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

1 List prefixes, suffixes, and homeomorphic embedding

theory *Sublist*
imports *Main*
begin

---

* This paper is a revised and expanded version of [3]. Compared with that paper we give a second definition for POSIX values introduced by Okui Suzuki [12,13] 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 [16]. 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.

1.1 Prefix order on lists

```
definition prefix :: 'a list ⇒ 'a list ⇒ bool
  where prefix xs ys ⟷ (∃ zs. ys = xs @ zs)

definition strict_prefix :: 'a list ⇒ 'a list ⇒ bool
  where strict_prefix xs ys ⟷ prefix xs ys ∧ xs ≠ ys

interpretation prefix_order: order prefix strict_prefix
  by standard (auto simp: prefix_def strict_prefix_def)

interpretation prefix_bot: order_bot Nil prefix strict_prefix
  by standard (simp add: prefix_def)

lemma prefixI [intro?]: ys = xs @ zs ⟹ prefix xs ys
  unfolding prefix_def by blast

lemma prefixE [elim?]:
  assumes prefix xs ys
  obtains zs where ys = xs @ zs
  using assms unfolding prefix_def by blast

lemma strict_prefixI' [intro?]: ys = xs @ z # zs ⟹ strict_prefix xs ys
  unfolding strict_prefix_def prefix_def by blast

lemma strict_prefixE' [elim?]:
  assumes strict_prefix xs ys
  obtains z zs where ys = xs @ z # zs
proof −
  from ‹strict_prefix xs ys› obtain us where ys = xs @ us and xs ≠ ys
    unfolding strict_prefix_def prefix_def by blast
  with that show ?thesis by (auto simp add: neq_Nil_conv)
qed


lemma strict_prefixI [intro?]: prefix xs ys ⟹ xs ≠ ys ⟹ strict_prefix xs ys
by(fact prefix_order.le_neq_trans)

lemma strict_prefixE [elim?]:
  fixes xs ys :: 'a list
  assumes strict_prefix xs ys
  obtains prefix xs ys and xs ≠ ys
  using assms unfolding strict_prefix_def by blast
```

1.2 Basic properties of prefixes

```
theorem Nil_prefix [simp]: prefix [] xs
```

```
 by (fact prefix_bot.bot_least)


theorem prefix_Nil [simp]: (prefix xs []) = (xs = [])
 by (fact prefix_bot.bot_unique)

lemma prefix_snoc [simp]: prefix xs (ys @ [y]) ⟷ xs = ys @ [y] ∨ prefix xs ys
proof
 assume prefix xs (ys @ [y])
 then obtain zs where zs: ys @ [y] = xs @ zs ..
 show xs = ys @ [y] ∨ prefix xs ys
  by (metis append_Nil2 butlast_append butlast_snoc prefixI zs)
next
 assume xs = ys @ [y] ∨ prefix xs ys
 then show prefix xs (ys @ [y])
  by (metis prefix_order.eq_iff prefix_order.order_trans prefixI)
qed

lemma Cons_prefix_Cons [simp]: prefix (x # xs) (y # ys) = (x = y ∧ prefix xs ys)
 by (auto simp add: prefix_def)

lemma prefix_code [code]:
 prefix [] xs ⟷ True
 prefix (x # xs) [] ⟷ False
 prefix (x # xs) (y # ys) ⟷ x = y ∧ prefix xs ys
 by simp_all

lemma same_prefix_prefix [simp]: prefix (xs @ ys) (xs @ zs) = prefix ys zs
 by (induct xs) simp_all

lemma same_prefix_nil [simp]: prefix (xs @ ys) xs = (ys = [])
 by (metis append_Nil2 append_self_conv prefix_order.eq_iff prefixI)

lemma prefix_prefix [simp]: prefix xs ys ⟹ prefix xs (ys @ zs)
 unfolding prefix_def by fastforce

lemma append_prefixD: prefix (xs @ ys) zs ⟹ prefix xs zs
 by (auto simp add: prefix_def)

theorem prefix_Cons: prefix xs (y # ys) = (xs = [] ∨ (∃ zs. xs = y # zs ∧ prefix zs ys))
 by (cases xs) (auto simp add: prefix_def)

theorem prefix_append:
 prefix xs (ys @ zs) = (prefix xs ys ∨ (∃ us. xs = ys @ us ∧ prefix us zs))
 apply (induct zs rule: rev_induct)
```

```
   apply force
  apply (simp flip: append_assoc)
  apply (metis append_eq_appendI)
  done

lemma append_one_prefix:
  prefix xs ys ⟹ length xs < length ys ⟹ prefix (xs @ [ys ! length xs]) ys
  proof (unfold prefix_def)
    assume a1: ∃ zs. ys = xs @ zs
    then obtain sk :: 'a list where sk: ys = xs @ sk by fastforce
    assume a2: length xs < length ys
    have f1: ⋀v. ([]::'a list) @ v = v using append_Nil2 by simp
    have [] ≠ sk using a1 a2 sk less_not_refl by force
    hence ∃ v. xs @ hd sk # v = ys using sk by (metis hd_Cons_tl)
    thus ∃ zs. ys = (xs @ [ys ! length xs]) @ zs using f1 by fastforce
  qed

theorem prefix_length_le: prefix xs ys ⟹ length xs ≤ length ys
  by (auto simp add: prefix_def)

lemma prefix_same_cases:
  prefix (xs_1::'a list) ys ⟹ prefix xs_2 ys ⟹ prefix xs_1 xs_2 ∨ prefix xs_2 xs_1
  unfolding prefix_def by (force simp: append_eq_append_conv2)

lemma prefix_length_prefix:
  prefix ps xs ⟹ prefix qs xs ⟹ length ps ≤ length qs ⟹ prefix ps qs
by (auto simp: prefix_def) (metis append_Nil2 append_eq_append_conv_if)

lemma set_mono_prefix: prefix xs ys ⟹ set xs ⊆ set ys
  by (auto simp add: prefix_def)

lemma take_is_prefix: prefix (take n xs) xs
  unfolding prefix_def by (metis append_take_drop_id)

lemma prefixeq_butlast: prefix (butlast xs) xs
by (simp add: butlast_conv_take take_is_prefix)

lemma map_mono_prefix: prefix xs ys ⟹ prefix (map f xs) (map f ys)
by (auto simp: prefix_def)

lemma filter_mono_prefix: prefix xs ys ⟹ prefix (filter P xs) (filter P ys)
by (auto simp: prefix_def)

lemma sorted_antimono_prefix: prefix xs ys ⟹ sorted ys ⟹ sorted xs
by (metis sorted_append prefix_def)
```

```
lemma prefix_length_less: strict_prefix xs ys ⟹ length xs < length ys
  by (auto simp: strict_prefix_def prefix_def)

lemma prefix_snocD: prefix (xs@[x]) ys ⟹ strict_prefix xs ys
  by (simp add: strict_prefixI' prefix_order.dual_order.strict_trans1)

lemma strict_prefix_simps [simp, code]:
  strict_prefix xs [] ⟷ False
  strict_prefix [] (x # xs) ⟷ True
  strict_prefix (x # xs) (y # ys) ⟷ x = y ∧ strict_prefix xs ys
  by (simp_all add: strict_prefix_def cong: conj_cong)

lemma take_strict_prefix: strict_prefix xs ys ⟹ strict_prefix (take n xs) ys
proof (induct n arbitrary: xs ys)
  case 0
  then show ?case by (cases ys) simp_all
next
  case (Suc n)
  then show ?case by (metis prefix_order.less_trans strict_prefixI take_is_prefix)
qed

lemma not_prefix_cases:
  assumes pfx: ¬ prefix ps ls
  obtains
    (c1) ps ≠ [] and ls = []
  | (c2) a as x xs where ps = a#as and ls = x#xs and x = a and ¬ prefix as xs
  | (c3) a as x xs where ps = a#as and ls = x#xs and x ≠ a
proof (cases ps)
  case Nil
  then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ‹ps = a#as›
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append_Nil2 pfx c1 same_prefix_nil)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ prefix as xs using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
```

```
   case False
   with c Cons show ?thesis by (rule c3)
  qed
 qed
qed

lemma not_prefix_induct [consumes 1, case_names Nil Neq Eq]:
 assumes np: ¬ prefix ps ls
  and base: ⋀x xs. P (x#xs) []
  and r1: ⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)
  and r2: ⋀x xs y ys. ⟦ x = y; ¬ prefix xs ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
 shows P ps ls using np
proof (induct ls arbitrary: ps)
 case Nil
 then show ?case
  by (auto simp: neq_Nil_conv elim!: not_prefix_cases intro!: base)
next
 case (Cons y ys)
 then have npfx: ¬ prefix ps (y # ys) by simp
 then obtain x xs where pv: ps = x # xs
  by (rule not_prefix_cases) auto
 show ?case by (metis Cons.hyps Cons_prefix_Cons npfx pv r1 r2)
qed
```

1.3 Prefixes

```
primrec prefixes where
prefixes [] = [[]] |
prefixes (x#xs) = [] # map ((#) x) (prefixes xs)

lemma in_set_prefixes[simp]: xs ∈ set (prefixes ys) ⟷ prefix xs ys
proof (induct xs arbitrary: ys)
 case Nil
 then show ?case by (cases ys) auto
next
 case (Cons a xs)
 then show ?case by (cases ys) auto
qed

lemma length_prefixes[simp]: length (prefixes xs) = length xs+1
 by (induction xs) auto

lemma distinct_prefixes [intro]: distinct (prefixes xs)
 by (induction xs) (auto simp: distinct_map)

lemma prefixes_snoc [simp]: prefixes (xs@[x]) = prefixes xs @ [xs@[x]]
```

```
  by (induction xs) auto

lemma prefixes_not_Nil [simp]: prefixes xs ≠ []
  by (cases xs) auto

lemma hd_prefixes [simp]: hd (prefixes xs) = []
  by (cases xs) simp_all

lemma last_prefixes [simp]: last (prefixes xs) = xs
  by (induction xs) (simp_all add: last_map)

lemma prefixes_append:
  prefixes (xs @ ys) = prefixes xs @ map (λys'. xs @ ys') (tl (prefixes ys))
proof (induction xs)
  case Nil
  thus ?case by (cases ys) auto
qed simp_all

lemma prefixes_eq_snoc:
  prefixes ys = xs @ [x] ⟷
  (ys = [] ∧ xs = [] ∨ (∃z zs. ys = zs@[z] ∧ xs = prefixes zs)) ∧ x = ys
  by (cases ys rule: rev_cases) auto

lemma prefixes_tailrec [code]:
  prefixes xs = rev (snd (foldl (λ(acc1, acc2) x. (x#acc1, rev (x#acc1)#acc2)) ([],[[]])
xs))
proof −
  have foldl (λ(acc1, acc2) x. (x#acc1, rev (x#acc1)#acc2)) (ys, rev ys # zs) xs =
        (rev xs @ ys, rev (map (λas. rev ys @ as) (prefixes xs)) @ zs) for ys zs
  proof (induction xs arbitrary: ys zs)
    case (Cons x xs ys zs)
    from Cons.IH[of x # ys rev ys # zs]
      show ?case by (simp add: o_def)
  qed simp_all
  from this [of [] []] show ?thesis by simp
qed

lemma set_prefixes_eq: set (prefixes xs) = {ys. prefix ys xs}
  by auto

lemma card_set_prefixes [simp]: card (set (prefixes xs)) = Suc (length xs)
  by (subst distinct_card) auto

lemma set_prefixes_append:
  set (prefixes (xs @ ys)) = set (prefixes xs) ∪ {xs @ ys' |ys'. ys' ∈ set (prefixes ys)}
```

```
 by (subst prefixes_append, cases ys) auto
```

