@*bs1*) *r*
| *r1* ⇝ *r2* ⟹ *ASEQ bs r1 r3* ⇝ *ASEQ bs r2 r3*
| *r3* ⇝ *r4* ⟹ *ASEQ bs r1 r3* ⇝ *ASEQ bs r1 r4*
| *r* ⇝ *r′* ⟹ (*AALTs bs* (*rs1* @ [*r*] @ *rs2*)) ⇝ (*AALTs bs* (*rs1* @ [*r′*] @ *rs2*))

| *AALTs bs* (*rsa*@*AZERO* # *rsb*) ⇝ *AALTs bs* (*rsa*@*rsb*)
| *AALTs bs* (*rsa*@(*AALTs bs1 rs1*)# *rsb*) ⇝ *AALTs bs* (*rsa*@(*map* (*fuse bs1*) *rs1*)@*rsb*)

| *AALTs bs* (*map* (*fuse bs1*) *rs*) ⇝ *AALTs* (*bs*@*bs1*) *rs*

| *AALTs* (*bs*@*bs1*) *rs* ⇝ *AALTs bs* (*map* (*fuse bs1*) *rs*)
| *AALTs bs* [] ⇝ *AZERO*
| *AALTs bs* [*r*] ⇝ *fuse bs r*

| *erase a1 = erase a2* $\Longrightarrow$ *AALTs bs* (*rsa*@[*a1*]@*rsb*@[*a2*]@*rsc*) $\leadsto$ *AALTs bs* (*rsa*@[*a1*]@*rsb*@*rsc*)

inductive *rrewrites*:: *arexp* ⇒ *arexp* ⇒ *bool* (_ $\leadsto *$ _ [*100*, *100*] *100*)
 where
rs1[*intro*, *simp*]:*r* $\leadsto *$ *r*
| *rs2*[*intro*]: ⟦*r1* $\leadsto *$ *r2*; *r2* $\leadsto$ *r3*⟧ $\Longrightarrow$ *r1* $\leadsto *$ *r3*

inductive *srewrites*:: *arexp list* ⇒ *arexp list* ⇒ *bool* (_ *s*$\leadsto *$ _ [*100*, *100*] *100*)
 where
ss1: [] *s*$\leadsto *$ []
|*ss2*: ⟦*r* $\leadsto *$ *r'*; *rs s*$\leadsto *$ *rs'*⟧ $\Longrightarrow$ (*r*#*rs*) *s*$\leadsto *$ (*r'*#*rs'*)

lemma *r_in_rstar* : *r1* $\leadsto$ *r2* $\Longrightarrow$ *r1* $\leadsto *$ *r2*
 using *rrewrites.intros*(*1*) *rrewrites.intros*(*2*) **by** *blast*

lemma *real_trans*:
 assumes *a1*: *r1* $\leadsto *$ *r2* **and** *a2*: *r2* $\leadsto *$ *r3*
 shows *r1* $\leadsto *$ *r3*
 using *a2 a1*
 apply(*induct r2 r3 arbitrary*: *r1 rule*: *rrewrites.induct*)
 apply(*auto*)
 done

lemma *many_steps_later*: ⟦*r1* $\leadsto$ *r2*; *r2* $\leadsto *$ *r3* ⟧ $\Longrightarrow$ *r1* $\leadsto *$ *r3*
 by (*meson r_in_rstar real_trans*)

lemma *contextrewrites1*: *r* $\leadsto *$ *r'* $\Longrightarrow$ (*AALTs bs* (*r*#*rs*)) $\leadsto *$ (*AALTs bs* (*r'*#*rs*))
 apply(*induct r r' rule*: *rrewrites.induct*)
 apply *simp*
 by (*metis append_Cons append_Nil rrewrite.intros*(*6*) *rs2*)

lemma *contextrewrites2*: *r* $\leadsto *$ *r'* $\Longrightarrow$ (*AALTs bs* (*rs1*@[*r*]@*rs*)) $\leadsto *$ (*AALTs bs* (*rs1*@[*r'*]@*rs*))
 apply(*induct r r' rule*: *rrewrites.induct*)
 apply *simp*

using *rrewrite.intros*(*6*) **by** *blast*

lemma *srewrites__alt*: *rs1* $s\rightsquigarrow*$ *rs2* $\Longrightarrow$ (*AALTs bs* (*rs*@*rs1*)) $\rightsquigarrow*$ (*AALTs bs* (*rs*@*rs2*))

apply(*induct rs1 rs2 arbitrary*: *bs rs rule*: *srewrites.induct*)
apply(*rule rs1*)
apply(*drule__tac x* = *bs* **in** *meta__spec*)
apply(*drule__tac x* = *rsa*@[*r'*] **in** *meta__spec*)
apply *simp*
apply(*rule real__trans*)
prefer *2*
apply(*assumption*)
apply(*drule contextrewrites2*)
apply *auto*
done

corollary *srewrites__alt1*: *rs1* $s\rightsquigarrow*$ *rs2* $\Longrightarrow$ *AALTs bs rs1* $\rightsquigarrow*$ *AALTs bs rs2*
by (*metis append.left__neutral srewrites__alt*)

lemma *star__seq*: *r1* $\rightsquigarrow*$ *r2* $\Longrightarrow$ *ASEQ bs r1 r3* $\rightsquigarrow*$ *ASEQ bs r2 r3*
apply(*induct r1 r2 arbitrary*: *r3 rule*: *rrewrites.induct*)
apply(*rule rs1*)
apply(*erule rrewrites.cases*)
apply(*simp*)
apply(*rule r__in__rstar*)
apply(*rule rrewrite.intros*(*4*))
apply *simp*
apply(*rule rs2*)
apply(*assumption*)
apply(*rule rrewrite.intros*(*4*))
by *assumption*

lemma *star__seq2*: *r3* $\rightsquigarrow*$ *r4* $\Longrightarrow$ *ASEQ bs r1 r3* $\rightsquigarrow*$ *ASEQ bs r1 r4*
apply(*induct r3 r4 arbitrary*: *r1 rule*: *rrewrites.induct*)
apply *auto*
using *rrewrite.intros*(*5*) **by** *blast*

lemma *continuous__rewrite*: $[\![$*r1* $\rightsquigarrow*$ *AZERO*$]\!]$ $\Longrightarrow$ *ASEQ bs1 r1 r2* $\rightsquigarrow*$ *AZERO*

apply(*induction ra* $\stackrel{def}{=}$ *r1 rb* $\stackrel{def}{=}$ *AZERO arbitrary*: *bs1 r1 r2 rule*: *rrewrites.induct*)
apply (*simp add*: *r___in___rstar rrewrite.intros*(*1*))

by (*meson rrewrite.intros*(*1*) *rrewrites.intros*(*2*) *star___seq*)

lemma *bsimp___aalts___simpcases*: *AONE bs* ⇝∗ (*bsimp* (*AONE bs*)) *AZERO* ⇝∗ *bsimp AZERO ACHAR bs c* ⇝∗ (*bsimp* (*ACHAR bs c*))
apply (*simp add*: *rrewrites.intros*(*1*))
apply (*simp add*: *rrewrites.intros*(*1*))
by (*simp add*: *rrewrites.intros*(*1*))

lemma *trivialbsimpsrewrites*: ⟦⋀*x*. *x* ∈ *set rs* ⟹ *x* ⇝∗ *f x* ⟧ ⟹ *rs s*⇝∗ (*map f rs*)

apply(*induction rs*)
apply *simp*
apply(*rule ss1*)
by (*metis insert___iff list.simps*(*15*) *list.simps*(*9*) *srewrites.simps*)

lemma *bsimp___AALTsrewrites*: *AALTs bs1 rs* ⇝∗ *bsimp___AALTs bs1 rs*
apply(*induction rs*)
apply *simp*
apply(*rule r___in___rstar*)
apply(*simp add*: *rrewrite.intros*(*11*))
apply(*case___tac rs* = *Nil*)
apply(*simp*)
using *rrewrite.intros*(*12*) **apply** *auto*[*1*]
apply(*subgoal___tac length* (*a*#*rs*) > *1*)
apply(*simp add*: *bsimp___AALTs___qq*)
apply(*simp*)
done