1.4 Longest Common Prefix

```
definition Longest_common_prefix :: 'a list set ⇒ 'a list where
Longest_common_prefix L = (ARG_MAX length ps. ∀xs ∈ L. prefix ps xs)

lemma Longest_common_prefix_ex: L ≠ {} ⟹
 ∃ps. (∀xs ∈ L. prefix ps xs) ∧ (∀qs. (∀xs ∈ L. prefix qs xs) ⟶ size qs ≤ size ps)
 (is _ ⟹ ∃ps. ?P L ps)
proof(induction LEAST n. ∃xs ∈L. n = length xs arbitrary: L)
 case 0
 have [] ∈ L using 0.hyps LeastI[of λn. ∃xs∈L. n = length xs] ‹L ≠ {}›
  by auto
 hence ?P L [] by(auto)
 thus ?case ..
next
 case (Suc n)
 let ?EX = λn. ∃xs∈L. n = length xs
 obtain x xs where xxs: x#xs ∈ L size xs = n using Suc.prems Suc.hyps(2)
  by(metis LeastI_ex[of ?EX] Suc_length_conv ex_in_conv)
 hence [] ∉ L using Suc.hyps(2) by auto
 show ?case
 proof (cases ∀xs ∈ L. ∃ys. xs = x#ys)
  case True
  let ?L = {ys. x#ys ∈ L}
  have 1: (LEAST n. ∃xs ∈ ?L. n = length xs) = n
   using xxs Suc.prems Suc.hyps(2) Least_le[of ?EX]
   by − (rule Least_equality, fastforce+)
  have 2: ?L ≠ {} using ‹x # xs ∈ L› by auto
  from Suc.hyps(1)[OF 1[symmetric] 2] obtain ps where IH: ?P ?L ps ..
  { fix qs
   assume ∀qs. (∀xa. x # xa ∈ L ⟶ prefix qs xa) ⟶ length qs ≤ length ps
   and ∀xs∈L. prefix qs xs
   hence length (tl qs) ≤ length ps
    by (metis Cons_prefix_Cons hd_Cons_tl list.sel(2) Nil_prefix)
   hence length qs ≤ Suc (length ps) by auto
  }
  hence ?P L (x#ps) using True IH by auto
  thus ?thesis ..
 next
  case False
  then obtain y ys where yys: x≠y y#ys ∈ L using ‹[] ∉ L›
   by (auto) (metis list.exhaust)
  have ∀qs. (∀xs∈L. prefix qs xs) ⟶ qs = [] using yys ‹x#xs ∈ L›
   by auto (metis Cons_prefix_Cons prefix_Cons)
```

```
    hence ?P L [] by auto
    thus ?thesis ..
  qed
qed

lemma Longest_common_prefix_unique: L ≠ {} ⟹
  ∃! ps. (∀xs ∈ L. prefix ps xs) ∧ (∀qs. (∀xs ∈ L. prefix qs xs) ⟶ size qs ≤ size ps)
by(rule ex_ex1I[OF Longest_common_prefix_ex];
   meson equals0I prefix_length_prefix prefix_order.antisym)

lemma Longest_common_prefix_eq:
 ⟦ L ≠ {}; ∀xs ∈ L. prefix ps xs;
   ∀qs. (∀xs ∈ L. prefix qs xs) ⟶ size qs ≤ size ps ⟧
  ⟹ Longest_common_prefix L = ps
unfolding Longest_common_prefix_def arg_max_def is_arg_max_linorder
by(rule some1_equality[OF Longest_common_prefix_unique]) auto

lemma Longest_common_prefix_prefix:
  xs ∈ L ⟹ prefix (Longest_common_prefix L) xs
unfolding Longest_common_prefix_def arg_max_def is_arg_max_linorder
by(rule someI2_ex[OF Longest_common_prefix_ex]) auto

lemma Longest_common_prefix_longest:
  L ≠ {} ⟹ ∀xs∈L. prefix ps xs ⟹ length ps ≤ length(Longest_common_prefix L)
unfolding Longest_common_prefix_def arg_max_def is_arg_max_linorder
by(rule someI2_ex[OF Longest_common_prefix_ex]) auto

lemma Longest_common_prefix_max_prefix:
  L ≠ {} ⟹ ∀xs∈L. prefix ps xs ⟹ prefix ps (Longest_common_prefix L)
by(metis Longest_common_prefix_prefix Longest_common_prefix_longest
     prefix_length_prefix ex_in_conv)

lemma Longest_common_prefix_Nil: [] ∈ L ⟹ Longest_common_prefix L = []
using Longest_common_prefix_prefix prefix_Nil by blast

lemma Longest_common_prefix_image_Cons: L ≠ {} ⟹
  Longest_common_prefix ((#) x ` L) = x # Longest_common_prefix L
apply(rule Longest_common_prefix_eq)
  apply(simp)
 apply (simp add: Longest_common_prefix_prefix)
apply simp
by(metis Longest_common_prefix_longest[of L] Cons_prefix_Cons Nitpick.size_list_simp(2)
    Suc_le_mono hd_Cons_tl order.strict_implies_order zero_less_Suc)

lemma Longest_common_prefix_eq_Cons: assumes L ≠ {} [] ∉ L ∀xs∈L. hd xs = x
```

```
shows Longest_common_prefix L = x # Longest_common_prefix {ys. x#ys ∈ L}
proof -
  have L = (#) x ‘ {ys. x#ys ∈ L} using assms(2,3)
    by (auto simp: image_def)(metis hd_Cons_tl)
  thus ?thesis
    by (metis Longest_common_prefix_image_Cons image_is_empty assms(1))
qed

lemma Longest_common_prefix_eq_Nil:
  ⟦x#ys ∈ L; y#zs ∈ L; x ≠ y ⟧ ⟹ Longest_common_prefix L = []
by (metis Longest_common_prefix_prefix list.inject prefix_Cons)


fun longest_common_prefix :: 'a list ⇒ 'a list ⇒ 'a list where
longest_common_prefix (x#xs) (y#ys) =
  (if x=y then x # longest_common_prefix xs ys else []) |
longest_common_prefix _ _ = []

lemma longest_common_prefix_prefix1:
  prefix (longest_common_prefix xs ys) xs
by(induction xs ys rule: longest_common_prefix.induct) auto

lemma longest_common_prefix_prefix2:
  prefix (longest_common_prefix xs ys) ys
by(induction xs ys rule: longest_common_prefix.induct) auto

lemma longest_common_prefix_max_prefix:
  ⟦ prefix ps xs; prefix ps ys ⟧
   ⟹ prefix ps (longest_common_prefix xs ys)
by(induction xs ys arbitrary: ps rule: longest_common_prefix.induct)
  (auto simp: prefix_Cons)
```

1.5 Parallel lists

```
definition parallel :: 'a list ⇒ 'a list ⇒ bool  (infixl ∥ 50)
  where (xs ∥ ys) = (¬ prefix xs ys ∧ ¬ prefix ys xs)

lemma parallelI [intro]: ¬ prefix xs ys ⟹ ¬ prefix ys xs ⟹ xs ∥ ys
  unfolding parallel_def by blast

lemma parallelE [elim]:
  assumes xs ∥ ys
  obtains ¬ prefix xs ys ∧ ¬ prefix ys xs
  using assms unfolding parallel_def by blast

theorem prefix_cases:
```

```
  obtains prefix xs ys | strict_prefix ys xs | xs ∥ ys
  unfolding parallel_def strict_prefix_def by blast

theorem parallel_decomp:
  xs ∥ ys ⟹ ∃as b bs c cs. b ≠ c ∧ xs = as @ b # bs ∧ ys = as @ c # cs
proof (induct xs rule: rev_induct)
  case Nil
  then have False by auto
  then show ?case ..
next
  case (snoc x xs)
  show ?case
  proof (rule prefix_cases)
    assume le: prefix xs ys
    then obtain ys' where ys: ys = xs @ ys' ..
    show ?thesis
    proof (cases ys')
      assume ys' = []
      then show ?thesis by (metis append_Nil2 parallelE prefixI snoc.prems ys)
    next
      fix c cs assume ys': ys' = c # cs
      have x ≠ c using snoc.prems ys ys' by fastforce
      thus ∃as b bs c cs. b ≠ c ∧ xs @ [x] = as @ b # bs ∧ ys = as @ c # cs
        using ys ys' by blast
    qed
  next
    assume strict_prefix ys xs
    then have prefix ys (xs @ [x]) by (simp add: strict_prefix_def)
    with snoc have False by blast
    then show ?thesis ..
  next
    assume xs ∥ ys
    with snoc obtain as b bs c cs where neq: (b::'a) ≠ c
      and xs: xs = as @ b # bs and ys: ys = as @ c # cs
      by blast
    from xs have xs @ [x] = as @ b # (bs @ [x]) by simp
    with neq ys show ?thesis by blast
  qed
qed

lemma parallel_append: a ∥ b ⟹ a @ c ∥ b @ d
  apply (rule parallelI)
    apply (erule parallelE, erule conjE,
      induct rule: not_prefix_induct, simp+)+
  done
```

lemma *parallel_appendI*: $xs \parallel ys \Longrightarrow x = xs \mathbin{@} xs' \Longrightarrow y = ys \mathbin{@} ys' \Longrightarrow x \parallel y$
by (*simp add*: *parallel_append*)

lemma *parallel_commute*: $a \parallel b \longleftrightarrow b \parallel a$
unfolding *parallel_def* **by** *auto*