inductive *frewrites*:: *arexp list* ⇒ *arexp list* ⇒ *bool* (___ *f*⇝∗ ___ [*100*, *100*] *100*)
where
fs1: [] *f*⇝∗ []
|*fs2*: ⟦*rs f*⇝∗ *rs*′⟧ ⟹ (*AZERO*#*rs*) *f*⇝∗ *rs*′
|*fs3*: ⟦*rs f*⇝∗ *rs*′⟧ ⟹ ((*AALTs bs rs1*) # *rs*) *f*⇝∗ ((*map* (*fuse bs*) *rs1*) @ *rs*′)
|*fs4*: ⟦*rs f*⇝∗ *rs*′;*nonalt r*; *nonazero r*⟧ ⟹ (*r*#*rs*) *f*⇝∗ (*r*#*rs*′)

```
lemma flts_prepend: ⟦nonalt a; nonazero a⟧ ⟹ flts (a#rs) = a # (flts rs)
  by (metis append_Cons append_Nil flts_single1 k00)

lemma fltsfrewrites: rs f↝* (flts rs)
  apply(induction rs)
  apply simp
   apply(rule fs1)

  apply(case_tac a = AZERO)


  using fs2 apply auto[1]
  apply(case_tac ∃bs rs. a = AALTs bs rs)
   apply(erule exE)+

   apply (simp add: fs3)
  apply(subst flts_prepend)
    apply(rule nonalt.elims(2))
  prefer 2
  thm nonalt.elims

        apply blast

  using bbbbs1 apply blast
       apply(simp add: nonalt.simps)+

   apply (meson nonazero.elims(3))

  by (meson fs4 nonalt.elims(3) nonazero.elims(3))


lemma rrewrite0away: AALTs bs ( AZERO # rsb) ↝ AALTs bs rsb
  by (metis append_Nil rrewrite.intros(7))


lemma frewritesaalts:rs f↝* rs' ⟹ (AALTs bs (rs1@rs)) ↝* (AALTs bs
(rs1@rs'))
  apply(induct rs rs' arbitrary: bs rs1 rule:frewrites.induct)
    apply(rule rs1)
    apply(drule_tac x = bs in meta_spec)
  apply(drule_tac x = rs1 @ [AZERO] in meta_spec)
    apply(rule real_trans)
     apply simp
```

```
  using r__in__rstar rrewrite.intros(7) apply presburger
    apply(drule__tac x = bsa in meta__spec)
  apply(drule__tac x = rs1a @ [AALTs bs rs1] in meta__spec)
   apply(rule real__trans)
    apply simp
  using r__in__rstar rrewrite.intros(8) apply presburger
    apply(drule__tac x = bs in meta__spec)
  apply(drule__tac x = rs1@[r] in meta__spec)
    apply(rule real__trans)
   apply simp
  apply auto
  done

lemma fltsrewrites:   AALTs bs1 rs ⤳* AALTs bs1 (flts rs)
  apply(induction rs)
   apply simp
  apply(case__tac a = AZERO)
 apply (metis append__Nil flts.simps(2) many__steps__later rrewrite.intros(7))



  apply(case__tac ∃ bs2 rs2. a = AALTs bs2 rs2)
   apply(erule exE)+
   apply(simp add: flts.simps)
   prefer 2

  apply(subst flts__prepend)

      apply (meson nonalt.elims(3))

    apply (meson nonazero.elims(3))
   apply(subgoal__tac (a#rs) f⤳* (a#flts rs))
  apply (metis append__Nil frewritesaalts)
  apply (meson fltsfrewrites fs4 nonalt.elims(3) nonazero.elims(3))
  by (metis append__Cons append__Nil fltsfrewrites frewritesaalts k00 k0a)

lemma alts__simpalts: ⋀bs1 rs. (⋀x. x ∈ set rs ⟹ x ⤳* bsimp x) ⟹
AALTs bs1 rs ⤳* AALTs bs1 (map bsimp rs)
  apply(subgoal__tac rs s⤳* (map bsimp rs))
   prefer 2
  using trivialbsimpsrewrites apply auto[1]
  using srewrites__alt1 by auto


lemma threelistsappend: rsa@a#rsb = (rsa@[a])@rsb
```

```
  apply auto
  done

fun distinctByAcc :: 'a list ⇒ ('a ⇒ 'b) ⇒ 'b set ⇒ 'b set
  where
  distinctByAcc [] f acc = acc
| distinctByAcc (x#xs) f acc =
     (if (f x) ∈ acc then distinctByAcc xs f acc
      else (distinctByAcc xs f ({f x} ∪ acc)))

lemma dB_single_step: distinctBy (a#rs) f {} = a # distinctBy rs f {f a}
  apply simp
  done

lemma somewhereInside: r ∈ set rs ⟹ ∃ rs1 rs2. rs = rs1@[r]@rs2
  using split_list by fastforce

lemma somewhereMapInside: f r ∈ f ` set rs ⟹ ∃ rs1 rs2 a. rs = rs1@[a]@rs2
∧ f a = f r
  apply auto
  by (metis split_list)

lemma alts_dBrewrites_withFront: AALTs bs (rsa @ rs) ⇝* AALTs bs
(rsa @ distinctBy rs erase (erase ` set rsa))
  apply(induction rs arbitrary: rsa)
   apply simp
  apply(drule_tac x = rsa@[a] in meta_spec)
  apply(subst threelistsappend)
  apply(rule real_trans)
  apply simp
  apply(case_tac a ∈ set rsa)
   apply simp
   apply(drule somewhereInside)
   apply(erule exE)+
   apply simp
  apply(subgoal_tac AALTs bs
            (rs1 @
             a #
             rs2 @
             a #
             distinctBy rs erase
              (insert (erase a)
                (erase `
                  (set rs1 ∪ set rs2)))) ⇝ AALTs bs (rs1@ a # rs2 @ distinctBy
rs erase
```

```
            (insert (erase a)
              (erase `
               (set rs1 ∪ set rs2)))) )
  prefer 2
  using rrewrite.intros(13) apply force
  using r_in_rstar apply force
   apply(subgoal_tac erase ` set (rsa @ [a]) = insert (erase a) (erase ` set
rsa))
  prefer 2

   apply auto[1]
  apply(case_tac erase a ∈ erase `set rsa)

   apply simp
  apply(subgoal_tac AALTs bs (rsa @ a # distinctBy rs erase (insert (erase
a) (erase ` set rsa))) ⇝
                AALTs bs (rsa @ distinctBy rs erase (insert (erase a) (erase `
set rsa))))
  apply force
 apply (smt (verit, ccfv_threshold) append_Cons append_assoc append_self_conv2
r_in_rstar rrewrite.intros(13) same_append_eq somewhereMapInside)
  by force



lemma alts_dBrewrites: AALTs bs rs ⇝* AALTs bs (distinctBy rs erase {})
  apply(induction rs)
   apply simp
  apply simp
  using alts_dBrewrites_withFront
  by (metis append_Nil dB_single_step empty_set image_empty)




lemma bsimp_rewrite: (rrewrites r ( bsimp r))
  apply(induction r rule: bsimp.induct)
       apply simp
       apply(case_tac bsimp r1 = AZERO)
        apply simp
  using continuous_rewrite apply blast
       apply(case_tac ∃ bs. bsimp r1 = AONE bs)
        apply(erule exE)
```

```
        apply simp
        apply(subst bsimp_ASEQ2)
      apply (meson real_trans rrewrite.intros(3) rrewrites.intros(2) star_seq
star_seq2)
        apply (smt (verit, best) bsimp_ASEQ0 bsimp_ASEQ1 real_trans
rrewrite.intros(2) rs2 star_seq star_seq2)
      defer
  using bsimp_aalts_simpcases(2) apply blast
  apply simp
  apply simp
  apply simp

  apply auto


  apply(subgoal_tac AALTs bs1 rs ⇝* AALTs bs1 (map bsimp rs))
   apply(subgoal_tac AALTs bs1 (map bsimp rs) ⇝* AALTs bs1 (flts (map
bsimp rs)))
 apply(subgoal_tac AALTs bs1 (flts (map bsimp rs)) ⇝* AALTs bs1 (distinctBy
(flts (map bsimp rs)) erase {}))
    apply(subgoal_tac AALTs bs1 (distinctBy (flts (map bsimp rs)) erase {})
⇝* bsimp_AALTs bs1 (distinctBy (flts (map bsimp rs)) erase {} ))


      apply (meson real_trans)

   apply (meson bsimp_AALTsrewrites)

  apply (meson alts_dBrewrites)

  using fltsrewrites apply auto[1]

  using alts_simpalts by force


lemma rewritenullable: ⟦r1 ⇝ r2; bnullable r1 ⟧ ⟹ bnullable r2
  apply(induction r1 r2 rule: rrewrite.induct)
             apply(simp)+
  apply (metis bnullable_correctness erase_fuse)
          apply simp
         apply simp
        apply auto[1]
       apply auto[1]
      apply auto[4]
     apply (metis UnCI bnullable_correctness erase_fuse imageI)
```

```
    apply (metis bnullable_correctness erase_fuse)
    apply (metis bnullable_correctness erase_fuse)

   apply (metis bnullable_correctness erase.simps(5) erase_fuse)


 by (smt (z3) Un_iff bnullable_correctness insert_iff list.set(2) qq3 set_append)

lemma rewrite_non_nullable: ⟦r1 ⇝ r2; ¬bnullable r1 ⟧ ⟹ ¬bnullable r2
  apply(induction r1 r2 rule: rrewrite.induct)
           apply auto
      apply (metis bnullable_correctness erase_fuse)+
  done


lemma rewritesnullable: ⟦ r1 ⇝* r2; bnullable r1 ⟧ ⟹ bnullable r2
  apply(induction r1 r2 rule: rrewrites.induct)
   apply simp
  apply(rule rewritenullable)
   apply simp
  apply simp
  done

lemma nonbnullable_lists_concat: ⟦ ¬ (∃ r0∈ set rs1. bnullable r0); ¬ bnul-
lable r; ¬ (∃ r0∈ set rs2. bnullable r0)⟧ ⟹
¬(∃ r0 ∈ (set (rs1@[r]@rs2)). bnullable r0 )
  apply simp
  apply blast
  done



lemma nomember_bnullable: ⟦ ¬ (∃ r0∈ set rs1. bnullable r0); ¬ bnullable r;
¬ (∃ r0∈ set rs2. bnullable r0)⟧
 ⟹ ¬bnullable (AALTs bs (rs1 @ [r] @ rs2))
  using nonbnullable_lists_concat qq3 by presburger

lemma bnullable_segment: bnullable (AALTs bs (rs1@[r]@rs2)) ⟹ bnullable
(AALTs bs rs1) ∨ bnullable (AALTs bs rs2) ∨ bnullable r
  apply(case_tac ∃ r0∈ set rs1.  bnullable r0)

  using qq3 apply blast
  apply(case_tac bnullable r)

  apply blast
```

```
  apply(case_tac ∃ r0 ∈ set rs2.  bnullable r0)

  using bnullable.simps(4) apply presburger
  apply(subgoal_tac False)

  apply blast

  using nomember_bnullable by blast



lemma bnullablewhichbmkeps: ⟦bnullable  (AALTs bs (rs1@[r]@rs2)); ¬ bnullable (AALTs bs rs1); bnullable r ⟧
 ⟹ bmkeps (AALTs bs (rs1@[r]@rs2)) = bs @ (bmkeps r)
  using qq2 bnullable_Hdbmkeps_Hd by force

lemma rrewrite_nbnullable: ⟦ r1 ↝ r2 ; ¬ bnullable r1 ⟧ ⟹ ¬bnullable r2
  apply(induction rule: rrewrite.induct)
             apply auto[1]
            apply auto[1]
           apply auto[1]
           apply (metis bnullable_correctness erase_fuse)
          apply auto[1]
         apply auto[1]
        apply auto[1]
       apply auto[1]
      apply auto[1]
      apply (metis bnullable_correctness erase_fuse)
     apply auto[1]
     apply (metis bnullable_correctness erase_fuse)
    apply auto[1]
    apply (metis bnullable_correctness erase_fuse)
   apply auto[1]
   apply auto[1]

  apply (metis bnullable_correctness erase_fuse)

  by (meson rewrite_non_nullable rrewrite.intros(13))




lemma spillbmkepslistr: bnullable (AALTs bs1 rs1)
    ⟹ bmkeps (AALTs bs (AALTs bs1 rs1 # rsb)) = bmkeps (AALTs bs ( map (fuse bs1) rs1 @ rsb))
```

apply(*subst bnullable___Hdbmkeps___Hd*)

apply *simp*
by (*metis bmkeps.simps*(*3*) *k0a list.set___intros*(*1*) *qq1 qq4 qs3*)

lemma *third___segment___bnullable*: ⟦*bnullable* (*AALTs bs* (*rs1*@*rs2*@*rs3*)); ¬*bnullable* (*AALTs bs rs1*); ¬*bnullable* (*AALTs bs rs2*)⟧ ⟹
bnullable (*AALTs bs rs3*)

by (*metis append.left___neutral append___Cons bnullable.simps*(*1*) *bnullable___segment rrewrite.intros*(*7*) *rrewrite___nbnullable*)

lemma *third___segment___bmkeps*: ⟦*bnullable* (*AALTs bs* (*rs1*@*rs2*@*rs3*)); ¬*bnullable* (*AALTs bs rs1*); ¬*bnullable* (*AALTs bs rs2*)⟧ ⟹
bmkeps (*AALTs bs* (*rs1*@*rs2*@*rs3*)) = *bmkeps* (*AALTs bs rs3*)
apply(*subgoal___tac bnullable* (*AALTs bs rs3*))
apply(*subgoal___tac* ∀ *r* ∈ *set* (*rs1*@*rs2*). ¬*bnullable r*)
apply(*subgoal___tac bmkeps* (*AALTs bs* (*rs1*@*rs2*@*rs3*)) = *bmkeps* (*AALTs bs* ((*rs1*@*rs2*)@*rs3*)))
apply (*metis qq2 qq3*)

apply (*metis append.assoc*)

apply (*metis append.assoc in___set___conv___decomp r2 third___segment___bnullable*)

using *third___segment___bnullable* **by** *blast*

lemma *rewrite___bmkepsalt*: ⟦*bnullable* (*AALTs bs* (*rsa* @ *AALTs bs1 rs1* # *rsb*)); *bnullable* (*AALTs bs* (*rsa* @ *map* (*fuse bs1*) *rs1* @ *rsb*))⟧
⟹ *bmkeps* (*AALTs bs* (*rsa* @ *AALTs bs1 rs1* # *rsb*)) = *bmkeps* (*AALTs bs* (*rsa* @ *map* (*fuse bs1*) *rs1* @ *rsb*))
apply(*case___tac bnullable* (*AALTs bs rsa*))

using *qq1* **apply** *force*
apply(*case___tac bnullable* (*AALTs bs1 rs1*))
apply(*subst qq2*)

using *r2* **apply** *blast*

apply (*metis list.set___intros*(*1*))
apply (*smt* (*verit, ccfv___threshold*) *append___eq___append___conv2 list.set___intros*(*1*) *qq2 qq3 rewritenullable rrewrite.intros*(*8*) *self___append___conv2 spillbmkepslistr*)

```
 thm qq1
  apply(subgoal_tac bmkeps  (AALTs bs (rsa @ AALTs bs1 rs1 # rsb)) =
bmkeps (AALTs bs rsb) )
   prefer 2

 apply (metis append_Cons append_Nil bnullable.simps(1) bnullable_segment
rewritenullable rrewrite.intros(11) third_segment_bmkeps)

 by (metis bnullable.simps(4) rewrite_non_nullable rrewrite.intros(10) third_segment_bmkeps)



lemma rewrite_bmkeps: ⟦ r1 ⇝ r2; (bnullable r1)⟧ ⟹ bmkeps r1 = bmkeps
r2

  apply(frule rewritenullable)
  apply simp
  apply(induction r1 r2 rule: rrewrite.induct)
             apply simp
  using bnullable.simps(1) bnullable.simps(5) apply blast
        apply (simp add: b2)
       apply simp
        apply simp
  apply(frule bnullable_segment)
       apply(case_tac bnullable (AALTs bs rs1))
  using qq1 apply force
       apply(case_tac bnullable r)
  using bnullablewhichbmkeps rewritenullable apply presburger
       apply(subgoal_tac bnullable (AALTs bs rs2))
  apply(subgoal_tac ¬ bnullable r')
  apply (simp add: qq2 r1)

  using rrewrite_nbnullable apply blast

       apply blast
      apply (simp add: flts_append qs3)

  apply (meson rewrite_bmkepsalt)

  using bnullable.simps(4) q3a apply blast

  apply (simp add: q3a)
```

using *bnullable.simps*(*1*) **apply** *blast*

apply (*simp add*: *b2*)

by (*smt* (*z3*) *Un___iff bnullable___correctness erase.simps*(*5*) *qq1 qq2 qq3 set___append*)

lemma *rewrites___bmkeps*: ⟦ (*r1* ⇝* *r2*); (*bnullable r1*)⟧ ⟹ *bmkeps r1* = *bmkeps r2*
apply(*induction r1 r2 rule*: *rrewrites.induct*)
apply *simp*
apply(*subgoal___tac bnullable r2*)
prefer *2*
apply(*metis rewritesnullable*)
apply(*subgoal___tac bmkeps r1* = *bmkeps r2*)
prefer *2*
apply *fastforce*
using *rewrite___bmkeps* **by** *presburger*

thm *rrewrite.intros*(*12*)
lemma *alts___rewrite___front*: *r* ⇝ *r′* ⟹ *AALTs bs* (*r* # *rs*) ⇝ *AALTs bs* (*r′* # *rs*)
by (*metis append___Cons append___Nil rrewrite.intros*(*6*))

lemma *alt___rewrite___front*: *r* ⇝ *r′* ⟹ *AALT bs r r2* ⇝ *AALT bs r′ r2*
using *alts___rewrite___front* **by** *blast*

lemma *to___zero___in___alt*: *AALT bs* (*ASEQ* [] *AZERO r*) *r2* ⇝ *AALT bs AZERO r2*
by (*simp add*: *alts___rewrite___front rrewrite.intros*(*1*))

lemma *alt___remove0___front*: *AALT bs AZERO r* ⇝ *AALTs bs* [*r*]
by (*simp add*: *rrewrite0away*)

lemma *alt___rewrites___back*: *r2* ⇝* *r2′* ⟹*AALT bs r1 r2* ⇝* *AALT bs r1 r2′*
apply(*induction r2 r2′ arbitrary*: *bs rule*: *rrewrites.induct*)
apply *simp*
by (*meson rs1 rs2 srewrites___alt1 ss1 ss2*)

lemma *rewrite___fuse*: *r2* ⇝ *r3* ⟹ *fuse bs r2* ⇝* *fuse bs r3*
apply(*induction r2 r3 arbitrary*: *bs rule*: *rrewrite.induct*)

```
        apply auto
       apply (simp add: continuous_rewrite)
      apply (simp add: r_in_rstar rrewrite.intros(2))
     apply (metis fuse_append r_in_rstar rrewrite.intros(3))
  using r_in_rstar star_seq apply blast
  using r_in_rstar star_seq2 apply blast
  using contextrewrites2 r_in_rstar apply auto[1]
    apply (simp add: r_in_rstar rrewrite.intros(7))
  using rrewrite.intros(8) apply auto[1]
   apply (metis append_assoc r_in_rstar rrewrite.intros(9))
  apply (metis append_assoc r_in_rstar rrewrite.intros(10))
  apply (simp add: r_in_rstar rrewrite.intros(11))
  apply (metis fuse_append r_in_rstar rrewrite.intros(12))
  using rrewrite.intros(13) by auto


lemma rewrites_fuse: r2 ↝* r2' ⟹ (fuse bs1 r2) ↝* (fuse bs1 r2')
  apply(induction r2 r2' arbitrary: bs1 rule: rrewrites.induct)
   apply simp
  by (meson real_trans rewrite_fuse)

lemma bder_fuse_list: map (bder c ∘ fuse bs1) rs1 = map (fuse bs1 ∘ bder
c) rs1
  apply(induction rs1)
  apply simp
  by (simp add: bder_fuse)


lemma rewrite_der_altmiddle: bder c (AALTs bs (rsa @ AALTs bs1 rs1 #
rsb)) ↝* bder c (AALTs bs (rsa @ map (fuse bs1) rs1 @ rsb))
```

```
  apply simp
  apply(simp add: bder__fuse__list)
 apply(rule many__steps__later)
  apply(subst rrewrite.intros(8))
  apply simp

 by fastforce

lemma lock__step__der__removal:
 shows  erase a1 = erase a2 ⟹
                       bder c (AALTs bs (rsa @ [a1] @ rsb @ [a2] @ rsc))
⇝*
                       bder c (AALTs bs (rsa @ [a1] @ rsb @ rsc))
 apply(simp)

 using rrewrite.intros(13) by auto

lemma rewrite__after__der: r1 ⇝ r2 ⟹ (bder c r1) ⇝* (bder c r2)
 apply(induction r1 r2 arbitrary: c rule: rrewrite.induct)

        apply (simp add: r__in__rstar rrewrite.intros(1))
 apply simp

 apply (meson contextrewrites1 r__in__rstar rrewrite.intros(11) rrewrite.intros(2)
rrewrite0away rs2)
     apply(simp)
     apply(rule many__steps__later)
      apply(rule to__zero__in__alt)
     apply(rule many__steps__later)
 apply(rule alt__remove0__front)
     apply(rule many__steps__later)
      apply(rule rrewrite.intros(12))
 using bder__fuse fuse__append rs1 apply presburger
    apply(case__tac bnullable r1)
 prefer 2
     apply(subgoal__tac ¬bnullable r2)
      prefer 2
 using rewrite__non__nullable apply presburger
     apply simp+

 using star__seq apply auto[1]
    apply(subgoal__tac bnullable r2)
     apply simp+
 apply(subgoal__tac bmkeps r1 = bmkeps r2)
 prefer 2
```

using *rewrite___bmkeps* **apply** *auto*[*1*]
using *contextrewrites1 star___seq* **apply** *auto*[*1*]
using *rewritenullable* **apply** *auto*[*1*]
apply(*case___tac bnullable r1*)
apply *simp*
apply(*subgoal___tac ASEQ* [] (*bder c r1*) *r3* ⇝ *ASEQ* [] (*bder c r1*) *r4*)
prefer *2*
using *rrewrite.intros*(*5*) **apply** *blast*
apply(*rule many___steps___later*)
apply(*rule alt___rewrite___front*)
apply *assumption*
apply (*meson alt___rewrites___back rewrites___fuse*)

apply (*simp add*: *r___in___rstar rrewrite.intros*(*5*))

using *contextrewrites2* **apply** *force*

using *rrewrite.intros*(*7*) **apply** *force*

using *rewrite___der___altmiddle* **apply** *auto*[*1*]

apply (*metis bder.simps*(*4*) *bder___fuse___list map___map r___in___rstar rrewrite.intros*(*9*))

apply (*metis List.map.compositionality bder.simps*(*4*) *bder___fuse___list r___in___rstar rrewrite.intros*(*10*))

apply (*simp add*: *r___in___rstar rrewrite.intros*(*11*))

apply (*metis bder.simps*(*4*) *bder___bsimp___AALTs bsimp___AALTs.simps*(*2*) *bsimp___AALTsrewrites*)

using *lock___step___der___removal* **by** *auto*

lemma *rewrites___after___der*: *r1* ⇝∗ *r2* ⟹ (*bder c r1*) ⇝∗ (*bder c r2*)
apply(*induction r1 r2 rule*: *rrewrites.induct*)
apply(*rule rs1*)
by (*meson real___trans rewrite___after___der*)

lemma *central*: (*bders r s*) ⇝∗ (*bders___simp r s*)

apply(*induct s arbitrary*: *r rule*: *rev___induct*)

apply *simp*
apply(*subst bders___append*)
apply(*subst bders___simp___append*)
by (*metis bders.simps*(*1*) *bders.simps*(*2*) *bders___simp.simps*(*1*) *bders___simp.simps*(*2*) *bsimp___rewrite real___trans rewrites___after___der*)

thm *arexp.induct*

lemma *quasi___main*: *bnullable* (*bders r s*) $\Longrightarrow$ *bmkeps* (*bders r s*) = *bmkeps* (*bders___simp r s*)
using *central rewrites___bmkeps* **by** *blast*

theorem *main___main*: *blexer r s* = *blexer___simp r s*
by (*simp add*: *b4 blexer___def blexer___simp___def quasi___main*)

theorem *blexersimp___correctness*: *blexer___simp r s*= *lexer r s*
using *blexer___correctness main___main* **by** *auto*

unused-thms

end

3 Introduction

This works builds on previous work by Ausaf and Urban using regular expression'd bit-coded derivatives to do lexing that is both fast and satisfied the POSIX specification. In their work, a bit-coded algorithm introduced by Sulzmann and Lu was formally verified in Isabelle, by a very clever use of flex function and retrieve to carefully mimic the way a value is built up by the injection funciton.

In the previous work, Ausaf and Urban established the below equality:

Lemma 1. *If* $v : (r^{\downarrow})\backslash c$ *then* *retrieve* $(r\backslash\!\!\backslash c)\ v$ = *retrieve* r (*inj* $(r^{\downarrow})\ c\ v$).

This lemma links the derivative of a bit-coded regular expression with the regular expression itself before the derivative.

Brzozowski [**?**] introduced the notion of the *derivative* $r\backslash c$ of a regular expression r w.r.t. a character c, and showed that it gave a simple solution to the problem of matching a string s with a regular expression r: if the derivative of r w.r.t. (in succession) all the characters of the string matches the empty string, then r matches s (and *vice versa*). The derivative has the property (which may almost be regarded as its specification) that, for every string s and regular expression r and character c, one has $cs \in L(r)$ if and only if $s \in L(r\backslash c)$. The beauty of Brzozowski's derivatives is that they are neatly expressible in any functional language, and easily definable and reasoned about in theorem provers—the definitions just consist of inductive datatypes and simple recursive functions. A mechanised correctness proof of Brzozowski's matcher in for example HOL4 has been mentioned by Owens and Slind [**?**]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [**?**]. And another one in Coq is given by Coquand and Siles [**?**].

If a regular expression matches a string, then in general there is more than one way of how the string is matched. There are two commonly used disambiguation strategies to generate a unique answer: one is called GREEDY matching [**?**] and the other is POSIX matching [**?**,**?**,**?**,**?**,**?**]. For example consider the string xy and the regular expression $(x + y + xy)^\star$. Either the string can be matched in two 'iterations' by the single letter-regular expressions x and y, or directly in one iteration by xy. The first case corresponds to GREEDY matching, which first matches with the left-most symbol and only matches the next symbol in case of a mismatch (this is greedy in the sense of preferring instant gratification to delayed repletion). The second case is POSIX matching, which prefers the longest match.

In the context of lexing, where an input string needs to be split up into a sequence of tokens, POSIX is the more natural disambiguation strategy for what programmers consider basic syntactic building blocks in their programs. These building blocks are often specified by some regular expressions, say r_{key} and r_{id} for recognising keywords and identifiers, respectively. There are a few underlying (informal) rules behind tokenising a string in a POSIX [**?**] fashion:

- *The Longest Match Rule* (or *"Maximal Munch Rule"*): The longest initial substring matched by any regular expression is taken as next token.
- *Priority Rule:* For a particular longest initial substring, the first (leftmost) regular expression that can match determines the token.
- *Star Rule:* A subexpression repeated by * shall not match an empty string unless this is the only match for the repetition.
- *Empty String Rule:* An empty string shall be considered to be longer than no match at all.

Consider for example a regular expression r_{key} for recognising keywords such as *if*, *then* and so on; and r_{id} recognising identifiers (say, a single character followed by characters or numbers). Then we can form the regular expression $(r_{key} + r_{id})^\star$ and use POSIX matching to tokenise strings, say *iffoo* and *if*. For *iffoo* we obtain by the Longest Match Rule a single identifier token, not a keyword followed by an identifier. For *if* we obtain by the Priority Rule a keyword token, not an identifier token—even if r_{id} matches also. By the Star Rule we know $(r_{key} + r_{id})^\star$ matches *iffoo*, respectively *if*, in exactly one 'iteration' of the star.

The Empty String Rule is for cases where, for example, the regular expression $(a^\star)^\star$ matches against the string *bc*. Then the longest initial matched substring is the empty string, which is matched by both the whole regular expression and the parenthesised subexpression.

One limitation of Brzozowski's matcher is that it only generates a YES/NO answer for whether a string is being matched by a regular expression. Sulzmann and Lu [**?**] extended this matcher to allow generation not just of a YES/NO answer but of an actual matching, called a [lexical] *value*. Assuming a regular expression matches a string, values encode the information of *how* the string is matched by the regular expression—that is, which part of the string is matched by which part of the regular expression. For this consider again the string *xy* and the regular expression $(x + (y + xy))^\star$ (this time fully parenthesised). We can view this regular expression as tree and if the string *xy* is matched by two Star 'iterations', then the *x* is matched by the left-most alternative in this tree and the *y* by the right-left alternative. This suggests to record this matching as

$$\mathit{Stars}\ [\mathit{Left}\ (\mathit{Char}\ x),\ \mathit{Right}\ (\mathit{Left}\ (\mathit{Char}\ y))]$$

where *Stars*, *Left*, *Right* and *Char* are constructors for values. *Stars* records how many iterations were used; *Left*, respectively *Right*, which alternative is used. This 'tree view' leads naturally to the idea that regular expressions act as types and values as inhabiting those types (see, for example, [**?**]). The value for matching *xy* in a single 'iteration', i.e. the POSIX value, would look as follows

$$\mathit{Stars}\ [\mathit{Seq}\ (\mathit{Char}\ x)\ (\mathit{Char}\ y)]$$

where *Stars* has only a single-element list for the single iteration and *Seq* indicates that *xy* is matched by a sequence regular expression.

Sulzmann and Lu give a simple algorithm to calculate a value that appears to be the value associated with POSIX matching. The challenge then is to specify that value, in an algorithm-independent fashion, and to show that Sulzmann and Lu's derivative-based algorithm does indeed calculate a value that is correct according to the specification. The answer given by Sulzmann and Lu [**?**] is to define a relation (called an "order relation") on the set of values of r, and to show that (once a string to be matched is chosen) there is a maximum element and that it is computed by their derivative-based algorithm. This proof idea is inspired by work of Frisch and Cardelli [**?**] on a GREEDY regular expression matching algorithm. However, we were not able to establish transitivity and totality for the "order relation" by Sulzmann and Lu. There are some inherent problems with their approach (of which some of the proofs are not published in [**?**]); perhaps more importantly, we give in this paper a simple inductive (and algorithm-independent) definition of what we call being a *POSIX value* for a regular expression r and a string s; we show that the algorithm by Sulzmann and Lu computes such a value and that such a value is unique. Our proofs are both done by hand and checked in Isabelle/HOL. The experience of doing our proofs has been that this mechanical checking was absolutely essential: this subject area has hidden snares. This was also noted by Kuklewicz [**?**] who found that nearly all POSIX matching implementations are “buggy” [**?**, Page 203] and by Grathwohl et al [**?**, Page 36] who wrote:

> *“The POSIX strategy is more complicated than the greedy because of the dependence on information about the length of matched strings in the various subexpressions.”*

Contributions: We have implemented in Isabelle/HOL the derivative-based regular expression matching algorithm of Sulzmann and Lu [**?**]. We have proved the correctness of this algorithm according to our specification of what a POSIX value is (inspired by work of Vansummeren [**?**]). Sulzmann and Lu sketch in [**?**] an informal correctness proof: but to us it contains unfillable gaps.[4] Our specification of a POSIX value consists of a simple inductive definition that given a string and a regular expression uniquely determines this value. We also show that our definition is equivalent to an ordering of values based on positions by Okui and Suzuki [**?**].

We extend our results to ??? Bitcoded version??

4 Preliminaries

Strings in Isabelle/HOL are lists of characters with the empty string being represented by the empty list, written [], and list-cons being written as _ :: _. Often we use the usual bracket notation for lists also for strings; for example a string consisting of just a single character c is written $[c]$. We use the usual definitions for *prefixes* and *strict prefixes* of strings. By using the type *char* for characters we have a supply of finitely many characters roughly corresponding to the ASCII character set. Regular expressions are defined as usual as the elements of the following inductive datatype:

$$r := \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^\star$$

where $\mathbf{0}$ stands for the regular expression that does not match any string, $\mathbf{1}$ for the regular expression that matches only the empty string and c for matching a character literal. The language of a regular expression is also defined as usual by the recursive function L with the six clauses:

$$\begin{array}{lrcl}
(1) & L(\mathbf{0}) & \stackrel{\text{def}}{=} & \varnothing \\
(2) & L(\mathbf{1}) & \stackrel{\text{def}}{=} & \{[]\} \\
(3) & L(c) & \stackrel{\text{def}}{=} & \{[c]\} \\
(4) & L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L(r_1) \mathbin{@} L(r_2) \\
(5) & L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) \\
(6) & L(r^\star) & \stackrel{\text{def}}{=} & (L(r))\star
\end{array}$$

[4] An extended version of [**?**] is available at the website of its first author; this extended version already includes remarks in the appendix that their informal proof contains gaps, and possible fixes are not fully worked out.