1.6 Suffix order on lists

definition *suffix* :: $'a\ list \Rightarrow {'a}\ list \Rightarrow bool$
where *suffix xs ys* $= (\exists zs.\ ys = zs \mathbin{@} xs)$

definition *strict_suffix* :: $'a\ list \Rightarrow {'a}\ list \Rightarrow bool$
where *strict_suffix xs ys* $\longleftrightarrow$ *suffix xs ys* $\wedge\ xs \neq ys$

interpretation *suffix_order*: *order suffix strict_suffix*
by *standard* (*auto simp*: *suffix_def strict_suffix_def*)

interpretation *suffix_bot*: *order_bot Nil suffix strict_suffix*
by *standard* (*simp add*: *suffix_def*)

lemma *suffixI* [*intro?*]: $ys = zs \mathbin{@} xs \Longrightarrow$ *suffix xs ys*
unfolding *suffix_def* **by** *blast*

lemma *suffixE* [*elim?*]:
assumes *suffix xs ys*
obtains *zs* **where** $ys = zs \mathbin{@} xs$
using *assms* **unfolding** *suffix_def* **by** *blast*

lemma *suffix_tl* [*simp*]: *suffix* (*tl xs*) *xs*
by (*induct xs*) (*auto simp*: *suffix_def*)

lemma *strict_suffix_tl* [*simp*]: $xs \neq [] \Longrightarrow$ *strict_suffix* (*tl xs*) *xs*
by (*induct xs*) (*auto simp*: *strict_suffix_def suffix_def*)

lemma *Nil_suffix* [*simp*]: *suffix* [] *xs*
by (*simp add*: *suffix_def*)

lemma *suffix_Nil* [*simp*]: (*suffix xs* []) $=$ ($xs = []$)
by (*auto simp add*: *suffix_def*)

lemma *suffix_ConsI*: *suffix xs ys* $\Longrightarrow$ *suffix xs* ($y \# ys$)
by (*auto simp add*: *suffix_def*)

lemma *suffix_ConsD*: *suffix* ($x \# xs$) *ys* $\Longrightarrow$ *suffix xs ys*
by (*auto simp add*: *suffix_def*)

```
lemma suffix_appendI: suffix xs ys ⟹ suffix xs (zs @ ys)
  by (auto simp add: suffix_def)

lemma suffix_appendD: suffix (zs @ xs) ys ⟹ suffix xs ys
  by (auto simp add: suffix_def)

lemma strict_suffix_set_subset: strict_suffix xs ys ⟹ set xs ⊆ set ys
  by (auto simp: strict_suffix_def suffix_def)

lemma set_mono_suffix: suffix xs ys ⟹ set xs ⊆ set ys
by (auto simp: suffix_def)

lemma sorted_antimono_suffix: suffix xs ys ⟹ sorted ys ⟹ sorted xs
by (metis sorted_append suffix_def)

lemma suffix_ConsD2: suffix (x # xs) (y # ys) ⟹ suffix xs ys
proof −
  assume suffix (x # xs) (y # ys)
  then obtain zs where y # ys = zs @ x # xs ..
  then show ?thesis
    by (induct zs) (auto intro!: suffix_appendI suffix_ConsI)
qed

lemma suffix_to_prefix [code]: suffix xs ys ⟷ prefix (rev xs) (rev ys)
proof
  assume suffix xs ys
  then obtain zs where ys = zs @ xs ..
  then have rev ys = rev xs @ rev zs by simp
  then show prefix (rev xs) (rev ys) ..
next
  assume prefix (rev xs) (rev ys)
  then obtain zs where rev ys = rev xs @ zs ..
  then have rev (rev ys) = rev zs @ rev (rev xs) by simp
  then have ys = rev zs @ xs by simp
  then show suffix xs ys ..
qed

lemma strict_suffix_to_prefix [code]: strict_suffix xs ys ⟷ strict_prefix (rev xs) (rev
ys)
  by (auto simp: suffix_to_prefix strict_suffix_def strict_prefix_def)

lemma distinct_suffix: distinct ys ⟹ suffix xs ys ⟹ distinct xs
  by (clarsimp elim!: suffixE)
```

lemma *map_mono_suffix*: *suffix xs ys* $\Longrightarrow$ *suffix* (*map f xs*) (*map f ys*)
by (*auto elim*!: *suffixE intro*: *suffixI*)

lemma *filter_mono_suffix*: *suffix xs ys* $\Longrightarrow$ *suffix* (*filter P xs*) (*filter P ys*)
by (*auto simp*: *suffix_def*)

lemma *suffix_drop*: *suffix* (*drop n as*) *as*
unfolding *suffix_def* **by** (*rule exI* [**where** $x =$ *take n as*]) *simp*

lemma *suffix_take*: *suffix xs ys* $\Longrightarrow$ *ys* = *take* (*length ys* $-$ *length xs*) *ys* @ *xs*
by (*auto elim*!: *suffixE*)

lemma *strict_suffix_reflclp_conv*: *strict_suffix*$^{==}$ = *suffix*
by (*intro ext*) (*auto simp*: *suffix_def strict_suffix_def*)

lemma *suffix_lists*: *suffix xs ys* $\Longrightarrow$ *ys* $\in$ *lists A* $\Longrightarrow$ *xs* $\in$ *lists A*
unfolding *suffix_def* **by** *auto*

lemma *suffix_snoc* [*simp*]: *suffix xs* (*ys* @ [*y*]) $\longleftrightarrow$ *xs* = [] $\vee$ ($\exists$ *zs*. *xs* = *zs* @ [*y*] $\wedge$ *suffix zs ys*)
by (*cases xs rule*: *rev_cases*) (*auto simp*: *suffix_def*)

lemma *snoc_suffix_snoc* [*simp*]: *suffix* (*xs* @ [*x*]) (*ys* @ [*y*]) = ($x = y$ $\wedge$ *suffix xs ys*)
by (*auto simp add*: *suffix_def*)

lemma *same_suffix_suffix* [*simp*]: *suffix* (*ys* @ *xs*) (*zs* @ *xs*) = *suffix ys zs*
by (*simp add*: *suffix_to_prefix*)

lemma *same_suffix_nil* [*simp*]: *suffix* (*ys* @ *xs*) *xs* = (*ys* = [])
by (*simp add*: *suffix_to_prefix*)

theorem *suffix_Cons*: *suffix xs* (*y* # *ys*) $\longleftrightarrow$ *xs* = *y* # *ys* $\vee$ *suffix xs ys*
unfolding *suffix_def* **by** (*auto simp*: *Cons_eq_append_conv*)

theorem *suffix_append*:
suffix xs (*ys* @ *zs*) $\longleftrightarrow$ *suffix xs zs* $\vee$ ($\exists$ *xs'*. *xs* = *xs'* @ *zs* $\wedge$ *suffix xs' ys*)
by (*auto simp*: *suffix_def append_eq_append_conv2*)

theorem *suffix_length_le*: *suffix xs ys* $\Longrightarrow$ *length xs* $\leq$ *length ys*
by (*auto simp add*: *suffix_def*)

lemma *suffix_same_cases*:
suffix (xs_1::'*a list*) *ys* $\Longrightarrow$ *suffix* xs_2 *ys* $\Longrightarrow$ *suffix* xs_1 xs_2 $\vee$ *suffix* xs_2 xs_1
unfolding *suffix_def* **by** (*force simp*: *append_eq_append_conv2*)

```
lemma suffix_length_suffix:
  suffix ps xs ⟹ suffix qs xs ⟹ length ps ≤ length qs ⟹ suffix ps qs
  by (auto simp: suffix_to_prefix intro: prefix_length_prefix)

lemma suffix_length_less: strict_suffix xs ys ⟹ length xs < length ys
  by (auto simp: strict_suffix_def suffix_def)

lemma suffix_ConsD': suffix (x#xs) ys ⟹ strict_suffix xs ys
  by (auto simp: strict_suffix_def suffix_def)

lemma drop_strict_suffix: strict_suffix xs ys ⟹ strict_suffix (drop n xs) ys
proof (induct n arbitrary: xs ys)
  case 0
  then show ?case by (cases ys) simp_all
next
  case (Suc n)
  then show ?case
    by (cases xs) (auto intro: Suc dest: suffix_ConsD' suffix_order.less_imp_le)
qed

lemma not_suffix_cases:
  assumes pfx: ¬ suffix ps ls
  obtains
    (c1) ps ≠ [] and ls = []
  | (c2) a as x xs where ps = as@[a] and ls = xs@[x] and x = a and ¬ suffix as xs
  | (c3) a as x xs where ps = as@[a] and ls = xs@[x] and x ≠ a
proof (cases ps rule: rev_cases)
  case Nil
  then show ?thesis using pfx by simp
next
  case (snoc as a)
  note c = ⟨ps = as@[a]⟩
  show ?thesis
  proof (cases ls rule: rev_cases)
    case Nil then show ?thesis by (metis append_Nil2 pfx c1 same_suffix_nil)
  next
    case (snoc xs x)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ suffix as xs using pfx c snoc True by simp
      with c snoc True show ?thesis by (rule c2)
    next
      case False
      with c snoc show ?thesis by (rule c3)
```

```
   qed
  qed
qed

lemma not_suffix_induct [consumes 1, case_names Nil Neq Eq]:
  assumes np: ¬ suffix ps ls
   and base: ⋀x xs. P (xs@[x]) []
   and r1: ⋀x xs y ys. x ≠ y ⟹ P (xs@[x]) (ys@[y])
   and r2: ⋀x xs y ys. ⟦ x = y; ¬ suffix xs ys; P xs ys ⟧ ⟹ P (xs@[x]) (ys@[y])
  shows P ps ls using np
proof (induct ls arbitrary: ps rule: rev_induct)
  case Nil
  then show ?case by (cases ps rule: rev_cases) (auto intro: base)
next
  case (snoc y ys ps)
  then have npfx: ¬ suffix ps (ys @ [y]) by simp
  then obtain x xs where pv: ps = xs @ [x]
   by (rule not_suffix_cases) auto
  show ?case by (metis snoc.hyps snoc_suffix_snoc npfx pv r1 r2)
qed


lemma parallelD1: x ∥ y ⟹ ¬ prefix x y
  by blast

lemma parallelD2: x ∥ y ⟹ ¬ prefix y x
  by blast

lemma parallel_Nil1 [simp]: ¬ x ∥ []
  unfolding parallel_def by simp

lemma parallel_Nil2 [simp]: ¬ [] ∥ x
  unfolding parallel_def by simp

lemma Cons_parallelI1: a ≠ b ⟹ a # as ∥ b # bs
  by auto

lemma Cons_parallelI2: ⟦ a = b; as ∥ bs ⟧ ⟹ a # as ∥ b # bs
  by (metis Cons_prefix_Cons parallelE parallelI)

lemma not_equal_is_parallel:
  assumes neq: xs ≠ ys
   and len: length xs = length ys
  shows xs ∥ ys
  using len neq
```

proof (*induct rule*: *list_induct2*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a as b bs*)
  have *ih*: $as \neq bs \Longrightarrow as \parallel bs$ **by** *fact*
  show *?case*
  proof (*cases* $a = b$)
    case *True*
    then have $as \neq bs$ **using** *Cons* **by** *simp*
    then show *?thesis* **by** (*rule Cons_parallelI2* [*OF True ih*])
  next
    case *False*
    then show *?thesis* **by** (*rule Cons_parallelI1*)
  qed
qed