In clause *(4)* we use the operation _ @ _ for the concatenation of two languages (it is also list-append for strings). We use the star-notation for regular expressions and for languages (in the last clause above). The star for languages is defined inductively by two clauses: (*i*) the empty string being in the star of a language and (*ii*) if s_1 is in a language and s_2 in the star of this language, then also s_1 @ s_2 is in the star of this language. It will also be convenient to use the following notion of a *semantic derivative* (or *left quotient*) of a language defined as

$$\mathit{Der}\ c\ A \stackrel{\mathit{def}}{=} \{s \mid c :: s \in A\}\ .$$

For semantic derivatives we have the following equations (for example mechanically proved in [**?**]):

$$\begin{array}{lcl}
\mathit{Der}\ c\ \varnothing & \stackrel{\text{def}}{=} & \varnothing \\
\mathit{Der}\ c\ \{[]\} & \stackrel{\text{def}}{=} & \varnothing \\
\mathit{Der}\ c\ \{[d]\} & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \{[]\}\ \mathit{else}\ \varnothing \\
\mathit{Der}\ c\ (A \cup B) & \stackrel{\text{def}}{=} & \mathit{Der}\ c\ A \cup \mathit{Der}\ c\ B \\
\mathit{Der}\ c\ (A\ @\ B) & \stackrel{\text{def}}{=} & (\mathit{Der}\ c\ A\ @\ B) \cup (\mathit{if}\ [] \in A\ \mathit{then}\ \mathit{Der}\ c\ B\ \mathit{else}\ \varnothing) \\
\mathit{Der}\ c\ (A\star) & \stackrel{\text{def}}{=} & \mathit{Der}\ c\ A\ @\ A\star
\end{array} \tag{1}$$

Brzozowski's derivatives of regular expressions [**?**] can be easily defined by two recursive functions: the first is from regular expressions to booleans (implementing a test when a regular expression can match the empty string), and the second takes a regular expression and a character to a (derivative) regular expression:

$$\begin{array}{lcl}
\mathit{nullable}\ (\mathbf{0}) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{nullable}\ (\mathbf{1}) & \stackrel{\text{def}}{=} & \mathit{True} \\
\mathit{nullable}\ (c) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{nullable}\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}\ r_1 \vee \mathit{nullable}\ r_2 \\
\mathit{nullable}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}\ r_1 \wedge \mathit{nullable}\ r_2 \\
\mathit{nullable}\ (r^\star) & \stackrel{\text{def}}{=} & \mathit{True}
\end{array}$$

$$\begin{array}{lcl}
\mathbf{0}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathbf{1}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
d\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathbf{1}\ \mathit{else}\ \mathbf{0} \\
(r_1 + r_2)\backslash c & \stackrel{\text{def}}{=} & (r_1\backslash c) + (r_2\backslash c) \\
(r_1 \cdot r_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}\ r_1\ \mathit{then}\ (r_1\backslash c) \cdot r_2 + (r_2\backslash c)\ \mathit{else}\ (r_1\backslash c) \cdot r_2 \\
(r^\star)\backslash c & \stackrel{\text{def}}{=} & (r\backslash c) \cdot r^\star
\end{array}$$

We may extend this definition to give derivatives w.r.t. strings:

$$\begin{array}{lcl}
r\backslash[] & \stackrel{\text{def}}{=} & r \\
r\backslash(c :: s) & \stackrel{\text{def}}{=} & (r\backslash c)\backslash s
\end{array}$$

Given the equations in (1), it is a relatively easy exercise in mechanical reasoning to establish that

Proposition 1.
(1) nullable r if and only if $[] \in L(r)$, *and*
(2) $L(r\backslash c) = Der\ c\ (L(r))$.

With this in place it is also very routine to prove that the regular expression matcher defined as

$$match\ r\ s \stackrel{def}{=} nullable\ (r\backslash s)$$

gives a positive answer if and only if $s \in L(r)$. Consequently, this regular expression matching algorithm satisfies the usual specification for regular expression matching. While the matcher above calculates a provably correct YES/NO answer for whether a regular expression matches a string or not, the novel idea of Sulzmann and Lu [**?**] is to append another phase to this algorithm in order to calculate a [lexical] value. We will explain the details next.

5 POSIX Regular Expression Matching

There have been many previous works that use values for encoding *how* a regular expression matches a string. The clever idea by Sulzmann and Lu [**?**] is to define a function on values that mirrors (but inverts) the construction of the derivative on regular expressions. *Values* are defined as the inductive datatype

$$v := Empty \mid Char\ c \mid Left\ v \mid Right\ v \mid Seq\ v_1\ v_2 \mid Stars\ vs$$

where we use *vs* to stand for a list of values. (This is similar to the approach taken by Frisch and Cardelli for GREEDY matching [**?**], and Sulzmann and Lu for POSIX matching [**?**]). The string underlying a value can be calculated by the *flat* function, written |_| and defined as:

$$\begin{array}{lcl@{\qquad}lcl}
|Empty| & \stackrel{\text{def}}{=} & [] & |Seq\ v_1\ v_2| & \stackrel{\text{def}}{=} & |v_1|\ @\ |v_2| \\
|Char\ c| & \stackrel{\text{def}}{=} & [c] & |Stars\ []| & \stackrel{\text{def}}{=} & [] \\
|Left\ v| & \stackrel{\text{def}}{=} & |v| & |Stars\ (v :: vs)| & \stackrel{\text{def}}{=} & |v|\ @\ |Stars\ vs| \\
|Right\ v| & \stackrel{\text{def}}{=} & |v| & & &
\end{array}$$

We will sometimes refer to the underlying string of a value as *flattened value*. We will also overload our notation and use $|vs|$ for flattening a list of values and concatenating the resulting strings.

Sulzmann and Lu define inductively an *inhabitation relation* that associates values to regular expressions. We define this relation as follows:[5]

[5] Note that the rule for *Stars* differs from our earlier paper [**?**]. There we used the original definition by Sulzmann and Lu which does not require that the values $v \in vs$ flatten to a non-empty string. The reason for introducing the more restricted version of lexical values is convenience later on when reasoning about an ordering relation for values.

$$\frac{}{\mathit{Empty} : \mathbf{1}} \qquad \frac{}{\mathit{Char}\ c : c}$$

$$\frac{v_1 : r_1}{\mathit{Left}\ v_1 : r_1 + r_2} \qquad \frac{v_2 : r_1}{\mathit{Right}\ v_2 : r_2 + r_1}$$

$$\frac{v_1 : r_1 \qquad v_2 : r_2}{\mathit{Seq}\ v_1\ v_2 : r_1 \cdot r_2} \qquad \frac{\forall\, v \in vs.\ v : r \wedge |v| \neq []}{\mathit{Stars}\ vs : r^\star}$$

where in the clause for *Stars* we use the notation $v \in vs$ for indicating that v is a member in the list vs. We require in this rule that every value in vs flattens to a non-empty string. The idea is that *Stars*-values satisfy the informal Star Rule (see Introduction) where the * does not match the empty string unless this is the only match for the repetition. Note also that no values are associated with the regular expression $\mathbf{0}$, and that the only value associated with the regular expression $\mathbf{1}$ is *Empty*. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely

Proposition 2. $L(r) = \{|v| \mid v : r\}$

Given a regular expression r and a string s, we define the set of all *Lexical Values* inhabited by r with the underlying string being s:[6]

$$\mathit{LV}\ r\ s \stackrel{def}{=} \{v \mid v : r \wedge |v| = s\}$$

The main property of $\mathit{LV}\ r\ s$ is that it is alway finite.

Proposition 3. *finite* $(\mathit{LV}\ r\ s)$

This finiteness property does not hold in general if we remove the side-condition about $|v| \neq []$ in the *Stars*-rule above. For example using Sulzmann and Lu's less restrictive definition, $\mathit{LV}\ (\mathbf{1}^\star)\ []$ would contain infinitely many values, but according to our more restricted definition only a single value, namely $\mathit{LV}\ (\mathbf{1}^\star)\ [] = \{\mathit{Stars}\ []\}$.

If a regular expression r matches a string s, then generally the set $\mathit{LV}\ r\ s$ is not just a singleton set. In case of POSIX matching the problem is to calculate the unique lexical value that satisfies the (informal) POSIX rules from the Introduction. Graphically the POSIX value calculation algorithm by Sulzmann and Lu can be illustrated by the picture in Figure 1 where the path from the left to the right involving *derivatives*/*nullable* is the first phase of the algorithm (calculating successive Brzozowski's derivatives) and *mkeps*/*inj*, the path from right to left, the second phase. This picture shows the steps required when a regular expression, say r_1, matches the string $[a, b, c]$. We first build the three derivatives (according to a, b and c). We then use *nullable* to find out whether the resulting derivative regular expression r_4 can match the empty string. If yes, we call the function *mkeps* that produces a value v_4 for how r_4 can match the empty string (taking into account the POSIX constraints in case there are several ways). This function is defined by the clauses:

[6] Okui and Suzuki refer to our lexical values as *canonical values* in [**?**]. The notion of *non-problematic values* by Cardelli and Frisch [**?**] is related, but not identical to our lexical values.

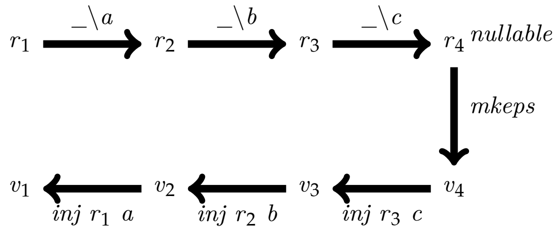

Fig. 1. The two phases of the algorithm by Sulzmann & Lu [**?**], matching the string $[a,\ b,\ c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building successive derivatives. If the last regular expression is *nullable*, then the functions of the second phase are called (the top-down and right-to-left arrows): first *mkeps* calculates a value v_4 witnessing how the empty string has been recognised by r_4. After that the function *inj* "injects back" the characters of the string into the values.

$$
\begin{array}{lcl}
\mathit{mkeps}\ \mathbf{1} & \stackrel{\text{def}}{=} & \mathit{Empty}\\
\mathit{mkeps}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{mkeps}\ r_2)\\
\mathit{mkeps}\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}\ r_1\ \mathit{then}\ \mathit{Left}\ (\mathit{mkeps}\ r_1)\ \mathit{else}\ \mathit{Right}\ (\mathit{mkeps}\ r_2)\\
\mathit{mkeps}\ (r^\star) & \stackrel{\text{def}}{=} & \mathit{Stars}\ []
\end{array}
$$

Note that this function needs only to be partially defined, namely only for regular expressions that are nullable. In case *nullable* fails, the string $[a,\ b,\ c]$ cannot be matched by r_1 and the null value *None* is returned. Note also how this function makes some subtle choices leading to a POSIX value: for example if an alternative regular expression, say $r_1 + r_2$, can match the empty string and furthermore r_1 can match the empty string, then we return a *Left*-value. The *Right*-value will only be returned if r_1 cannot match the empty string.

The most interesting idea from Sulzmann and Lu [**?**] is the construction of a value for how r_1 can match the string $[a,\ b,\ c]$ from the value how the last derivative, r_4 in Fig. 1, can match the empty string. Sulzmann and Lu achieve this by stepwise "injecting back" the characters into the values thus inverting the operation of building derivatives, but on the level of values. The corresponding function, called *inj*, takes three arguments, a regular expression, a character and a value. For example in the first (or right-most) *inj*-step in Fig. 1 the regular expression r_3, the character c from the last derivative step and v_4, which is the value corresponding to the derivative regular expression r_4. The result is the new value v_3. The final result of the algorithm is the value v_1. The *inj* function is defined by recursion on regular expressions and by analysing the shape of values (corresponding to the derivative regular expressions).

(1)	$inj\ d\ c\ (Empty)$	$\stackrel{\text{def}}{=}$	$Char\ d$
(2)	$inj\ (r_1 + r_2)\ c\ (Left\ v_1)$	$\stackrel{\text{def}}{=}$	$Left\ (inj\ r_1\ c\ v_1)$
(3)	$inj\ (r_1 + r_2)\ c\ (Right\ v_2)$	$\stackrel{\text{def}}{=}$	$Right\ (inj\ r_2\ c\ v_2)$
(4)	$inj\ (r_1 \cdot r_2)\ c\ (Seq\ v_1\ v_2)$	$\stackrel{\text{def}}{=}$	$Seq\ (inj\ r_1\ c\ v_1)\ v_2$
(5)	$inj\ (r_1 \cdot r_2)\ c\ (Left\ (Seq\ v_1\ v_2))$	$\stackrel{\text{def}}{=}$	$Seq\ (inj\ r_1\ c\ v_1)\ v_2$
(6)	$inj\ (r_1 \cdot r_2)\ c\ (Right\ v_2)$	$\stackrel{\text{def}}{=}$	$Seq\ (mkeps\ r_1)\ (inj\ r_2\ c\ v_2)$
(7)	$inj\ (r^\star)\ c\ (Seq\ v\ (Stars\ vs))$	$\stackrel{\text{def}}{=}$	$Stars\ (inj\ r\ c\ v :: vs)$

To better understand what is going on in this definition it might be instructive to look first at the three sequence cases (clauses *(4)* – *(6)*). In each case we need to construct an "injected value" for $r_1 \cdot r_2$. This must be a value of the form *Seq* _ _. Recall the clause of the *derivative*-function for sequence regular expressions:

$$(r_1 \cdot r_2)\backslash c \stackrel{\text{def}}{=} \textit{if nullable } r_1 \textit{ then } (r_1\backslash c) \cdot r_2 + (r_2\backslash c) \textit{ else } (r_1\backslash c) \cdot r_2$$

Consider first the *else*-branch where the derivative is $(r_1\backslash c) \cdot r_2$. The corresponding value must therefore be of the form $Seq\ v_1\ v_2$, which matches the left-hand side in clause *(4)* of *inj*. In the *if*-branch the derivative is an alternative, namely $(r_1\backslash c) \cdot r_2 + (r_2\backslash c)$. This means we either have to consider a *Left*- or *Right*-value. In case of the *Left*-value we know further it must be a value for a sequence regular expression. Therefore the pattern we match in the clause *(5)* is $Left\ (Seq\ v_1\ v_2)$, while in *(6)* it is just $Right\ v_2$. One more interesting point is in the right-hand side of clause *(6)*: since in this case the regular expression r_1 does not "contribute" to matching the string, that means it only matches the empty string, we need to call *mkeps* in order to construct a value for how r_1 can match this empty string. A similar argument applies for why we can expect in the left-hand side of clause *(7)* that the value is of the form $Seq\ v\ (Stars\ vs)$—the derivative of a star is $(r\backslash c) \cdot r^\star$. Finally, the reason for why we can ignore the second argument in clause *(1)* of *inj* is that it will only ever be called in cases where $c = d$, but the usual linearity restrictions in patterns do not allow us to build this constraint explicitly into our function definition.[7]

The idea of the *inj*-function to "inject" a character, say c, into a value can be made precise by the first part of the following lemma, which shows that the underlying string of an injected value has a prepended character c; the second part shows that the underlying string of an *mkeps*-value is always the empty string (given the regular expression is nullable since otherwise *mkeps* might not be defined).

Lemma 2.
(1) *If* $v : r\backslash c$ *then* $|inj\ r\ c\ v| = c :: |v|$.
(2) *If* $nullable\ r$ *then* $|mkeps\ r| = []$.