1.7 Suffixes

primrec *suffixes* **where**
  suffixes [] = [[]]
| *suffixes* (*x#xs*) = *suffixes xs* @ [*x* # *xs*]

lemma *in_set_suffixes* [*simp*]: *xs* $\in$ *set* (*suffixes ys*) $\longleftrightarrow$ *suffix xs ys*
  by (*induction ys*) (*auto simp*: *suffix_def Cons_eq_append_conv*)

lemma *distinct_suffixes* [*intro*]: *distinct* (*suffixes xs*)
  by (*induction xs*) (*auto simp*: *suffix_def*)

lemma *length_suffixes* [*simp*]: *length* (*suffixes xs*) = *Suc* (*length xs*)
  by (*induction xs*) *auto*

lemma *suffixes_snoc* [*simp*]: *suffixes* (*xs* @ [*x*]) = [] # *map* (λ*ys*. *ys* @ [*x*]) (*suffixes xs*)
  by (*induction xs*) *auto*

lemma *suffixes_not_Nil* [*simp*]: *suffixes xs* $\neq$ []
  by (*cases xs*) *auto*

lemma *hd_suffixes* [*simp*]: *hd* (*suffixes xs*) = []
  by (*induction xs*) *simp_all*

lemma *last_suffixes* [*simp*]: *last* (*suffixes xs*) = *xs*
  by (*cases xs*) *simp_all*

lemma *suffixes_append*:
  suffixes (*xs* @ *ys*) = *suffixes ys* @ *map* (λ*xs'*. *xs'* @ *ys*) (*tl* (*suffixes xs*))

```
proof (induction ys rule: rev_induct)
 case Nil
 thus ?case by (cases xs rule: rev_cases) auto
next
 case (snoc y ys)
 show ?case
   by (simp only: append.assoc [symmetric] suffixes_snoc snoc.IH) simp
qed

lemma suffixes_eq_snoc:
 suffixes ys = xs @ [x] ⟷
   (ys = [] ∧ xs = [] ∨ (∃z zs. ys = z#zs ∧ xs = suffixes zs)) ∧ x = ys
 by (cases ys) auto

lemma suffixes_tailrec [code]:
 suffixes xs = rev (snd (foldl (λ(acc1, acc2) x. (x#acc1, (x#acc1)#acc2)) ([],[[]])
(rev xs)))
proof −
 have foldl (λ(acc1, acc2) x. (x#acc1, (x#acc1)#acc2)) (ys, ys # zs) (rev xs) =
       (xs @ ys, rev (map (λas. as @ ys) (suffixes xs)) @ zs) for ys zs
 proof (induction xs arbitrary: ys zs)
  case (Cons x xs ys zs)
  from Cons.IH[of ys zs]
    show ?case by (simp add: o_def case_prod_unfold)
 qed simp_all
 from this [of [] []] show ?thesis by simp
qed

lemma set_suffixes_eq: set (suffixes xs) = {ys. suffix ys xs}
 by auto

lemma card_set_suffixes [simp]: card (set (suffixes xs)) = Suc (length xs)
 by (subst distinct_card) auto

lemma set_suffixes_append:
 set (suffixes (xs @ ys)) = set (suffixes ys) ∪ {xs' @ ys |xs'. xs' ∈ set (suffixes xs)}
 by (subst suffixes_append, cases xs rule: rev_cases) auto


lemma suffixes_conv_prefixes: suffixes xs = map rev (prefixes (rev xs))
 by (induction xs) auto

lemma prefixes_conv_suffixes: prefixes xs = map rev (suffixes (rev xs))
 by (induction xs) auto
```

```
lemma prefixes_rev: prefixes (rev xs) = map rev (suffixes xs)
  by (induction xs) auto

lemma suffixes_rev: suffixes (rev xs) = map rev (prefixes xs)
  by (induction xs) auto
```

1.8 Homeomorphic embedding on lists

```
inductive list_emb :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool
  for P :: ('a ⇒ 'a ⇒ bool)
where
  list_emb_Nil [intro, simp]: list_emb P [] ys
| list_emb_Cons [intro] : list_emb P xs ys ⟹ list_emb P xs (y#ys)
| list_emb_Cons2 [intro]: P x y ⟹ list_emb P xs ys ⟹ list_emb P (x#xs) (y#ys)

lemma list_emb_mono:
  assumes ⋀x y. P x y ⟶ Q x y
  shows list_emb P xs ys ⟶ list_emb Q xs ys
proof
  assume list_emb P xs ys
  then show list_emb Q xs ys by (induct) (auto simp: assms)
qed

lemma list_emb_Nil2 [simp]:
  assumes list_emb P xs [] shows xs = []
  using assms by (cases rule: list_emb.cases) auto

lemma list_emb_refl:
  assumes ⋀x. x ∈ set xs ⟹ P x x
  shows list_emb P xs xs
  using assms by (induct xs) auto

lemma list_emb_Cons_Nil [simp]: list_emb P (x#xs) [] = False
proof –
  { assume list_emb P (x#xs) []
    from list_emb_Nil2 [OF this] have False by simp
  } moreover {
    assume False
    then have list_emb P (x#xs) [] by simp
  } ultimately show ?thesis by blast
qed

lemma list_emb_append2 [intro]: list_emb P xs ys ⟹ list_emb P xs (zs @ ys)
  by (induct zs) auto

lemma list_emb_prefix [intro]:
```

```
  assumes list_emb P xs ys shows list_emb P xs (ys @ zs)
  using assms
  by (induct arbitrary: zs) auto

lemma list_emb_ConsD:
  assumes list_emb P (x#xs) ys
  shows ∃ us v vs. ys = us @ v # vs ∧ P x v ∧ list_emb P xs vs
using assms
proof (induct x ≝ x # xs ys arbitrary: x xs)
  case list_emb_Cons
  then show ?case by (metis append_Cons)
next
  case (list_emb_Cons2 x y xs ys)
  then show ?case by blast
qed

lemma list_emb_appendD:
  assumes list_emb P (xs @ ys) zs
  shows ∃ us vs. zs = us @ vs ∧ list_emb P xs us ∧ list_emb P ys vs
using assms
proof (induction xs arbitrary: ys zs)
  case Nil then show ?case by auto
next
  case (Cons x xs)
  then obtain us v vs where
    zs: zs = us @ v # vs and p: P x v and lh: list_emb P (xs @ ys) vs
    by (auto dest: list_emb_ConsD)
  obtain sk0 :: 'a list ⇒ 'a list ⇒ 'a list and sk1 :: 'a list ⇒ 'a list ⇒ 'a list where
    sk: ∀ x0 x1. ¬ list_emb P (xs @ x0) x1 ∨ sk0 x0 x1 @ sk1 x0 x1 = x1 ∧ list_emb P xs
(sk0 x0 x1) ∧ list_emb P x0 (sk1 x0 x1)
    using Cons(1) by (metis (no_types))
  hence ∀ x2. list_emb P (x # xs) (x2 @ v # sk0 ys vs) using p lh by auto
  thus ?case using lh zs sk by (metis (no_types) append_Cons append_assoc)
qed

lemma list_emb_strict_suffix:
  assumes list_emb P xs ys and strict_suffix ys zs
  shows list_emb P xs zs
 using assms(2) and list_emb_append2 [OF assms(1)] by (auto simp: strict_suffix_def
suffix_def)

lemma list_emb_suffix:
  assumes list_emb P xs ys and suffix ys zs
  shows list_emb P xs zs
using assms and list_emb_strict_suffix
```

```
unfolding strict_suffix_reflclp_conv[symmetric] by auto

lemma list_emb_length: list_emb P xs ys ⟹ length xs ≤ length ys
  by (induct rule: list_emb.induct) auto

lemma list_emb_trans:
  assumes ⋀x y z. ⟦x ∈ set xs; y ∈ set ys; z ∈ set zs; P x y; P y z⟧ ⟹ P x z
  shows ⟦list_emb P xs ys; list_emb P ys zs⟧ ⟹ list_emb P xs zs
proof –
  assume list_emb P xs ys and list_emb P ys zs
  then show list_emb P xs zs using assms
  proof (induction arbitrary: zs)
    case list_emb_Nil show ?case by blast
  next
    case (list_emb_Cons xs ys y)
    from list_emb_ConsD [OF ‹list_emb P (y#ys) zs›] obtain us v vs
      where zs: zs = us @ v # vs and P== y v and list_emb P ys vs by blast
    then have list_emb P ys (v#vs) by blast
    then have list_emb P ys zs unfolding zs by (rule list_emb_append2)
    from list_emb_Cons.IH [OF this] and list_emb_Cons.prems show ?case by auto
  next
    case (list_emb_Cons2 x y xs ys)
    from list_emb_ConsD [OF ‹list_emb P (y#ys) zs›] obtain us v vs
      where zs: zs = us @ v # vs and P y v and list_emb P ys vs by blast
    with list_emb_Cons2 have list_emb P xs vs by auto
    moreover have P x v
    proof –
      from zs have v ∈ set zs by auto
      moreover have x ∈ set (x#xs) and y ∈ set (y#ys) by simp_all
      ultimately show ?thesis
        using ‹P x y› and ‹P y v› and list_emb_Cons2
        by blast
    qed
    ultimately have list_emb P (x#xs) (v#vs) by blast
    then show ?case unfolding zs by (rule list_emb_append2)
  qed
qed

lemma list_emb_set:
  assumes list_emb P xs ys and x ∈ set xs
  obtains y where y ∈ set ys and P x y
  using assms by (induct) auto

lemma list_emb_Cons_iff1 [simp]:
  assumes P x y
```

```
  shows  list_emb P (x#xs) (y#ys) ⟷ list_emb P xs ys
  using assms by (subst list_emb.simps) (auto dest: list_emb_ConsD)

lemma list_emb_Cons_iff2 [simp]:
  assumes ¬P x y
  shows  list_emb P (x#xs) (y#ys) ⟷ list_emb P (x#xs) ys
  using assms by (subst list_emb.simps) auto

lemma list_emb_code [code]:
  list_emb P [] ys ⟷ True
  list_emb P (x#xs) [] ⟷ False
  list_emb P (x#xs) (y#ys) ⟷ (if P x y then list_emb P xs ys else list_emb P (x#xs)
ys)
  by simp_all
```

1.9 Subsequences (special case of homeomorphic embedding)

```
abbreviation subseq :: 'a list ⇒ 'a list ⇒ bool
  where subseq xs ys ≝ list_emb (=) xs ys

definition strict_subseq where strict_subseq xs ys ⟷ xs ≠ ys ∧ subseq xs ys

lemma subseq_Cons2: subseq xs ys ⟹ subseq (x#xs) (x#ys) by auto

lemma subseq_same_length:
  assumes subseq xs ys and length xs = length ys shows xs = ys
  using assms by (induct) (auto dest: list_emb_length)

lemma not_subseq_length [simp]: length ys < length xs ⟹ ¬ subseq xs ys
  by (metis list_emb_length linorder_not_less)

lemma subseq_Cons': subseq (x#xs) ys ⟹ subseq xs ys
  by (induct xs, simp, blast dest: list_emb_ConsD)

lemma subseq_Cons2':
  assumes subseq (x#xs) (x#ys) shows subseq xs ys
  using assms by (cases) (rule subseq_Cons')

lemma subseq_Cons2_neq:
  assumes subseq (x#xs) (y#ys)
  shows x ≠ y ⟹ subseq (x#xs) ys
  using assms by (cases) auto

lemma subseq_Cons2_iff [simp]:
  subseq (x#xs) (y#ys) = (if x = y then subseq xs ys else subseq (x#xs) ys)
  by simp
```

```
lemma subseq_append′: subseq (zs @ xs) (zs @ ys) ⟷ subseq xs ys
 by (induct zs) simp_all

interpretation subseq_order: order subseq strict_subseq
proof
 fix xs ys :: ′a list
 {
  assume subseq xs ys and subseq ys xs
  thus xs = ys
  proof (induct)
   case list_emb_Nil
   from list_emb_Nil2 [OF this] show ?case by simp
  next
   case list_emb_Cons2
   thus ?case by simp
  next
   case list_emb_Cons
   hence False using subseq_Cons′ by fastforce
   thus ?case ..
  qed
 }
 thus strict_subseq xs ys ⟷ (subseq xs ys ∧ ¬subseq ys xs)
  by (auto simp: strict_subseq_def)
qed (auto simp: list_emb_refl intro: list_emb_trans)

lemma in_set_subseqs [simp]: xs ∈ set (subseqs ys) ⟷ subseq xs ys
proof
 assume xs ∈ set (subseqs ys)
 thus subseq xs ys
  by (induction ys arbitrary: xs) (auto simp: Let_def)
next
 have [simp]: [] ∈ set (subseqs ys) for ys :: ′a list
  by (induction ys) (auto simp: Let_def)
 assume subseq xs ys
 thus xs ∈ set (subseqs ys)
  by (induction xs ys rule: list_emb.induct) (auto simp: Let_def)
qed

lemma set_subseqs_eq: set (subseqs ys) = {xs. subseq xs ys}
 by auto

lemma subseq_append_le_same_iff: subseq (xs @ ys) ys ⟷ xs = []
 by (auto dest: list_emb_length)
```

```
lemma subseq_singleton_left: subseq [x] ys ⟷ x ∈ set ys
  by (fastforce dest: list_emb_ConsD split_list_last)

lemma list_emb_append_mono:
  ⟦ list_emb P xs xs'; list_emb P ys ys' ⟧ ⟹ list_emb P (xs@ys) (xs'@ys')
  by (induct rule: list_emb.induct) auto

lemma prefix_imp_subseq [intro]: prefix xs ys ⟹ subseq xs ys
  by (auto simp: prefix_def)

lemma suffix_imp_subseq [intro]: suffix xs ys ⟹ subseq xs ys
  by (auto simp: suffix_def)
```

1.10 Appending elements

```
lemma subseq_append [simp]:
  subseq (xs @ zs) (ys @ zs) ⟷ subseq xs ys (is ?l = ?r)
proof
  { fix xs' ys' xs ys zs :: 'a list assume subseq xs' ys'
    then have xs' = xs @ zs ∧ ys' = ys @ zs ⟶ subseq xs ys
    proof (induct arbitrary: xs ys zs)
      case list_emb_Nil show ?case by simp
    next
      case (list_emb_Cons xs' ys' x)
      { assume ys=[] then have ?case using list_emb_Cons(1) by auto }
      moreover
      { fix us assume ys = x#us
        then have ?case using list_emb_Cons(2) by(simp add: list_emb.list_emb_Cons)
}
      ultimately show ?case by (auto simp:Cons_eq_append_conv)
    next
      case (list_emb_Cons2 x y xs' ys')
      { assume xs=[] then have ?case using list_emb_Cons2(1) by auto }
      moreover
      { fix us vs assume xs=x#us ys=x#vs then have ?case using list_emb_Cons2 by
auto}
      moreover
        { fix us assume xs=x#us ys=[] then have ?case using list_emb_Cons2(2) by
bestsimp }
      ultimately show ?case using ‹(=) x y› by (auto simp: Cons_eq_append_conv)
    qed }
  moreover assume ?l
  ultimately show ?r by blast
next
  assume ?r then show ?l by (metis list_emb_append_mono subseq_order.order_refl)
qed
```

```
lemma subseq_append_iff:
  subseq xs (ys @ zs) ⟷ (∃xs1 xs2. xs = xs1 @ xs2 ∧ subseq xs1 ys ∧ subseq xs2 zs)
  (is ?lhs = ?rhs)
proof
  assume ?lhs thus ?rhs
  proof (induction xs ys @ zs arbitrary: ys zs rule: list_emb.induct)
    case (list_emb_Cons xs ws y ys zs)
   from list_emb_Cons(2)[of tl ys zs] and list_emb_Cons(2)[of [] tl zs] and list_emb_Cons(1,3)
      show ?case by (cases ys) auto
  next
    case (list_emb_Cons2 x y xs ws ys zs)
    from list_emb_Cons2(3)[of tl ys zs] and list_emb_Cons2(3)[of [] tl zs]
      and list_emb_Cons2(1,2,4)
    show ?case by (cases ys) (auto simp: Cons_eq_append_conv)
  qed auto
qed (auto intro: list_emb_append_mono)

lemma subseq_appendE [case_names append]:
  assumes subseq xs (ys @ zs)
  obtains xs1 xs2 where xs = xs1 @ xs2 subseq xs1 ys subseq xs2 zs
  using assms by (subst (asm) subseq_append_iff) auto

lemma subseq_drop_many: subseq xs ys ⟹ subseq xs (zs @ ys)
  by (induct zs) auto

lemma subseq_rev_drop_many: subseq xs ys ⟹ subseq xs (ys @ zs)
  by (metis append_Nil2 list_emb_Nil list_emb_append_mono)
```

1.11 Relation to standard list operations

```
lemma subseq_map:
  assumes subseq xs ys shows subseq (map f xs) (map f ys)
  using assms by (induct) auto

lemma subseq_filter_left [simp]: subseq (filter P xs) xs
  by (induct xs) auto

lemma subseq_filter [simp]:
  assumes subseq xs ys shows subseq (filter P xs) (filter P ys)
  using assms by induct auto

lemma subseq_conv_nths:
  subseq xs ys ⟷ (∃N. xs = nths ys N) (is ?L = ?R)
proof
  assume ?L
```

```
 then show ?R
 proof (induct)
  case list_emb_Nil show ?case by (metis nths_empty)
 next
  case (list_emb_Cons xs ys x)
  then obtain N where xs = nths ys N by blast
  then have xs = nths (x#ys) (Suc ‘ N)
   by (clarsimp simp add: nths_Cons inj_image_mem_iff)
  then show ?case by blast
 next
  case (list_emb_Cons2 x y xs ys)
  then obtain N where xs = nths ys N by blast
  then have x#xs = nths (x#ys) (insert 0 (Suc ‘ N))
   by (clarsimp simp add: nths_Cons inj_image_mem_iff)
  moreover from list_emb_Cons2 have x = y by simp
  ultimately show ?case by blast
 qed
next
 assume ?R
 then obtain N where xs = nths ys N ..
 moreover have subseq (nths ys N) ys
 proof (induct ys arbitrary: N)
  case Nil show ?case by simp
 next
  case Cons then show ?case by (auto simp: nths_Cons)
 qed
 ultimately show ?L by simp
qed
```

1.12 Contiguous sublists

```
definition sublist :: 'a list ⇒ 'a list ⇒ bool where
 sublist xs ys = (∃ ps ss. ys = ps @ xs @ ss)

definition strict_sublist :: 'a list ⇒ 'a list ⇒ bool where
 strict_sublist xs ys ⟷ sublist xs ys ∧ xs ≠ ys

interpretation sublist_order: order sublist strict_sublist
proof
 fix xs ys zs :: 'a list
 assume sublist xs ys sublist ys zs
 then obtain xs1 xs2 ys1 ys2 where ys = xs1 @ xs @ xs2 zs = ys1 @ ys @ ys2
  by (auto simp: sublist_def)
 hence zs = (ys1 @ xs1) @ xs @ (xs2 @ ys2) by simp
 thus sublist xs zs unfolding sublist_def by blast
next
```

```
 fix xs ys :: 'a list
 {
  assume sublist xs ys sublist ys xs
  then obtain as bs cs ds
   where xs: xs = as @ ys @ bs and ys: ys = cs @ xs @ ds
   by (auto simp: sublist_def)
  have xs = as @ cs @ xs @ ds @ bs by (subst xs, subst ys) auto
  also have length ... = length as + length cs + length xs + length bs + length ds
   by simp
  finally have as = [] bs = [] by simp_all
  with xs show xs = ys by simp
 }
 thus strict_sublist xs ys ⟷ (sublist xs ys ∧ ¬sublist ys xs)
  by (auto simp: strict_sublist_def)
qed (auto simp: strict_sublist_def sublist_def intro: exI[of _ []])

lemma sublist_Nil_left [simp, intro]: sublist [] ys
 by (auto simp: sublist_def)

lemma sublist_Cons_Nil [simp]: ¬sublist (x#xs) []
 by (auto simp: sublist_def)

lemma sublist_Nil_right [simp]: sublist xs [] ⟷ xs = []
 by (cases xs) auto

lemma sublist_appendI [simp, intro]: sublist xs (ps @ xs @ ss)
 by (auto simp: sublist_def)

lemma sublist_append_leftI [simp, intro]: sublist xs (ps @ xs)
 by (auto simp: sublist_def intro: exI[of _ []])

lemma sublist_append_rightI [simp, intro]: sublist xs (xs @ ss)
 by (auto simp: sublist_def intro: exI[of _ []])

lemma sublist_altdef: sublist xs ys ⟷ (∃ys′. prefix ys′ ys ∧ suffix xs ys′)
proof safe
 assume sublist xs ys
 then obtain ps ss where ys = ps @ xs @ ss by (auto simp: sublist_def)
 thus ∃ys′. prefix ys′ ys ∧ suffix xs ys′
  by (intro exI[of _ ps @ xs] conjI suffix_appendI) auto
next
 fix ys′
 assume prefix ys′ ys suffix xs ys′
 thus sublist xs ys by (auto simp: prefix_def suffix_def)
qed
```

```
lemma sublist_altdef': sublist xs ys ⟷ (∃ys'. suffix ys' ys ∧ prefix xs ys')
proof safe
  assume sublist xs ys
  then obtain ps ss where ys = ps @ xs @ ss by (auto simp: sublist_def)
  thus ∃ys'. suffix ys' ys ∧ prefix xs ys'
    by (intro exI[of _ xs @ ss] conjI suffixI) auto
next
  fix ys'
  assume suffix ys' ys prefix xs ys'
  thus sublist xs ys by (auto simp: prefix_def suffix_def)
qed

lemma sublist_Cons_right: sublist xs (y # ys) ⟷ prefix xs (y # ys) ∨ sublist xs ys
  by (auto simp: sublist_def prefix_def Cons_eq_append_conv)

lemma sublist_code [code]:
  sublist [] ys ⟷ True
  sublist (x # xs) [] ⟷ False
  sublist (x # xs) (y # ys) ⟷ prefix (x # xs) (y # ys) ∨ sublist (x # xs) ys
  by (simp_all add: sublist_Cons_right)


lemma sublist_append:
  sublist xs (ys @ zs) ⟷
    sublist xs ys ∨ sublist xs zs ∨ (∃xs1 xs2. xs = xs1 @ xs2 ∧ suffix xs1 ys ∧ prefix xs2
zs)
  by (auto simp: sublist_altdef prefix_append suffix_append)

primrec sublists :: 'a list ⇒ 'a list list where
  sublists [] = [[]]
| sublists (x # xs) = sublists xs @ map ((#) x) (prefixes xs)

lemma in_set_sublists [simp]: xs ∈ set (sublists ys) ⟷ sublist xs ys
  by (induction ys arbitrary: xs) (auto simp: sublist_Cons_right prefix_Cons)

lemma set_sublists_eq: set (sublists xs) = {ys. sublist ys xs}
  by auto

lemma length_sublists [simp]: length (sublists xs) = Suc (length xs ∗ Suc (length xs)
div 2)
  by (induction xs) simp_all

lemma sublist_length_le: sublist xs ys ⟹ length xs ≤ length ys
  by (auto simp add: sublist_def)
```

```
lemma set_mono_sublist: sublist xs ys ⟹ set xs ⊆ set ys
  by (auto simp add: sublist_def)

lemma prefix_imp_sublist [simp, intro]: prefix xs ys ⟹ sublist xs ys
  by (auto simp: sublist_def prefix_def intro: exI[of _ []])

lemma suffix_imp_sublist [simp, intro]: suffix xs ys ⟹ sublist xs ys
  by (auto simp: sublist_def suffix_def intro: exI[of _ []])

lemma sublist_take [simp, intro]: sublist (take n xs) xs
  by (rule prefix_imp_sublist) (simp_all add: take_is_prefix)

lemma sublist_drop [simp, intro]: sublist (drop n xs) xs
  by (rule suffix_imp_sublist) (simp_all add: suffix_drop)

lemma sublist_tl [simp, intro]: sublist (tl xs) xs
  by (rule suffix_imp_sublist) (simp_all add: suffix_drop)

lemma sublist_butlast [simp, intro]: sublist (butlast xs) xs
  by (rule prefix_imp_sublist) (simp_all add: prefixeq_butlast)

lemma sublist_rev [simp]: sublist (rev xs) (rev ys) = sublist xs ys
proof
  assume sublist (rev xs) (rev ys)
  then obtain as bs where rev ys = as @ rev xs @ bs
    by (auto simp: sublist_def)
  also have rev ... = rev bs @ xs @ rev as by simp
  finally show sublist xs ys by simp
next
  assume sublist xs ys
  then obtain as bs where ys = as @ xs @ bs
    by (auto simp: sublist_def)
  also have rev ... = rev bs @ rev xs @ rev as by simp
  finally show sublist (rev xs) (rev ys) by simp
qed

lemma sublist_rev_left: sublist (rev xs) ys = sublist xs (rev ys)
  by (subst sublist_rev [symmetric]) (simp only: rev_rev_ident)

lemma sublist_rev_right: sublist xs (rev ys) = sublist (rev xs) ys
  by (subst sublist_rev [symmetric]) (simp only: rev_rev_ident)

lemma snoc_sublist_snoc:
  sublist (xs @ [x]) (ys @ [y]) ⟷
```