---

[7] Sulzmann and Lu state this clause as $inj\ c\ c\ (Empty) \stackrel{\text{def}}{=} Char\ c$, but our deviation is harmless.

Proof. Both properties are by routine inductions: the first one can, for example, be proved by induction over the definition of *derivatives*; the second by an induction on r. There are no interesting cases. □

Having defined the *mkeps* and *inj* function we can extend Brzozowski's matcher so that a value is constructed (assuming the regular expression matches the string). The clauses of the Sulzmann and Lu lexer are

$$\begin{array}{lcl} lexer\ r\ [] & \stackrel{\text{def}}{=} & if\ nullable\ r\ then\ Some\ (mkeps\ r)\ else\ None \\ lexer\ r\ (c::s) & \stackrel{\text{def}}{=} & case\ lexer\ (r\backslash c)\ s\ of \\ & & \quad None \Rightarrow None \\ & & \mid Some\ v \Rightarrow Some\ (inj\ r\ c\ v) \end{array}$$

If the regular expression does not match the string, *None* is returned. If the regular expression *does* match the string, then *Some* value is returned. One important virtue of this algorithm is that it can be implemented with ease in any functional programming language and also in Isabelle/HOL. In the remaining part of this section we prove that this algorithm is correct.

The well-known idea of POSIX matching is informally defined by some rules such as the Longest Match and Priority Rules (see Introduction); as correctly argued in [**?**], this needs formal specification. Sulzmann and Lu define an "ordering relation" between values and argue that there is a maximum value, as given by the derivative-based algorithm. In contrast, we shall introduce a simple inductive definition that specifies directly what a *POSIX value* is, incorporating the POSIX-specific choices into the side-conditions of our rules. Our definition is inspired by the matching relation given by Vansummeren [**?**]. The relation we define is ternary and written as $(s,\ r) \rightarrow v$, relating strings, regular expressions and values; the inductive rules are given in Figure 2. We can prove that given a string s and regular expression r, the POSIX value v is uniquely determined by $(s,\ r) \rightarrow v$.

Theorem 1.
(1) If $(s,\ r) \rightarrow v$ *then* $s \in L(r)$ *and* $|v| = s$.
(2) If $(s,\ r) \rightarrow v$ *and* $(s,\ r) \rightarrow v'$ *then* $v = v'$.

Proof. Both by induction on the definition of $(s,\ r) \rightarrow v$. The second parts follows by a case analysis of $(s,\ r) \rightarrow v'$ and the first part. □

We claim that our $(s,\ r) \rightarrow v$ relation captures the idea behind the four informal POSIX rules shown in the Introduction: Consider for example the rules $P{+}L$ and $P{+}R$ where the POSIX value for a string and an alternative regular expression, that is $(s,\ r_1 + r_2)$, is specified—it is always a *Left*-value, *except* when the string to be matched is not in the language of r_1; only then it is a *Right*-value (see the side-condition in $P{+}R$). Interesting is also the rule for sequence regular expressions (PS). The first two premises state that v_1 and v_2 are the POSIX values for $(s_1,\ r_1)$ and $(s_2,\ r_2)$ respectively. Consider now the third premise and note that the POSIX value of this rule should match the string $s_1 @ s_2$.

$$\frac{}{([],\ \mathbf{1}) \rightarrow \mathit{Empty}}P\mathbf{1} \qquad \frac{}{([c],\ c) \rightarrow \mathit{Char}\ c}Pc$$

$$\frac{(s,\ r_1) \rightarrow v}{(s,\ r_1 + r_2) \rightarrow \mathit{Left}\ v}P{+}L \qquad \frac{(s,\ r_2) \rightarrow v \qquad s \notin L(r_1)}{(s,\ r_1 + r_2) \rightarrow \mathit{Right}\ v}P{+}R$$

$$\frac{\begin{array}{l}(s_1,\ r_1) \rightarrow v_1 \qquad (s_2,\ r_2) \rightarrow v_2 \\ \nexists s_3\ s_4.a.\ s_3 \neq [] \wedge s_3 \mathbin{@} s_4 = s_2 \wedge s_1 \mathbin{@} s_3 \in L(r_1) \wedge s_4 \in L(r_2)\end{array}}{(s_1 \mathbin{@} s_2,\ r_1 \cdot r_2) \rightarrow \mathit{Seq}\ v_1\ v_2}PS$$

$$\frac{}{([],\ r^\star) \rightarrow \mathit{Stars}\ []}P[]$$

$$\frac{\begin{array}{l}(s_1,\ r) \rightarrow v \qquad (s_2,\ r^\star) \rightarrow \mathit{Stars}\ vs \qquad |v| \neq [] \\ \nexists s_3\ s_4.a.\ s_3 \neq [] \wedge s_3 \mathbin{@} s_4 = s_2 \wedge s_1 \mathbin{@} s_3 \in L(r) \wedge s_4 \in L(r^\star)\end{array}}{(s_1 \mathbin{@} s_2,\ r^\star) \rightarrow \mathit{Stars}\ (v :: vs)}P\star$$

Fig. 2. Our inductive definition of POSIX values.

According to the Longest Match Rule, we want that the s_1 is the longest initial split of $s_1 @ s_2$ such that s_2 is still recognised by r_2. Let us assume, contrary to the third premise, that there *exist* an s_3 and s_4 such that s_2 can be split up into a non-empty string s_3 and a possibly empty string s_4. Moreover the longer string $s_1 @ s_3$ can be matched by r_1 and the shorter s_4 can still be matched by r_2. In this case s_1 would *not* be the longest initial split of $s_1 @ s_2$ and therefore *Seq* v_1 v_2 cannot be a POSIX value for $(s_1 @ s_2,\ r_1 \cdot r_2)$. The main point is that our side-condition ensures the Longest Match Rule is satisfied.

A similar condition is imposed on the POSIX value in the $P\star$-rule. Also there we want that s_1 is the longest initial split of $s_1 @ s_2$ and furthermore the corresponding value v cannot be flattened to the empty string. In effect, we require that in each "iteration" of the star, some non-empty substring needs to be "chipped" away; only in case of the empty string we accept *Stars* [] as the POSIX value. Indeed we can show that our POSIX values are lexical values which exclude those *Stars* that contain subvalues that flatten to the empty string.

Lemma 3. *If* $(s,\ r) \rightarrow v$ *then* $v \in LV\ r\ s$.

Proof. By routine induction on $(s,\ r) \rightarrow v$. □

Next is the lemma that shows the function *mkeps* calculates the POSIX value for the empty string and a nullable regular expression.

Lemma 4. *If nullable r then* $([],\ r) \rightarrow \mathit{mkeps}\ r$.

Proof. By routine induction on r. □

The central lemma for our POSIX relation is that the *inj*-function preserves POSIX values.

Lemma 5. *If* $(s, r\backslash c) \to v$ *then* $(c :: s, r) \to inj\ r\ c\ v$.

Proof. By induction on r. We explain two cases.

- Case $r = r_1 + r_2$. There are two subcases, namely (a) $v = Left\ v'$ and $(s, r_1\backslash c) \to v'$; and (b) $v = Right\ v'$, $s \notin L(r_1\backslash c)$ and $(s, r_2\backslash c) \to v'$. In (a) we know $(s, r_1\backslash c) \to v'$, from which we can infer $(c :: s, r_1) \to inj\ r_1\ c\ v'$ by induction hypothesis and hence $(c :: s, r_1 + r_2) \to inj\ (r_1 + r_2)\ c\ (Left\ v')$ as needed. Similarly in subcase (b) where, however, in addition we have to use Proposition 1(2) in order to infer $c :: s \notin L(r_1)$ from $s \notin L(r_1\backslash c)$.
- Case $r = r_1 \cdot r_2$. There are three subcases:
  (a) $v = Left\ (Seq\ v_1\ v_2)$ and $nullable\ r_1$
  (b) $v = Right\ v_1$ and $nullable\ r_1$
  (c) $v = Seq\ v_1\ v_2$ and $\neg\ nullable\ r_1$
  For (a) we know $(s_1, r_1\backslash c) \to v_1$ and $(s_2, r_2) \to v_2$ as well as

  $$\nexists s_3\ s_4.a.\ s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L(r_1\backslash c) \wedge s_4 \in L(r_2)$$

  From the latter we can infer by Proposition 1(2):

  $$\nexists s_3\ s_4.a.\ s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge c :: s_1 @ s_3 \in L(r_1) \wedge s_4 \in L(r_2)$$

  We can use the induction hypothesis for r_1 to obtain $(c :: s_1, r_1) \to inj\ r_1\ c\ v_1$. Putting this all together allows us to infer $(c :: s_1 @ s_2, r_1 \cdot r_2) \to Seq\ (inj\ r_1\ c\ v_1)\ v_2$. The case (c) is similar.
  For (b) we know $(s, r_2\backslash c) \to v_1$ and $s_1 @ s_2 \notin L((r_1\backslash c) \cdot r_2)$. From the former we have $(c :: s, r_2) \to inj\ r_2\ c\ v_1$ by induction hypothesis for r_2. From the latter we can infer

  $$\nexists s_3\ s_4.a.\ s_3 \neq [] \wedge s_3 @ s_4 = c :: s \wedge s_3 \in L(r_1) \wedge s_4 \in L(r_2)$$

  By Lemma 4 we know $([], r_1) \to mkeps\ r_1$ holds. Putting this all together, we can conclude with $(c :: s, r_1 \cdot r_2) \to Seq\ (mkeps\ r_1)\ (inj\ r_2\ c\ v_1)$, as required.
  Finally suppose $r = r_1{}^\star$. This case is very similar to the sequence case, except that we need to also ensure that $|inj\ r_1\ c\ v_1| \neq []$. This follows from $(c :: s_1, r_1) \to inj\ r_1\ c\ v_1$ (which in turn follows from $(s_1, r_1\backslash c) \to v_1$ and the induction hypothesis). □

With Lemma 5 in place, it is completely routine to establish that the Sulzmann and Lu lexer satisfies our specification (returning the null value *None* iff the string is not in the language of the regular expression, and returning a unique POSIX value iff the string *is* in the language):

Theorem 2.
(1) $s \notin L(r)$ *if and only if* $lexer\ r\ s = None$
(2) $s \in L(r)$ *if and only if* $\exists v.\ lexer\ r\ s = Some\ v \wedge (s, r) \to v$

Proof. By induction on s using Lemma 4 and 5. □

In *(2)* we further know by Theorem 1 that the value returned by the lexer must be unique. A simple corollary of our two theorems is:

Corollary 1.
(1) $lexer\ r\ s = None$ *if and only if* $\nexists\, v.a.\ (s,\ r) \rightarrow v$
(2) $lexer\ r\ s = Some\ v$ *if and only if* $(s,\ r) \rightarrow v$

This concludes our correctness proof. Note that we have not changed the algorithm of Sulzmann and Lu,[8] but introduced our own specification for what a correct result—a POSIX value—should be. In the next section we show that our specification coincides with another one given by Okui and Suzuki using a different technique.

6 Ordering of Values according to Okui and Suzuki

While in the previous section we have defined POSIX values directly in terms of a ternary relation (see inference rules in Figure 2), Sulzmann and Lu took a different approach in [**?**]: they introduced an ordering for values and identified POSIX values as the maximal elements. An extended version of [**?**] is available at the website of its first author; this includes more details of their proofs, but which are evidently not in final form yet. Unfortunately, we were not able to verify claims that their ordering has properties such as being transitive or having maximal elements.

Okui and Suzuki [**?**,**?**] described another ordering of values, which they use to establish the correctness of their automata-based algorithm for POSIX matching. Their ordering resembles some aspects of the one given by Sulzmann and Lu, but overall is quite different. To begin with, Okui and Suzuki identify POSIX values as minimal, rather than maximal, elements in their ordering. A more substantial difference is that the ordering by Okui and Suzuki uses *positions* in order to identify and compare subvalues. Positions are lists of natural numbers. This allows them to quite naturally formalise the Longest Match and Priority rules of the informal POSIX standard. Consider for example the value v

$$v \stackrel{def}{=} Stars\ [Seq\ (Char\ x)\ (Char\ y),\ Char\ z]$$

At position $[0,1]$ of this value is the subvalue $Char\ y$ and at position $[1]$ the subvalue $Char\ z$. At the 'root' position, or empty list $[]$, is the whole value v. Positions such as $[0,1,0]$ or $[2]$ are outside of v. If it exists, the subvalue of v at a position p, written $v\downharpoonleft_p$, can be recursively defined by

[8] All deviations we introduced are harmless.

$$\begin{array}{rcl}
v\downharpoonleft_{[]} & \stackrel{def}{=} & v \\
\mathit{Left}\ v\downharpoonleft_{0::ps} & \stackrel{def}{=} & v\downharpoonleft_{ps} \\
\mathit{Right}\ v\downharpoonleft_{1::ps} & \stackrel{def}{=} & v\downharpoonleft_{ps} \\
\mathit{Seq}\ v_1\ v_2\downharpoonleft_{0::ps} & \stackrel{def}{=} & v_1\downharpoonleft_{ps} \\
\mathit{Seq}\ v_1\ v_2\downharpoonleft_{1::ps} & \stackrel{def}{=} & v_2\downharpoonleft_{ps} \\
\mathit{Stars}\ vs\downharpoonleft_{n::ps} & \stackrel{def}{=} & vs_{[n]}\downharpoonleft_{ps}
\end{array}$$

In the last clause we use Isabelle's notation $vs_{[n]}$ for the nth element in a list. The set of positions inside a value v, written $\mathit{Pos}\ v$, is given by

$$\begin{array}{lcl}
\mathit{Pos}\ (\mathit{Empty}) & \stackrel{def}{=} & \{[]\} \\
\mathit{Pos}\ (\mathit{Char}\ c) & \stackrel{def}{=} & \{[]\} \\
\mathit{Pos}\ (\mathit{Left}\ v) & \stackrel{def}{=} & \{[]\} \cup \{0 :: ps \mid ps \in \mathit{Pos}\ v\} \\
\mathit{Pos}\ (\mathit{Right}\ v) & \stackrel{def}{=} & \{[]\} \cup \{1 :: ps \mid ps \in \mathit{Pos}\ v\} \\
\mathit{Pos}\ (\mathit{Seq}\ v_1\ v_2) & \stackrel{def}{=} & \{[]\} \cup \{0 :: ps \mid ps \in \mathit{Pos}\ v_1\} \cup \{1 :: ps \mid ps \in \mathit{Pos}\ v_2\} \\
\mathit{Pos}\ (\mathit{Stars}\ vs) & \stackrel{def}{=} & \{[]\} \cup (\bigcup n < \mathit{len}\ vs\ \{n :: ps \mid ps \in \mathit{Pos}\ vs_{[n]}\})
\end{array}$$

whereby *len* in the last clause stands for the length of a list. Clearly for every position inside a value there exists a subvalue at that position.