(*x* = *y* ∧ *suffix xs ys* ∨ *sublist* (*xs* @ [*x*]) *ys*)
by (*subst* (*1 2*) *sublist_rev* [*symmetric*])
(*simp del*: *sublist_rev add*: *sublist_Cons_right suffix_to_prefix*)

lemma *sublist_snoc*:
sublist xs (*ys* @ [*y*]) ⟷ *suffix xs* (*ys* @ [*y*]) ∨ *sublist xs ys*
by (*subst* (*1 2*) *sublist_rev* [*symmetric*])
(*simp del*: *sublist_rev add*: *sublist_Cons_right suffix_to_prefix*)

lemma *sublist_imp_subseq* [*intro*]: *sublist xs ys* ⟹ *subseq xs ys*
by (*auto simp*: *sublist_def*)

1.13 Parametricity

context includes *lifting_syntax*
begin

private lemma *prefix_primrec*:
prefix = *rec_list* (λ*xs*. *True*) (λ*x xs xsa ys*.
case ys of [] ⇒ *False* | *y* # *ys* ⇒ *x* = *y* ∧ *xsa ys*)
proof (*intro ext*, *goal_cases*)
case (*1 xs ys*)
show *?case* **by** (*induction xs arbitrary*: *ys*) (*auto simp*: *prefix_Cons split*: *list.splits*)
qed

private lemma *sublist_primrec*:
sublist = (λ*xs ys*. *rec_list* (λ*xs*. *xs* = []) (λ*y ys ysa xs*. *prefix xs* (*y* # *ys*) ∨ *ysa xs*) *ys*
xs)
proof (*intro ext*, *goal_cases*)
case (*1 xs ys*)
show *?case* **by** (*induction ys*) (*auto simp*: *sublist_Cons_right*)
qed

private lemma *list_emb_primrec*:
list_emb = (λ*uu uua uuaa*. *rec_list* (λ*P xs*. *List.null xs*) (λ*y ys ysa P xs*. *case xs of* []
⇒ *True*
| *x* # *xs* ⇒ *if P x y then ysa P xs else ysa P* (*x* # *xs*)) *uuaa uu uua*)
proof (*intro ext*, *goal_cases*)
case (*1 P xs ys*)
show *?case*
by (*induction ys arbitrary*: *xs*)
(*auto simp*: *list_emb_code List.null_def split*: *list.splits*)
qed

lemma *prefix_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*

shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *prefix prefix*
unfolding *prefix_primrec* **by** *transfer_prover*

lemma *suffix_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*
shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *suffix suffix*
unfolding *suffix_to_prefix* [*abs_def*] **by** *transfer_prover*

lemma *sublist_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*
shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *sublist sublist*
unfolding *sublist_primrec* **by** *transfer_prover*

lemma *parallel_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*
shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *parallel parallel*
unfolding *parallel_def* **by** *transfer_prover*

lemma *list_emb_transfer* [*transfer_rule*]:
((*A* ===> *A* ===> (=)) ===> *list_all2 A* ===> *list_all2 A* ===> (=)) *list_emb list_emb*
unfolding *list_emb_primrec* **by** *transfer_prover*

lemma *strict_prefix_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*
shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *strict_prefix strict_prefix*
unfolding *strict_prefix_def* **by** *transfer_prover*

lemma *strict_suffix_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*
shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *strict_suffix strict_suffix*
unfolding *strict_suffix_def* **by** *transfer_prover*

lemma *strict_subseq_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*
shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *strict_subseq strict_subseq*
unfolding *strict_subseq_def* **by** *transfer_prover*

lemma *strict_sublist_transfer* [*transfer_rule*]:
assumes [*transfer_rule*]: *bi_unique A*
shows (*list_all2 A* ===> *list_all2 A* ===> (=)) *strict_sublist strict_sublist*
unfolding *strict_sublist_def* **by** *transfer_prover*

lemma *prefixes_transfer* [*transfer_rule*]:
  assumes [*transfer_rule*]: *bi_unique A*
  shows (*list_all2 A* ===> *list_all2* (*list_all2 A*)) *prefixes prefixes*
  unfolding *prefixes_def* **by** *transfer_prover*

lemma *suffixes_transfer* [*transfer_rule*]:
  assumes [*transfer_rule*]: *bi_unique A*
  shows (*list_all2 A* ===> *list_all2* (*list_all2 A*)) *suffixes suffixes*
  unfolding *suffixes_def* **by** *transfer_prover*

lemma *sublists_transfer* [*transfer_rule*]:
  assumes [*transfer_rule*]: *bi_unique A*
  shows (*list_all2 A* ===> *list_all2* (*list_all2 A*)) *sublists sublists*
  unfolding *sublists_def* **by** *transfer_prover*

end

end

theory *Spec*
  imports *Main* ~~/*src*/*HOL*/*Library*/*Sublist*
begin

2 Sequential Composition of Languages

definition
  Sequ :: *string set* ⇒ *string set* ⇒ *string set* (_ ;; _ [*100*,*100*] *100*)
where
  A ;; *B* = {*s1* @ *s2* | *s1 s2*. *s1* ∈ *A* ∧ *s2* ∈ *B*}

Two Simple Properties about Sequential Composition

lemma *Sequ_empty_string* [*simp*]:
  shows *A* ;; {[]} = *A*
  and {[]} ;; *A* = *A*
by (*simp_all add*: *Sequ_def*)

lemma *Sequ_empty* [*simp*]:
  shows *A* ;; {} = {}
  and {} ;; *A* = {}
by (*simp_all add*: *Sequ_def*)

3 Semantic Derivative (Left Quotient) of Languages

definition
  Der :: *char* ⇒ *string set* ⇒ *string set*

where
$Der\ c\ A \overset{def}{=} \{s.\ c \# s \in A\}$

definition
$Ders :: string \Rightarrow string\ set \Rightarrow string\ set$
where
$Ders\ s\ A \overset{def}{=} \{s'.\ s\ @\ s' \in A\}$

lemma *Der_null* [*simp*]:
shows $Der\ c\ \{\} = \{\}$
unfolding *Der_def*
by *auto*

lemma *Der_empty* [*simp*]:
shows $Der\ c\ \{[]\} = \{\}$
unfolding *Der_def*
by *auto*

lemma *Der_char* [*simp*]:
shows $Der\ c\ \{[d]\} = (if\ c = d\ then\ \{[]\}\ else\ \{\})$
unfolding *Der_def*
by *auto*

lemma *Der_union* [*simp*]:
shows $Der\ c\ (A \cup B) = Der\ c\ A \cup Der\ c\ B$
unfolding *Der_def*
by *auto*

lemma *Der_Sequ* [*simp*]:
shows $Der\ c\ (A\ ;;\ B) = (Der\ c\ A)\ ;;\ B \cup (if\ [] \in A\ then\ Der\ c\ B\ else\ \{\})$
unfolding *Der_def Sequ_def*
by (*auto simp add*: *Cons_eq_append_conv*)

4 Kleene Star for Languages

inductive-set
$Star :: string\ set \Rightarrow string\ set\ (\_\star\ [101]\ 102)$
for $A :: string\ set$
where
start[*intro*]: $[] \in A\star$
| *step*[*intro*]: $[\![s1 \in A;\ s2 \in A\star]\!] \Longrightarrow s1\ @\ s2 \in A\star$

lemma *Star_cases*:

```
  shows A⋆ = {[]} ∪ A ;; A⋆
unfolding Sequ_def
by (auto) (metis Star.simps)

lemma Star_decomp:
  assumes c # x ∈ A⋆
  shows ∃ s1 s2. x = s1 @ s2 ∧ c # s1 ∈ A ∧ s2 ∈ A⋆
using assms
by (induct x ≝ c # x rule: Star.induct)
  (auto simp add: append_eq_Cons_conv)

lemma Star_Der_Sequ:
  shows Der c (A⋆) ⊆ (Der c A) ;; A⋆
unfolding Der_def Sequ_def
by(auto simp add: Star_decomp)


lemma Der_star [simp]:
  shows Der c (A⋆) = (Der c A) ;; A⋆
proof −
  have Der c (A⋆) = Der c ({[]} ∪ A ;; A⋆)
    by (simp only: Star_cases[symmetric])
  also have ... = Der c (A ;; A⋆)
    by (simp only: Der_union Der_empty) (simp)
  also have ... = (Der c A) ;; A⋆ ∪ (if [] ∈ A then Der c (A⋆) else {})
    by simp
  also have ... =  (Der c A) ;; A⋆
    using Star_Der_Sequ by auto
  finally show Der c (A⋆) = (Der c A) ;; A⋆ .
qed
```

5 Regular Expressions

```
datatype rexp =
  ZERO
| ONE
| CHAR char
| SEQ rexp rexp
| ALT rexp rexp
| STAR rexp
```

6 Semantics of Regular Expressions

```
fun
```

```
 L :: rexp ⇒ string set
where
 L (ZERO) = {}
| L (ONE) = {[]}
| L (CHAR c) = {[c]}
| L (SEQ r1 r2) = (L r1) ;; (L r2)
| L (ALT r1 r2) = (L r1) ∪ (L r2)
| L (STAR r) = (L r)⋆
```

7 Nullable, Derivatives

```
fun
 nullable :: rexp ⇒ bool
where
 nullable (ZERO) = False
| nullable (ONE) = True
| nullable (CHAR c) = False
| nullable (ALT r1 r2) = (nullable r1 ∨ nullable r2)
| nullable (SEQ r1 r2) = (nullable r1 ∧ nullable r2)
| nullable (STAR r) = True
```

```
fun
 der :: char ⇒ rexp ⇒ rexp
where
 der c (ZERO) = ZERO
| der c (ONE) = ZERO
| der c (CHAR d) = (if c = d then ONE else ZERO)
| der c (ALT r1 r2) = ALT (der c r1) (der c r2)
| der c (SEQ r1 r2) =
    (if nullable r1
     then ALT (SEQ (der c r1) r2) (der c r2)
     else SEQ (der c r1) r2)
| der c (STAR r) = SEQ (der c r) (STAR r)
```

```
fun
 ders :: string ⇒ rexp ⇒ rexp
where
 ders [] r = r
| ders (c # s) r = ders s (der c r)
```

```
lemma nullable_correctness:
 shows nullable r ⟷ [] ∈ (L r)
by (induct r) (auto simp add: Sequ_def)
```

```
lemma der_correctness:
  shows L (der c r) = Der c (L r)
by (induct r) (simp_all add: nullable_correctness)

lemma ders_correctness:
  shows L (ders s r) = Ders s (L r)
by (induct s arbitrary: r)
   (simp_all add: Ders_def der_correctness Der_def)

lemma ders_append:
  shows ders (s1 @ s2) r = ders s2 (ders s1 r)
  apply(induct s1 arbitrary: s2 r)
  apply(auto)
  done
```

8 Values

```
datatype val =
  Void
| Char char
| Seq val val
| Right val
| Left val
| Stars val list
```

9 The string behind a value

```
fun
  flat :: val ⇒ string
where
  flat (Void) = []
| flat (Char c) = [c]
| flat (Left v) = flat v
| flat (Right v) = flat v
| flat (Seq v1 v2) = (flat v1) @ (flat v2)
| flat (Stars []) = []
| flat (Stars (v#vs)) = (flat v) @ (flat (Stars vs))

abbreviation
  flats vs ≝ concat (map flat vs)

lemma flat_Stars [simp]:
 flat (Stars vs) = flats vs
by (induct vs) (auto)
```

```
lemma Star_concat:
  assumes ∀ s ∈ set ss. s ∈ A
  shows concat ss ∈ A⋆
using assms by (induct ss) (auto)

lemma Star_cstring:
  assumes s ∈ A⋆
  shows ∃ ss. concat ss = s ∧ (∀ s ∈ set ss. s ∈ A ∧ s ≠ [])
using assms
apply(induct rule: Star.induct)
apply(auto)[1]
apply(rule_tac x=[] in exI)
apply(simp)
apply(erule exE)
apply(clarify)
apply(case_tac s1 = [])
apply(rule_tac x=ss in exI)
apply(simp)
apply(rule_tac x=s1#ss in exI)
apply(simp)
done
```

10 Lexical Values

```
inductive
  Prf :: val ⇒ rexp ⇒ bool (⊨ _ : _ [100, 100] 100)
where
 ⟦⊨ v1 : r1; ⊨ v2 : r2⟧ ⟹ ⊨ Seq v1 v2 : SEQ r1 r2
| ⊨ v1 : r1 ⟹ ⊨ Left v1 : ALT r1 r2
| ⊨ v2 : r2 ⟹ ⊨ Right v2 : ALT r1 r2
| ⊨ Void : ONE
| ⊨ Char c : CHAR c
| ∀ v ∈ set vs. ⊨ v : r ∧ flat v ≠ [] ⟹ ⊨ Stars vs : STAR r

inductive-cases Prf_elims:
  ⊨ v : ZERO
  ⊨ v : SEQ r1 r2
  ⊨ v : ALT r1 r2
  ⊨ v : ONE
  ⊨ v : CHAR c
  ⊨ vs : STAR r

lemma Prf_Stars_appendE:
  assumes ⊨ Stars (vs1 @ vs2) : STAR r
```

```
  shows ⊨ Stars vs1 : STAR r ∧ ⊨ Stars vs2 : STAR r
using assms
by (auto intro: Prf.intros elim!: Prf_elims)
```

```
lemma Star_cval:
  assumes ∀ s ∈ set ss. ∃ v. s = flat v ∧ ⊨ v : r
  shows ∃ vs. flats vs = concat ss ∧ (∀ v ∈ set vs. ⊨ v : r ∧ flat v ≠ [])
using assms
apply(induct ss)
apply(auto)
apply(rule_tac x=[] in exI)
apply(simp)
apply(case_tac flat v = [])
apply(rule_tac x=vs in exI)
apply(simp)
apply(rule_tac x=v#vs in exI)
apply(simp)
done
```

```
lemma L_flat_Prf1:
  assumes ⊨ v : r
  shows flat v ∈ L r
using assms
by (induct) (auto simp add: Sequ_def Star_concat)
```

```
lemma L_flat_Prf2:
  assumes s ∈ L r
  shows ∃ v. ⊨ v : r ∧ flat v = s
using assms
proof(induct r arbitrary: s)
  case (STAR r s)
  have IH: ⋀s. s ∈ L r ⟹ ∃ v. ⊨ v : r ∧ flat v = s by fact
  have s ∈ L (STAR r) by fact
  then obtain ss where concat ss = s ∀ s ∈ set ss. s ∈ L r ∧ s ≠ []
  using Star_cstring by auto
  then obtain vs where flats vs = s ∀ v ∈ set vs. ⊨ v : r ∧ flat v ≠ []
  using IH Star_cval by metis
  then show ∃ v. ⊨ v : STAR r ∧ flat v = s
  using Prf.intros(6) flat_Stars by blast
next
  case (SEQ r1 r2 s)
  then show ∃ v. ⊨ v : SEQ r1 r2 ∧ flat v = s
  unfolding Sequ_def L.simps by (fastforce intro: Prf.intros)
```

next
  case $(ALT\ r1\ r2\ s)$
  then show $\exists\, v.\ \models v : ALT\ r1\ r2 \wedge flat\ v = s$
  unfolding *L.simps* **by** (*fastforce intro*: *Prf .intros*)
qed (*auto intro*: *Prf .intros*)

lemma *L_flat_Prf*:
  shows $L(r) = \{flat\ v \mid v.\ \models v : r\}$
using *L_flat_Prf1 L_flat_Prf2* **by** *blast*

11 Sets of Lexical Values

Shows that lexical values are finite for a given regex and string.

definition
  $LV :: rexp \Rightarrow string \Rightarrow val\ set$
where $LV\ r\ s \overset{def}{=} \{v.\ \models v : r \wedge flat\ v = s\}$

lemma *LV_simps*:
  shows $LV\ ZERO\ s = \{\}$
  and $LV\ ONE\ s = (if\ s = []\ then\ \{Void\}\ else\ \{\})$
  and $LV\ (CHAR\ c)\ s = (if\ s = [c]\ then\ \{Char\ c\}\ else\ \{\})$
  and $LV\ (ALT\ r1\ r2)\ s = Left\ `\ LV\ r1\ s \cup Right\ `\ LV\ r2\ s$
unfolding *LV_def*
by (*auto intro*: *Prf .intros elim*: *Prf .cases*)

abbreviation
  $Prefixes\ s \overset{def}{=} \{s'.\ prefix\ s'\ s\}$

abbreviation
  $Suffixes\ s \overset{def}{=} \{s'.\ suffix\ s'\ s\}$

abbreviation
  $SSuffixes\ s \overset{def}{=} \{s'.\ strict\_suffix\ s'\ s\}$

lemma *Suffixes_cons* [*simp*]:
  shows $Suffixes\ (c\ \#\ s) = Suffixes\ s \cup \{c\ \#\ s\}$
by (*auto simp add*: *suffix_def Cons_eq_append_conv*)

lemma *finite_Suffixes*:
  shows *finite* (*Suffixes s*)
by (*induct s*) (*simp_all*)

```
lemma finite_SSuffixes:
 shows finite (SSuffixes s)
proof -
 have SSuffixes s ⊆ Suffixes s
  unfolding strict_suffix_def suffix_def by auto
 then show finite (SSuffixes s)
  using finite_Suffixes finite_subset by blast
qed

lemma finite_Prefixes:
 shows finite (Prefixes s)
proof -
 have finite (Suffixes (rev s))
   by (rule finite_Suffixes)
 then have finite (rev ‘ Suffixes (rev s)) by simp
 moreover
 have rev ‘ (Suffixes (rev s)) = Prefixes s
 unfolding suffix_def prefix_def image_def
  by (auto)(metis rev_append rev_rev_ident)+
 ultimately show finite (Prefixes s) by simp
qed

lemma LV_STAR_finite:
 assumes ∀ s. finite (LV r s)
 shows finite (LV (STAR r) s)
proof(induct s rule: length_induct)
 fix s::char list
 assume ∀ s'. length s' < length s ⟶ finite (LV (STAR r) s')
 then have IH: ∀ s' ∈ SSuffixes s. finite (LV (STAR r) s')
   by (force simp add: strict_suffix_def suffix_def)
 define f where f ≝ λ(v, vs). Stars (v # vs)
 define S1 where S1 ≝ ⋃s' ∈ Prefixes s. LV r s'
 define S2 where S2 ≝ ⋃s2 ∈ SSuffixes s. Stars -‘ (LV (STAR r) s2)
 have finite S1 using assms
   unfolding S1_def by (simp_all add: finite_Prefixes)
 moreover
 with IH have finite S2 unfolding S2_def
   by (auto simp add: finite_SSuffixes inj_on_def finite_vimageI)
 ultimately
 have finite ({Stars []} ∪ f ‘ (S1 × S2)) by simp
 moreover
 have LV (STAR r) s ⊆ {Stars []} ∪ f ‘ (S1 × S2)
 unfolding S1_def S2_def f_def
 unfolding LV_def image_def prefix_def strict_suffix_def
```

```
  apply(auto)
  apply(case_tac x)
  apply(auto elim: Prf_elims)
  apply(erule Prf_elims)
  apply(auto)
  apply(case_tac vs)
  apply(auto intro: Prf.intros)
  apply(rule exI)
  apply(rule conjI)
  apply(rule_tac x=flat a in exI)
  apply(rule conjI)
  apply(rule_tac x=flats list in exI)
  apply(simp)
   apply(blast)
  apply(simp add: suffix_def)
  using Prf.intros(6) by blast
  ultimately
  show finite (LV (STAR r) s) by (simp add: finite_subset)
qed


lemma LV_finite:
  shows finite (LV r s)
proof(induct r arbitrary: s)
  case (ZERO s)
  show finite (LV ZERO s) by (simp add: LV_simps)
next
  case (ONE s)
  show finite (LV ONE s) by (simp add: LV_simps)
next
  case (CHAR c s)
  show finite (LV (CHAR c) s) by (simp add: LV_simps)
next
  case (ALT r1 r2 s)
  then show finite (LV (ALT r1 r2) s) by (simp add: LV_simps)
next
  case (SEQ r1 r2 s)
  define f where f ≝ λ(v1, v2). Seq v1 v2
  define S1 where S1 ≝ ⋃s' ∈ Prefixes s. LV r1 s'
  define S2 where S2 ≝ ⋃s' ∈ Suffixes s. LV r2 s'
  have IHs: ⋀s. finite (LV r1 s) ⋀s. finite (LV r2 s) by fact+
  then have finite S1 finite S2 unfolding S1_def S2_def
    by (simp_all add: finite_Prefixes finite_Suffixes)
  moreover
  have LV (SEQ r1 r2) s ⊆ f ‘ (S1 × S2)
```

```
   unfolding f_def S1_def S2_def
   unfolding LV_def image_def prefix_def suffix_def
   apply (auto elim!: Prf_elims)
   by (metis (mono_tags, lifting) mem_Collect_eq)
  ultimately
  show finite (LV (SEQ r1 r2) s)
   by (simp add: finite_subset)
next
  case (STAR r s)
  then show finite (LV (STAR r) s) by (simp add: LV_STAR_finite)
qed
```