To help understanding the ordering of Okui and Suzuki, consider again the earlier value v and compare it with the following w:

$$\begin{array}{lcl}
v & \stackrel{def}{=} & \mathit{Stars}\ [\mathit{Seq}\ (\mathit{Char}\ x)\ (\mathit{Char}\ y),\ \mathit{Char}\ z] \\
w & \stackrel{def}{=} & \mathit{Stars}\ [\mathit{Char}\ x,\ \mathit{Char}\ y,\ \mathit{Char}\ z]
\end{array}$$

Both values match the string *xyz*, that means if we flatten these values at their respective root position, we obtain *xyz*. However, at position $[0]$, v matches *xy* whereas w matches only the shorter x. So according to the Longest Match Rule, we should prefer v, rather than w as POSIX value for string *xyz* (and corresponding regular expression). In order to formalise this idea, Okui and Suzuki introduce a measure for subvalues at position p, called the *norm* of v at position p. We can define this measure in Isabelle as an integer as follows

$$\|v\|_p \stackrel{def}{=} \mathit{if}\ p \in \mathit{Pos}\ v\ \mathit{then}\ \mathit{len}\ |v\downharpoonleft_p|\ \mathit{else}\ -\ 1$$

where we take the length of the flattened value at position p, provided the position is inside v; if not, then the norm is -1. The default for outside positions is crucial for the POSIX requirement of preferring a *Left*-value over a *Right*-value (if they can match the same string—see the Priority Rule from the Introduction). For this consider

$$v \stackrel{def}{=} \mathit{Left}\ (\mathit{Char}\ x) \qquad \text{and} \qquad w \stackrel{def}{=} \mathit{Right}\ (\mathit{Char}\ x)$$

Both values match x. At position $[\mathit{0}]$ the norm of v is $\mathit{1}$ (the subvalue matches x), but the norm of w is $-\mathit{1}$ (the position is outside w according to how we defined the 'inside' positions of *Left*- and *Right*-values). Of course at position $[\mathit{1}]$, the norms $\|v\|_{[\mathit{1}]}$ and $\|w\|_{[\mathit{1}]}$ are reversed, but the point is that subvalues will be analysed according to lexicographically ordered positions. According to this ordering, the position $[\mathit{0}]$ takes precedence over $[\mathit{1}]$ and thus also v will be preferred over w. The lexicographic ordering of positions, written $\_ \prec_{lex} \_$, can be conveniently formalised by three inference rules

$$\frac{}{[] \prec_{lex} p :: ps} \qquad \frac{p_1 < p_2}{p_1 :: ps_1 \prec_{lex} p_2 :: ps_2} \qquad \frac{ps_1 \prec_{lex} ps_2}{p :: ps_1 \prec_{lex} p :: ps_2}$$

With the norm and lexicographic order in place, we can state the key definition of Okui and Suzuki [**?**]: a value v_1 is *smaller at position* p than v_2, written $v_1 \prec_p v_2$, if and only if (i) the norm at position p is greater in v_1 (that is the string $|v_1{\downarrow}_p|$ is longer than $|v_2{\downarrow}_p|$) and (ii) all subvalues at positions that are inside v_1 or v_2 and that are lexicographically smaller than p, we have the same norm, namely

$$v_1 \prec_p v_2 \stackrel{def}{=} \begin{cases} (i) & \|v_2\|_p < \|v_1\|_p \quad \text{and} \\ (ii) & \forall\, q \in \mathit{Pos}\ v_1 \cup \mathit{Pos}\ v_2.\ q \prec_{lex} p \longrightarrow \|v_1\|_q = \|v_2\|_q \end{cases}$$

The position p in this definition acts as the *first distinct position* of v_1 and v_2, where both values match strings of different length [**?**]. Since at p the values v_1 and v_2 match different strings, the ordering is irreflexive. Derived from the definition above are the following two orderings:

$$\begin{aligned} v_1 \prec v_2 &\stackrel{def}{=} \exists\, p.\ v_1 \prec_p v_2 \\ v_1 \preccurlyeq v_2 &\stackrel{def}{=} v_1 \prec v_2 \vee v_1 = v_2 \end{aligned}$$

While we encountered a number of obstacles for establishing properties like transitivity for the ordering of Sulzmann and Lu (and which we failed to overcome), it is relatively straightforward to establish this property for the orderings $\_ \prec \_$ and $\_ \preccurlyeq \_$ by Okui and Suzuki.

Lemma 6 (Transitivity). *If* $v_1 \prec v_2$ *and* $v_2 \prec v_3$ *then* $v_1 \prec v_3$.

Proof. From the assumption we obtain two positions p and q, where the values v_1 and v_2 (respectively v_2 and v_3) are 'distinct'. Since $\prec_{lex}$ is trichotomous, we need to consider three cases, namely $p = q$, $p \prec_{lex} q$ and $q \prec_{lex} p$. Let us look at the first case. Clearly $\|v_2\|_p < \|v_1\|_p$ and $\|v_3\|_p < \|v_2\|_p$ imply $\|v_3\|_p < \|v_1\|_p$. It remains to show that for a $p' \in \mathit{Pos}\ v_1 \cup \mathit{Pos}\ v_3$ with $p' \prec_{lex} p$ that $\|v_1\|_{p'} = \|v_3\|_{p'}$ holds. Suppose $p' \in \mathit{Pos}\ v_1$, then we can infer from the first assumption that $\|v_1\|_{p'} = \|v_2\|_{p'}$. But this means that p' must be in $\mathit{Pos}\ v_2$ too (the norm cannot be $-\mathit{1}$ given $p' \in \mathit{Pos}\ v_1$). Hence we can use the second assumption and infer $\|v_2\|_{p'} = \|v_3\|_{p'}$, which concludes this case with $v_1 \prec v_3$. The reasoning in the other cases is similar. □

The proof for $\preccurlyeq$ is similar and omitted. It is also straightforward to show that $\prec$ and $\preccurlyeq$ are partial orders. Okui and Suzuki furthermore show that they are linear orderings for lexical values [?] of a given regular expression and given string, but we have not formalised this in Isabelle. It is not essential for our results. What we are going to show below is that for a given r and s, the orderings have a unique minimal element on the set $LV\ r\ s$, which is the POSIX value we defined in the previous section. We start with two properties that show how the length of a flattened value relates to the $\prec$-ordering.

Proposition 4.
(1) If $v_1 \prec v_2$ *then* $len\ |v_2| \leq len\ |v_1|$.
(2) If $len\ |v_2| < len\ |v_1|$ *then* $v_1 \prec v_2$.

Both properties follow from the definition of the ordering. Note that *(2)* entails that a value, say v_2, whose underlying string is a strict prefix of another flattened value, say v_1, then v_1 must be smaller than v_2. For our proofs it will be useful to have the following properties—in each case the underlying strings of the compared values are the same:

Proposition 5.
(1) If $|v_1| = |v_2|$ *then* $Left\ v_1 \prec Right\ v_2$.
(2) If $|v_1| = |v_2|$ *then* $Left\ v_1 \prec Left\ v_2$ *iff* $v_1 \prec v_2$
(3) If $|v_1| = |v_2|$ *then* $Right\ v_1 \prec Right\ v_2$ *iff* $v_1 \prec v_2$
(4) If $|v_2| = |w_2|$ *then* $Seq\ v\ v_2 \prec Seq\ v\ w_2$ *iff* $v_2 \prec w_2$
(5) If $|v_1|\ @\ |v_2| = |w_1|\ @\ |w_2|$ *and* $v_1 \prec w_1$ *then* $Seq\ v_1\ v_2 \prec Seq\ w_1\ w_2$
(6) If $|vs_1| = |vs_2|$ *then* $Stars\ (vs\ @\ vs_1) \prec Stars\ (vs\ @\ vs_2)$ *iff* $Stars\ vs_1 \prec Stars\ vs_2$
(7) If $|v_1 :: vs_1| = |v_2 :: vs_2|$ *and* $v_1 \prec v_2$ *then* $Stars\ (v_1 :: vs_1) \prec Stars\ (v_2 :: vs_2)$

One might prefer that statements *(4)* and *(5)* (respectively *(6)* and *(7)*) are combined into a single *iff*-statement (like the ones for *Left* and *Right*). Unfortunately this cannot be done easily: such a single statement would require an additional assumption about the two values $Seq\ v_1\ v_2$ and $Seq\ w_1\ w_2$ being inhabited by the same regular expression. The complexity of the proofs involved seems to not justify such a 'cleaner' single statement. The statements given are just the properties that allow us to establish our theorems without any difficulty. The proofs for Proposition 5 are routine.

Next we establish how Okui and Suzuki's orderings relate to our definition of POSIX values. Given a *POSIX* value v_1 for r and s, then any other lexical value v_2 in $LV\ r\ s$ is greater or equal than v_1, namely:

Theorem 3. *If* $(s, r) \rightarrow v_1$ *and* $v_2 \in LV\ r\ s$ *then* $v_1 \preccurlyeq v_2$.

Proof. By induction on our POSIX rules. By Theorem 1 and the definition of LV, it is clear that v_1 and v_2 have the same underlying string s. The three base cases are straightforward: for example for $v_1 = Empty$, we have that $v_2 \in LV\ \mathbf{1}\ []$ must also be of the form $v_2 = Empty$. Therefore we have $v_1 \preccurlyeq v_2$. The inductive cases for r being of the form $r_1 + r_2$ and $r_1 \cdot r_2$ are as follows:

- Case $P{+}L$ with $(s,\, r_1 + r_2) \rightarrow \mathit{Left}\ w_1$: In this case the value v_2 is either of the form $\mathit{Left}\ w_2$ or $\mathit{Right}\ w_2$. In the latter case we can immediately conclude with $v_1 \preccurlyeq v_2$ since a *Left*-value with the same underlying string s is always smaller than a *Right*-value by Proposition 5*(1)*. In the former case we have $w_2 \in \mathit{LV}\ r_1\ s$ and can use the induction hypothesis to infer $w_1 \preccurlyeq w_2$. Because w_1 and w_2 have the same underlying string s, we can conclude with $\mathit{Left}\ w_1 \preccurlyeq \mathit{Left}\ w_2$ using Proposition 5*(2)*.
- Case $P{+}R$ with $(s,\, r_1 + r_2) \rightarrow \mathit{Right}\ w_1$: This case similar to the previous case, except that we additionally know $s \notin L(r_1)$. This is needed when v_2 is of the form $\mathit{Left}\ w_2$. Since $|v_2| = |w_2| = s$ and $w_2 : r_1$, we can derive a contradiction for $s \notin L(r_1)$ using Proposition 2. So also in this case $v_1 \preccurlyeq v_2$.
- Case PS with $(s_1 \mathbin{@} s_2,\, r_1 \cdot r_2) \rightarrow \mathit{Seq}\ w_1\ w_2$: We can assume $v_2 = \mathit{Seq}\ u_1\ u_2$ with $u_1 : r_1$ and $u_2 : r_2$. We have $s_1 \mathbin{@} s_2 = |u_1| \mathbin{@} |u_2|$. By the side-condition of the *PS*-rule we know that either $s_1 = |u_1|$ or that $|u_1|$ is a strict prefix of s_1. In the latter case we can infer $w_1 \prec u_1$ by Proposition 4*(2)* and from this $v_1 \preccurlyeq v_2$ by Proposition 5*(5)* (as noted above v_1 and v_2 must have the same underlying string). In the former case we know $u_1 \in \mathit{LV}\ r_1\ s_1$ and $u_2 \in \mathit{LV}\ r_2\ s_2$. With this we can use the induction hypotheses to infer $w_1 \preccurlyeq u_1$ and $w_2 \preccurlyeq u_2$. By Proposition 5*(4,5)* we can again infer $v_1 \preccurlyeq v_2$.

The case for $P\star$ is similar to the *PS*-case and omitted. □

This theorem shows that our *POSIX* value for a regular expression r and string s is in fact a minimal element of the values in $\mathit{LV}\ r\ s$. By Proposition 4*(2)* we also know that any value in $\mathit{LV}\ r\ s'$, with s' being a strict prefix, cannot be smaller than v_1. The next theorem shows the opposite—namely any minimal element in $\mathit{LV}\ r\ s$ must be a *POSIX* value. This can be established by induction on r, but the proof can be drastically simplified by using the fact from the previous section about the existence of a *POSIX* value whenever a string $s \in L(r)$.

Theorem 4. *If* $v_1 \in \mathit{LV}\ r\ s$ *and* $\forall\, v_2 \in \mathit{LV}\ r\ s.\ v_2 \not\prec v_1$ *then* $(s,\, r) \rightarrow v_1$.

Proof. If $v_1 \in \mathit{LV}\ r\ s$ then $s \in L(r)$ by Proposition 2. Hence by Theorem 2(2) there exists a *POSIX* value v_P with $(s,\, r) \rightarrow v_P$ and by Lemma 3 we also have $v_P \in \mathit{LV}\ r\ s$. By Theorem 3 we therefore have $v_P \preccurlyeq v_1$. If $v_P = v_1$ then we are done. Otherwise we have $v_P \prec v_1$, which however contradicts the second assumption about v_1 being the smallest element in $\mathit{LV}\ r\ s$. So we are done in this case too. □

From this we can also show that if $\mathit{LV}\ r\ s$ is non-empty (or equivalently $s \in L(r)$) then it has a unique minimal element:

Corollary 2. *If* $\mathit{LV}\ r\ s \neq \varnothing$ *then* $\exists!\mathit{vmin}.\ \mathit{vmin} \in \mathit{LV}\ r\ s \wedge (\forall\, v \in \mathit{LV}\ r\ s.\ \mathit{vmin} \preccurlyeq v)$.

To sum up, we have shown that the (unique) minimal elements of the ordering by Okui and Suzuki are exactly the *POSIX* values we defined inductively in Section 5. This provides an independent confirmation that our ternary relation formalises the informal POSIX rules.

7 Bitcoded Lexing

Incremental calculation of the value. To simplify the proof we first define the function *flex* which calculates the "iterated" injection function. With this we can rewrite the lexer as

$$lexer\ r\ s = (if\ nullable\ (r\backslash s)\ then\ Some\ (flex\ r\ id\ s\ (mkeps\ (r\backslash s)))\ else\ None)$$

8 Optimisations

Derivatives as calculated by Brzozowski's method are usually more complex regular expressions than the initial one; the result is that the derivative-based matching and lexing algorithms are often abysmally slow. However, various optimisations are possible, such as the simplifications of $\mathbf{0} + r$, $r + \mathbf{0}$, $\mathbf{1} \cdot r$ and $r \cdot \mathbf{1}$ to r. These simplifications can speed up the algorithms considerably, as noted in [**?**]. One of the advantages of having a simple specification and correctness proof is that the latter can be refined to prove the correctness of such simplification steps. While the simplification of regular expressions according to rules like

$$\mathbf{0} + r \Rightarrow r \qquad r + \mathbf{0} \Rightarrow r \qquad \mathbf{1} \cdot r \Rightarrow r \qquad r \cdot \mathbf{1} \Rightarrow r \tag{2}$$

is well understood, there is an obstacle with the POSIX value calculation algorithm by Sulzmann and Lu: if we build a derivative regular expression and then simplify it, we will calculate a POSIX value for this simplified derivative regular expression, *not* for the original (unsimplified) derivative regular expression. Sulzmann and Lu [**?**] overcome this obstacle by not just calculating a simplified regular expression, but also calculating a *rectification function* that "repairs" the incorrect value.

The rectification functions can be (slightly clumsily) implemented in Isabelle/HOL as follows using some auxiliary functions:

$$
\begin{array}{lcl}
F_{Right}\, f\, v & \stackrel{\text{def}}{=} & Right\ (f\ v)\\
F_{Left}\, f\, v & \stackrel{\text{def}}{=} & Left\ (f\ v)\\
F_{Alt}\, f_1\ f_2\ (Right\ v) & \stackrel{\text{def}}{=} & Right\ (f_2\ v)\\
F_{Alt}\, f_1\ f_2\ (Left\ v) & \stackrel{\text{def}}{=} & Left\ (f_1\ v)\\
F_{Seq1}\, f_1\ f_2\ v & \stackrel{\text{def}}{=} & Seq\ (f_1\ ())\ (f_2\ v)\\
F_{Seq2}\, f_1\ f_2\ v & \stackrel{\text{def}}{=} & Seq\ (f_1\ v)\ (f_2\ ())\\
F_{Seq}\, f_1\ f_2\ (Seq\ v_1\ v_2) & \stackrel{\text{def}}{=} & Seq\ (f_1\ v_1)\ (f_2\ v_2)\\[1ex]
simp_{Alt}\ (\mathbf{0},\ \_)\ (r_2,\ f_2) & \stackrel{\text{def}}{=} & (r_2,\ F_{Right}\, f_2)\\
simp_{Alt}\ (r_1,\ f_1)\ (\mathbf{0},\ \_) & \stackrel{\text{def}}{=} & (r_1,\ F_{Left}\, f_1)\\
simp_{Alt}\ (r_1,\ f_1)\ (r_2,\ f_2) & \stackrel{\text{def}}{=} & (r_1 + r_2,\ F_{Alt}\, f_1\ f_2)\\
simp_{Seq}\ (\mathbf{1},\ f_1)\ (r_2,\ f_2) & \stackrel{\text{def}}{=} & (r_2,\ F_{Seq1}\, f_1\ f_2)\\
simp_{Seq}\ (r_1,\ f_1)\ (\mathbf{1},\ f_2) & \stackrel{\text{def}}{=} & (r_1,\ F_{Seq2}\, f_1\ f_2)\\
simp_{Seq}\ (r_1,\ f_1)\ (r_2,\ f_2) & \stackrel{\text{def}}{=} & (r_1 \cdot r_2,\ F_{Seq}\, f_1\ f_2)
\end{array}
$$

The functions $simp_{Alt}$ and $simp_{Seq}$ encode the simplification rules in (2) and compose the rectification functions (simplifications can occur deep inside the regular expression). The main simplification function is then

$$
\begin{array}{lcl}
simp\ (r_1 + r_2) & \stackrel{\text{def}}{=} & simp_{Alt}\ (simp\ r_1)\ (simp\ r_2)\\
simp\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & simp_{Seq}\ (simp\ r_1)\ (simp\ r_2)\\
simp\ r & \stackrel{\text{def}}{=} & (r,\ id)
\end{array}
$$

where *id* stands for the identity function. The function *simp* returns a simplified regular expression and a corresponding rectification function. Note that we do not simplify under stars: this seems to slow down the algorithm, rather than speed it up. The optimised lexer is then given by the clauses:

$$
\begin{array}{lcl}
lexer^+\ r\ [] & \stackrel{\text{def}}{=} & if\ nullable\ r\ then\ Some\ (mkeps\ r)\ else\ None\\
lexer^+\ r\ (c :: s) & \stackrel{\text{def}}{=} & let\ (r_s,\ f_r) = simp\ (r \backslash c)\ in\\
& & case\ lexer^+\ r_s\ s\ of\\
& & \quad None \Rightarrow None\\
& & \mid Some\ v \Rightarrow Some\ (inj\ r\ c\ (f_r\ v))
\end{array}
$$

In the second clause we first calculate the derivative $r \backslash c$ and then simpli

text *Incremental calculation of the value. To simplify the proof we first define the function* @{*const flex*} *which calculates the ''iterated'' injection function. With this we can rewrite the lexer as* \begin{*center*} @{*thm lexer___flex*} \end{*center*} \begin{*center*} \begin{*tabular*}{*lcl*} @{*thm* (*lhs*) *code.simps*(*1*)} & $\dn$ & @{*thm* (*rhs*) *code.simps*(*1*)}\\ @{*thm* (*lhs*) *code.simps*(*2*)} & $\dn$ & @{*thm* (*rhs*) *code.simps*(*2*)}\\ @{*thm* (*lhs*) *code.simps*(*3*)} & $\dn$

& @{thm (rhs) code.simps(3)}\\ @{thm (lhs) code.simps(4)} & $\dn$ & @{thm (rhs) code.simps(4)}\\ @{thm (lhs) code.simps(5)[of v_1 v_2]} & $\dn$ & @{thm (rhs) code.simps(5)[of v_1 v_2]}\\ @{thm (lhs) code.simps(6)} & $\dn$ & @{thm (rhs) code.simps(6)}\\ @{thm (lhs) code.simps(7)} & $\dn$ & @{thm (rhs) code.simps(7)} \end{tabular} \end{center} \begin{center} \begin{tabular}{lcl} @{term areg} & $::=$ & @{term AZERO}\\ & $\mid$ & @{term AONE bs}\\ & $\mid$ & @{term ACHAR bs c}\\ & $\mid$ & @{term AALT bs r1 r2}\\ & $\mid$ & @{term ASEQ bs r_1 r_2}\\ & $\mid$ & @{term ASTAR bs r} \end{tabular} \end{center} \begin{center} \begin{tabular}{lcl} @{thm (lhs) intern.simps(1)} & $\dn$ & @{thm (rhs) intern.simps(1)}\\ @{thm (lhs) intern.simps(2)} & $\dn$ & @{thm (rhs) intern.simps(2)}\\ @{thm (lhs) intern.simps(3)} & $\dn$ & @{thm (rhs) intern.simps(3)}\\ @{thm (lhs) intern.simps(4)[of r_1 r_2]} & $\dn$ & @{thm (rhs) intern.simps(4)[of r_1 r_2]}\\ @{thm (lhs) intern.simps(5)[of r_1 r_2]} & $\dn$ & @{thm (rhs) intern.simps(5)[of r_1 r_2]}\\ @{thm (lhs) intern.simps(6)} & $\dn$ & @{thm (rhs) intern.simps(6)}\\ \end{tabular} \end{center} \begin{center} \begin{tabular}{lcl} @{thm (lhs) erase.simps(1)} & $\dn$ & @{thm (rhs) erase.simps(1)}\\ @{thm (lhs) erase.simps(2)[of bs]} & $\dn$ & @{thm (rhs) erase.simps(2)[of bs]}\\ @{thm (lhs) erase.simps(3)[of bs]} & $\dn$ & @{thm (rhs) erase.simps(3)[of bs]}\\ @{thm (lhs) erase.simps(4)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) erase.simps(4)[of bs r_1 r_2]}\\ @{thm (lhs) erase.simps(5)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) erase.simps(5)[of bs r_1 r_2]}\\ @{thm (lhs) erase.simps(6)[of bs]} & $\dn$ & @{thm (rhs) erase.simps(6)[of bs]}\\ \end{tabular} \end{center} Some simple facts about erase \begin{lemma}\mbox{}\\ @{thm erase___bder}\\ @{thm erase___intern} \end{lemma} \begin{center} \begin{tabular}{lcl} @{thm (lhs) bnullable.simps(1)} & $\dn$ & @{thm (rhs) bnullable.simps(1)}\\ @{thm (lhs) bnullable.simps(2)} & $\dn$ & @{thm (rhs) bnullable.simps(2)}\\ @{thm (lhs) bnullable.simps(3)} & $\dn$ & @{thm (rhs) bnullable.simps(3)}\\ @{thm (lhs) bnullable.simps(4)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) bnullable.simps(4)[of bs r_1 r_2]}\\ @{thm (lhs) bnullable.simps(5)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) bnullable.simps(5)[of bs r_1 r_2]}\\ @{thm (lhs) bnullable.simps(6)} & $\dn$ & @{thm (rhs) bnullable.simps(6)}\medskip\\ % \end{tabular} % \end{center} % \begin{center} % \begin{tabular}{lcl} @{thm (lhs) bder.simps(1)} & $\dn$ & @{thm (rhs) bder.simps(1)}\\ @{thm (lhs) bder.simps(2)} & $\dn$ & @{thm (rhs) bder.simps(2)}\\ @{thm (lhs) bder.simps(3)} & $\dn$ & @{thm (rhs) bder.simps(3)}\\ @{thm (lhs) bder.simps(4)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) bder.simps(4)[of bs r_1 r_2]}\\ @{thm (lhs) bder.simps(5)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) bder.simps(5)[of bs r_1 r_2]}\\ @{thm (lhs) bder.simps(6)} & $\dn$ & @{thm (rhs) bder.simps(6)} \end{tabular} \end{center} \begin{center} \begin{tabular}{lcl} @{thm (lhs) bmkeps.simps(1)} & $\dn$ & @{thm (rhs) bmkeps.simps(1)}\\ @{thm (lhs) bmkeps.simps(2)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) bmkeps.simps(2)[of bs r_1 r_2]}\\ @{thm (lhs) bmkeps.simps(3)[of bs r_1 r_2]} & $\dn$ & @{thm (rhs) bmkeps.simps(3)[of bs r_1 r_2]}\\ @{thm (lhs) bmkeps.simps(4)} & $\dn$ & @{thm (rhs) bmkeps.simps(4)}\medskip\\ \end{tabular} \end{center} @{thm *[mode=IfThen] bder__retrieve} By induction on ⟨r⟩ \begin{theorem}[Main Lemma]\mbox{}\\ @{thm [mode=IfThen] MAIN__decode} \end{theorem} \noindent Definition of the bitcoded lexer @{thm blexer__def} \begin{theorem} @{thm blexer__correctness} \end{theorem}*

section *Optimisations*

text *Derivatives as calculated by \Brz's method are usually more complex regular expressions than the initial one; the result is that the derivative−based matching and lexing algorithms are often abysmally slow. However, various optimisations are possible, such as the simplifications of @{term ALT ZERO r}, @{term ALT r ZERO}, @{term SEQ ONE r} and @{term SEQ r ONE} to @{term r}. These simplifications can speed up the algorithms considerably, as noted in \cite{Sulzmann2014}. One of the advantages of having a simple specification and correctness proof is that the latter can be refined to prove the correctness of such simplification steps. While the simplification of regular expressions according to rules like \begin{equation}\label{Simpl} \begin{array}{lcllcllcllcl} @{term ALT ZERO r} & ⟨⇒⟩ & @{term r} \hspace{8mm}%\\ @{term ALT r ZERO} & ⟨⇒⟩ & @{term r} \hspace{8mm}%\\ @{term SEQ ONE r} & ⟨⇒⟩ & @{term r} \hspace{8mm}%\\ @{term SEQ r ONE} & ⟨⇒⟩ & @{term r} \end{array} \end{equation} \noindent is well understood, there is an obstacle with the POSIX value calculation algorithm by Sulzmann and Lu: if we build a derivative regular expression and then simplify it, we will calculate a POSIX value for this simplified derivative regular expression, \emph{not} for the original (unsimplified) derivative regular expression. Sulzmann and Lu \cite{Sulzmann2014} overcome this obstacle by not just calculating a simplified regular expression, but also calculating a \emph{rectification function} that ``repairs'' the incorrect value. The rectification functions can be (slightly clumsily) implemented in Isabelle/HOL as follows using some auxiliary functions: \begin{center} \begin{tabular}{lcl} @{thm (lhs) F__RIGHT.simps(1)} & \$\dn\$ & ⟨Right (f v)⟩\\ @{thm (lhs) F__LEFT.simps(1)} & \$\dn\$ & ⟨Left (f v)⟩\\ @{thm (lhs) F__ALT.simps(1)} & \$\dn\$ & ⟨Right (f_2 v)⟩\\ @{thm (lhs) F__ALT.simps(2)} & \$\dn\$ & ⟨Left (f_1 v)⟩\\ @{thm (lhs) F__SEQ1.simps(1)} & \$\dn\$ & ⟨Seq (f_1 ()) (f_2 v)⟩\\ @{thm (lhs) F__SEQ2.simps(1)} & \$\dn\$ & ⟨Seq (f_1 v) (f_2 ())⟩\\ @{thm (lhs) F__SEQ.simps(1)} & \$\dn\$ & ⟨Seq (f_1 v_1) (f_2 v_2)⟩\medskip\\ %\end{tabular} % %\begin{tabular}{lcl} @{term simp__ALT (ZERO, DUMMY) (r_2, f_2)} & \$\dn\$ & @{term (r_2, F__RIGHT f_2)}\\ @{term simp__ALT (r_1, f_1) (ZERO, DUMMY)} & \$\dn\$ & @{term (r_1, F__LEFT f_1)}\\ @{term simp__ALT (r_1, f_1) (r_2, f_2)} & \$\dn\$ & @{term (ALT r_1 r_2, F__ALT f_1 f_2)}\\ @{term simp__SEQ (ONE, f_1) (r_2, f_2)} & \$\dn\$ & @{term (r_2, F__SEQ1 f_1 f_2)}\\ @{term simp__SEQ (r_1, f_1) (ONE, f_2)} & \$\dn\$ & @{term (r_1, F__SEQ2 f_1 f_2)}\\ @{term simp__SEQ (r_1, f_1) (r_2, f_2)} & \$\dn\$ & @{term (SEQ r_1 r_2, F__SEQ f_1 f_2)}\\ \end{tabular} \end{center} \noindent The functions ⟨$simp_{Alt}$⟩ and ⟨$simp_{Seq}$⟩ encode the simplification rules in \eqref{Simpl} and compose the rectification functions (simplifications can occur deep inside the regular expression). The main simplification function is then \begin{center} \begin{tabular}{lcl}*

@{term simp (ALT r_1 r_2)} & $\dn$ & @{term simp___ALT (simp r_1) (simp r_2)}\\ @{term simp (SEQ r_1 r_2)} & $\dn$ & @{term simp___SEQ (simp r_1) (simp r_2)}\\ @{term simp r} & $\dn$ & @{term (r, id)}\\ \end{tabular} \end{center} \noindent where @{term id} stands for the identity function. The function @{const simp} returns a simplified regular expression and a corresponding rectification function. Note that we do not simplify under stars: this seems to slow down the algorithm, rather than speed it up. The optimised lexer is then given by the clauses: \begin{center} \begin{tabular}{lcl} @{thm (lhs) slexer.simps(1)} & $\dn$ & @{thm (rhs) slexer.simps(1)}\\ @{thm (lhs) slexer.simps(2)} & $\dn$ & ⟨let (r_s, f_r) = simp (r ⟩$\backslash$⟨ c) in⟩\\ & & ⟨case⟩ @{term slexer r_s s} ⟨of⟩\\ & & @{term None} ⟨⇒⟩ @{term None}\\ & & $|$ @{term Some v} ⟨⇒⟩ ⟨Some (inj r c (f_r v))⟩ \end{tabular} \end{center} \noindent In the second clause we first calculate the derivative @{term der c r} and then simplify the result. This gives us a simplified derivative ⟨r_s⟩ and a rectification function ⟨f_r⟩. The lexer is then recursively called with the simplified derivative, but before we inject the character @{term c} into the value @{term v}, we need to rectify @{term v} (that is construct @{term f_r v}). Before we can establish the correctness of @{term slexer}, we need to show that simplification preserves the language and simplification preserves our POSIX relation once the value is rectified (recall @{const simp} generates a (regular expression, rectification function) pair): \begin{lemma}\mbox{}\smallskip\\\label{slexeraux} \begin{tabular}{ll} (1) & @{thm L___fst___simp[symmetric]}\\ (2) & @{thm[mode=IfThen] Posix___simp} \end{tabular} \end{lemma} \begin{proof} Both are by induction on ⟨r⟩. There is no interesting case for the first statement. For the second statement, of interest are the @{term r = ALT r_1 r_2} and @{term r = SEQ r_1 r_2} cases. In each case we have to analyse four subcases whether @{term fst (simp r_1)} and @{term fst (simp r_2)} equals @{const ZERO} (respectively @{const ONE}). For example for @{term r = ALT r_1 r_2}, consider the subcase @{term fst (simp r_1) = ZERO} and @{term fst (simp r_2) ≠ ZERO}. By assumption we know @{term s ∈ fst (simp (ALT r_1 r_2)) → v}. From this we can infer @{term s ∈ fst (simp r_2) → v} and by IH also (∗) @{term s ∈ r_2 → (snd (simp r_2) v)}. Given @{term fst (simp r_1) = ZERO} we know @{term L (fst (simp r_1)) = {}}. By the first statement @{term L r_1} is the empty set, meaning (∗∗) @{term s ∉ L r_1}. Taking (∗) and (∗∗) together gives by the \mbox{⟨P+R⟩}−rule @{term s ∈ ALT r_1 r_2 → Right (snd (simp r_2) v)}. In turn this gives @{term s ∈ ALT r_1 r_2 → snd (simp (ALT r_1 r_2)) v} as we need to show. The other cases are similar.\qed \end{proof} \noindent We can now prove relatively straightforwardly that the optimised lexer produces the expected result: \begin{theorem} @{thm slexer___correctness} \end{theorem} \begin{proof} By induction on @{term s} generalising over @{term r}. The case @{term []} is trivial. For the cons−case suppose the string is of the form @{term c # s}. By induction hypothesis we know @{term slexer r s = lexer r s} holds for all @{term r} (in particular for @{term r} being the derivative @{term der c r}). Let @{term r_s} be the simplified derivative regular expression, that is @{term fst (simp (der c *r))}, and @{term f_r} be the rectification function, that is @{term snd (simp (der c r))}. We distinguish the cases whether (∗) @{term s ∈ L (der c r)} or not. In the first case we have by Theorem~\ref{lexercorrect}(2) a value @{term v} so that @{term lexer (der c r) s = Some v} and @{term s ∈ der c r → v} hold. By Lemma~\ref{slexeraux}(1) we can also infer from~(∗) that @{term s ∈ L r_s} holds. Hence we know by Theorem~\ref{lexercorrect}(2) that there exists a @{term v′} with @{term lexer r_s s = Some v′} and @{term s ∈ r_s → v′}. From the latter we know by Lemma~\ref{slexeraux}(2) that @{term s ∈ der c r → (f_r v′)} holds. By the uniqueness of the POSIX relation (Theorem~\ref{posixdeterm}) we can infer that @{term v} is equal to @{term f_r v′}−−−that is the rectification function applied to @{term v′} produces the original @{term v}. Now the case follows by the definitions of @{const lexer} and @{const slexer}. In the second case where @{term s ∉ L (der c r)} we have that @{term lexer (der c r) s = None} by Theorem~\ref{lexercorrect}(1). We also know by Lemma~\ref{slexeraux}(1) that @{term s ∉ L r_s}. Hence @{term lexer r_s s = None} by Theorem~\ref{lexercorrect}(1) and by IH then also @{term slexer r_s s = None}. With this we can conclude in this case too.\qed \end{proof}* fy the result. This gives us a simplified derivative r_s and a rectification function f_r. The lexer is then recursively called with the simplified derivative, but before we inject the character c into the value v, we need to rectify v (that is construct $f_r\ v$). Before we can establish the correctness of $lexer^+$, we need to show that simplification preserves the language and simplification preserves our POSIX relation once the value is rectified (recall *simp* generates a (regular expression, rectification function) pair):

Lemma 7.
(1) L(fst (simp r)) = L(r)
(2) If (s, fst (simp r)) → v then (s, r) → snd (simp r) v.

Proof. Both are by induction on r. There is no interesting case for the first statement. For the second statement, of interest are the $r = r_1 + r_2$ and $r = r_1 \cdot r_2$ cases. In each case we have to analyse four subcases whether *fst* (*simp* r_1) and *fst* (*simp* r_2) equals **0** (respectively **1**). For example for $r = r_1 + r_2$, consider the subcase *fst* (*simp* r_1) = **0** and *fst* (*simp* r_2) ≠ **0**. By assumption we know (*s*, *fst* (*simp* $(r_1 + r_2)$)) → *v*. From this we can infer (*s*, *fst* (*simp* r_2)) → *v* and by IH also (*) (*s*, r_2) → *snd* (*simp* r_2) *v*. Given *fst* (*simp* r_1) = **0** we know *L*(*fst* (*simp* r_1)) = ∅. By the first statement $L(r_1)$ is the empty set, meaning (**) $s \notin L(r_1)$. Taking (*) and (**) together gives by the $P{+}R$-rule (*s*, $r_1 + r_2$) → *Right* (*snd* (*simp* r_2) *v*). In turn this gives (*s*, $r_1 + r_2$) → *snd* (*simp* $(r_1 + r_2)$) *v* as we need to show. The other cases are similar. □

We can now prove relatively straightforwardly that the optimised lexer produces the expected result:

Theorem 5. *$lexer^+$ r s = lexer r s*

Proof. By induction on s generalising over r. The case [] is trivial. For the cons-case suppose the string is of the form $c :: s$. By induction hypothesis we know

$\mathit{lexer}^+\ r\ s = \mathit{lexer}\ r\ s$ holds for all r (in particular for r being the derivative $r\backslash c$). Let r_s be the simplified derivative regular expression, that is $\mathit{fst}\ (\mathit{simp}\ (r\backslash c))$, and f_r be the rectification function, that is $\mathit{snd}\ (\mathit{simp}\ (r\backslash c))$. We distinguish the cases whether (*) $s \in L(r\backslash c)$ or not. In the first case we have by Theorem 2(2) a value v so that $\mathit{lexer}\ (r\backslash c)\ s = \mathit{Some}\ v$ and $(s, r\backslash c) \to v$ hold. By Lemma 7(1) we can also infer from (*) that $s \in L(r_s)$ holds. Hence we know by Theorem 2(2) that there exists a v' with $\mathit{lexer}\ r_s\ s = \mathit{Some}\ v'$ and $(s, r_s) \to v'$. From the latter we know by Lemma 7(2) that $(s, r\backslash c) \to f_r\ v'$ holds. By the uniqueness of the POSIX relation (Theorem 1) we can infer that v is equal to $f_r\ v'$—that is the rectification function applied to v' produces the original v. Now the case follows by the definitions of *lexer* and lexer^+.

In the second case where $s \notin L(r\backslash c)$ we have that $\mathit{lexer}\ (r\backslash c)\ s = \mathit{None}$ by Theorem 2(1). We also know by Lemma 7(1) that $s \notin L(r_s)$. Hence $\mathit{lexer}\ r_s\ s = \mathit{None}$ by Theorem 2(1) and by IH then also $\mathit{lexer}^+\ r_s\ s = \mathit{None}$. With this we can conclude in this case too. □

9 HERE

Lemma 8. *If* $v : (r^\downarrow)\backslash c$ *then* $\mathit{retrieve}\ (r\backslash\backslash c)\ v = \mathit{retrieve}\ r\ (\mathit{inj}\ (r^\downarrow)\ c\ v)$.

Proof. By induction on the definition of $r^\downarrow$. The cases for rule 1) and 2) are straightforward as $\mathbf{0}\backslash c$ and $\mathbf{1}\backslash c$ are both equal to $\mathbf{0}$. This means $v : \mathbf{0}$ cannot hold. Similarly in case of rule 3) where r is of the form $\mathit{ACHAR}\ d$ with $c = d$. Then by assumption we know $v : \mathbf{1}$, which implies $v = \mathit{Empty}$. The equation follows by simplification of left- and right-hand side. In case $c \neq d$ we have again $v : \mathbf{0}$, which cannot hold.

For rule 4a) we have again $v : \mathbf{0}$. The property holds by IH for rule 4b). The induction hypothesis is

$$\mathit{retrieve}\ (r\backslash\backslash c)\ v = \mathit{retrieve}\ r\ (\mathit{inj}\ (r^\downarrow)\ c\ v)$$

which is what left- and right-hand side simplify to. The slightly more interesting case is for 4c). By assumption we have $v : ((r_1{}^\downarrow)\backslash c) + (((\mathit{AALTs}\ \mathit{bs}\ (r_2 :: \mathit{rs}))^\downarrow)\backslash c)$. This means we have either (*) $\mathit{v1} : (r_1{}^\downarrow)\backslash c$ with $v = \mathit{Left}\ \mathit{v1}$ or (**) $\mathit{v2} : ((\mathit{AALTs}\ \mathit{bs}\ (r_2 :: \mathit{rs}))^\downarrow)\backslash c$ with $v = \mathit{Right}\ \mathit{v2}$. The former case is straightforward by simplification. The second case is ...TBD.

Rule 5) TBD.

Finally for rule 6) the reasoning is as follows: By assumption we have $v : ((r^\downarrow)\backslash c) \cdot (r^\downarrow)^\star$. This means we also have $v = \mathit{Seq}\ \mathit{v1}\ \mathit{v2}$, $\mathit{v1} : (r^\downarrow)\backslash c$ and $\mathit{v2} = \mathit{Stars}\ \mathit{vs}$. We want to prove

$$\mathit{retrieve}\ (\mathit{ASEQ}\ \mathit{bs}\ (\mathit{fuse}\ [Z]\ (r\backslash\backslash c))\ (\mathit{ASTAR}\ []\ r))\ v \tag{3}$$

$$= \mathit{retrieve}\ (\mathit{ASTAR}\ \mathit{bs}\ r)\ (\mathit{inj}\ ((r^\downarrow)^\star)\ c\ v) \tag{4}$$

The right-hand side *inj*-expression is equal to $\mathit{Stars}\ (\mathit{inj}\ (r^{\downarrow})\ c\ \mathit{v1} :: \mathit{vs})$, which means the *retrieve*-expression simplifies to

$$\mathit{bs}\ @\ [Z]\ @\ \mathit{retrieve}\ r\ (\mathit{inj}\ (r^{\downarrow})\ c\ \mathit{v1})\ @\ \mathit{retrieve}\ (\mathit{ASTAR}\ []\ r)\ (\mathit{Stars}\ \mathit{vs})$$

The left-hand side (3) above simplifies to

$$\mathit{bs}\ @\ \mathit{retrieve}\ (\mathit{fuse}\ [Z]\ (r\backslash\!\backslash c))\ \mathit{v1}\ @\ \mathit{retrieve}\ (\mathit{ASTAR}\ []\ r)\ (\mathit{Stars}\ \mathit{vs})$$

We can move out the *fuse* $[Z]$ and then use the IH to show that left-hand side and right-hand side are equal. This completes the proof.