12 Our POSIX Definition

```
inductive
  Posix :: string ⇒ rexp ⇒ val ⇒ bool (_ ∈ _ → _ [100, 100, 100] 100)
where
  Posix_ONE: [] ∈ ONE → Void
| Posix_CHAR: [c] ∈ (CHAR c) → (Char c)
| Posix_ALT1: s ∈ r1 → v ⟹ s ∈ (ALT r1 r2) → (Left v)
| Posix_ALT2: ⟦s ∈ r2 → v; s ∉ L(r1)⟧ ⟹ s ∈ (ALT r1 r2) → (Right v)
| Posix_SEQ: ⟦s1 ∈ r1 → v1; s2 ∈ r2 → v2;
    ¬(∃s3 s4. s3 ≠ [] ∧ s3 @ s4 = s2 ∧ (s1 @ s3) ∈ L r1 ∧ s4 ∈ L r2)⟧ ⟹
    (s1 @ s2) ∈ (SEQ r1 r2) → (Seq v1 v2)
| Posix_STAR1: ⟦s1 ∈ r → v; s2 ∈ STAR r → Stars vs; flat v ≠ [];
    ¬(∃s3 s4. s3 ≠ [] ∧ s3 @ s4 = s2 ∧ (s1 @ s3) ∈ L r ∧ s4 ∈ L (STAR r))⟧
    ⟹ (s1 @ s2) ∈ STAR r → Stars (v # vs)
| Posix_STAR2: [] ∈ STAR r → Stars []

inductive-cases Posix_elims:
  s ∈ ZERO → v
  s ∈ ONE → v
  s ∈ CHAR c → v
  s ∈ ALT r1 r2 → v
  s ∈ SEQ r1 r2 → v
  s ∈ STAR r → v

lemma Posix1:
  assumes s ∈ r → v
  shows s ∈ L r flat v = s
using assms
by (induct s r v rule: Posix.induct)
  (auto simp add: Sequ_def)
```

Our Posix definition determines a unique value.

```
lemma Posix_determ:
```

```
  assumes s ∈ r → v1 s ∈ r → v2
  shows v1 = v2
using assms
proof (induct s r v1 arbitrary: v2 rule: Posix.induct)
  case (Posix_ONE v2)
  have [] ∈ ONE → v2 by fact
  then show Void = v2 by cases auto
next
  case (Posix_CHAR c v2)
  have [c] ∈ CHAR c → v2 by fact
  then show Char c = v2 by cases auto
next
  case (Posix_ALT1 s r1 v r2 v2)
  have s ∈ ALT r1 r2 → v2 by fact
  moreover
  have s ∈ r1 → v by fact
  then have s ∈ L r1 by (simp add: Posix1)
  ultimately obtain v′ where eq: v2 = Left v′ s ∈ r1 → v′ by cases auto
  moreover
  have IH: ⋀v2. s ∈ r1 → v2 ⟹ v = v2 by fact
  ultimately have v = v′ by simp
  then show Left v = v2 using eq by simp
next
  case (Posix_ALT2 s r2 v r1 v2)
  have s ∈ ALT r1 r2 → v2 by fact
  moreover
  have s ∉ L r1 by fact
  ultimately obtain v′ where eq: v2 = Right v′ s ∈ r2 → v′
    by cases (auto simp add: Posix1)
  moreover
  have IH: ⋀v2. s ∈ r2 → v2 ⟹ v = v2 by fact
  ultimately have v = v′ by simp
  then show Right v = v2 using eq by simp
next
  case (Posix_SEQ s1 r1 v1 s2 r2 v2 v′)
  have (s1 @ s2) ∈ SEQ r1 r2 → v′
       s1 ∈ r1 → v1 s2 ∈ r2 → v2
       ¬ (∃s3 s4. s3 ≠ [] ∧ s3 @ s4 = s2 ∧ s1 @ s3 ∈ L r1 ∧ s4 ∈ L r2) by fact+
  then obtain v1′ v2′ where v′ = Seq v1′ v2′ s1 ∈ r1 → v1′ s2 ∈ r2 → v2′
  apply(cases) apply (auto simp add: append_eq_append_conv2)
  using Posix1(1) by fastforce+
  moreover
  have IHs: ⋀v1′. s1 ∈ r1 → v1′ ⟹ v1 = v1′
            ⋀v2′. s2 ∈ r2 → v2′ ⟹ v2 = v2′ by fact+
  ultimately show Seq v1 v2 = v′ by simp
```

```
next
  case (Posix_STAR1 s1 r v s2 vs v2)
  have (s1 @ s2) ∈ STAR r → v2
       s1 ∈ r → v s2 ∈ STAR r → Stars vs flat v ≠ []
       ¬ (∃s₃ s₄. s₃ ≠ [] ∧ s₃ @ s₄ = s2 ∧ s1 @ s₃ ∈ L r ∧ s₄ ∈ L (STAR r)) by fact+
  then obtain v′ vs′ where v2 = Stars (v′ # vs′) s1 ∈ r → v′ s2 ∈ (STAR r) → (Stars vs′)
  apply(cases) apply (auto simp add: append_eq_append_conv2)
  using Posix1(1) apply fastforce
  apply (metis Posix1(1) Posix_STAR1.hyps(6) append_Nil append_Nil2)
  using Posix1(2) by blast
  moreover
  have IHs: ⋀v2. s1 ∈ r → v2 ⟹ v = v2
          ⋀v2. s2 ∈ STAR r → v2 ⟹ Stars vs = v2 by fact+
  ultimately show Stars (v # vs) = v2 by auto
next
  case (Posix_STAR2 r v2)
  have [] ∈ STAR r → v2 by fact
  then show Stars [] = v2 by cases (auto simp add: Posix1)
qed
```

Our POSIX values are lexical values.

```
lemma Posix_LV:
  assumes s ∈ r → v
  shows v ∈ LV r s
  using assms unfolding LV_def
  apply(induct rule: Posix.induct)
  apply(auto simp add: intro!: Prf.intros elim!: Prf_elims)
  done

lemma Posix_Prf:
  assumes s ∈ r → v
  shows ⊨ v : r
  using assms Posix_LV LV_def
  by simp

end

theory Lexer
  imports Spec
begin
```

13 The Lexer Functions by Sulzmann and Lu

```
fun
```

```
  mkeps :: rexp ⇒ val
where
  mkeps(ONE) = Void
| mkeps(SEQ r1 r2) = Seq (mkeps r1) (mkeps r2)
| mkeps(ALT r1 r2) = (if nullable(r1) then Left (mkeps r1) else Right (mkeps r2))
| mkeps(STAR r) = Stars []

fun injval :: rexp ⇒ char ⇒ val ⇒ val
where
  injval (CHAR d) c Void = Char d
| injval (ALT r1 r2) c (Left v1) = Left(injval r1 c v1)
| injval (ALT r1 r2) c (Right v2) = Right(injval r2 c v2)
| injval (SEQ r1 r2) c (Seq v1 v2) = Seq (injval r1 c v1) v2
| injval (SEQ r1 r2) c (Left (Seq v1 v2)) = Seq (injval r1 c v1) v2
| injval (SEQ r1 r2) c (Right v2) = Seq (mkeps r1) (injval r2 c v2)
| injval (STAR r) c (Seq v (Stars vs)) = Stars ((injval r c v) # vs)

fun
  lexer :: rexp ⇒ string ⇒ val option
where
  lexer r [] = (if nullable r then Some(mkeps r) else None)
| lexer r (c#s) = (case (lexer (der c r) s) of
                   None ⇒ None
                 | Some(v) ⇒ Some(injval r c v))
```

14 Mkeps, Injval Properties

```
lemma mkeps_nullable:
  assumes nullable(r)
  shows ⊨ mkeps r : r
using assms
by (induct rule: nullable.induct)
   (auto intro: Prf.intros)

lemma mkeps_flat:
  assumes nullable(r)
  shows flat (mkeps r) = []
using assms
by (induct rule: nullable.induct) (auto)

lemma Prf_injval_flat:
  assumes ⊨ v : der c r
  shows flat (injval r c v) = c # (flat v)
using assms
apply(induct c r arbitrary: v rule: der.induct)
```

```
apply(auto elim!: Prf_elims intro: mkeps_flat split: if_splits)
done

lemma Prf_injval:
  assumes ⊨ v : der c r
  shows ⊨ (injval r c v) : r
using assms
apply(induct r arbitrary: c v rule: rexp.induct)
apply(auto intro!: Prf.intros mkeps_nullable elim!: Prf_elims split: if_splits)
apply(simp add: Prf_injval_flat)
done
```

Mkeps and injval produce, or preserve, Posix values.

```
lemma Posix_mkeps:
  assumes nullable r
  shows [] ∈ r → mkeps r
using assms
apply(induct r rule: nullable.induct)
apply(auto intro: Posix.intros simp add: nullable_correctness Sequ_def)
apply(subst append.simps(1)[symmetric])
apply(rule Posix.intros)
apply(auto)
done

lemma Posix_injval:
  assumes s ∈ (der c r) → v
  shows (c # s) ∈ r → (injval r c v)
using assms
proof(induct r arbitrary: s v rule: rexp.induct)
  case ZERO
  have s ∈ der c ZERO → v by fact
  then have s ∈ ZERO → v by simp
  then have False by cases
  then show (c # s) ∈ ZERO → (injval ZERO c v) by simp
next
  case ONE
  have s ∈ der c ONE → v by fact
  then have s ∈ ZERO → v by simp
  then have False by cases
  then show (c # s) ∈ ONE → (injval ONE c v) by simp
next
  case (CHAR d)
  consider (eq) c = d | (ineq) c ≠ d by blast
  then show (c # s) ∈ (CHAR d) → (injval (CHAR d) c v)
  proof (cases)
    case eq
```

```
    have s ∈ der c (CHAR d) → v by fact
    then have s ∈ ONE → v using eq by simp
    then have eqs: s = [] ∧ v = Void by cases simp
    show (c # s) ∈ CHAR d → injval (CHAR d) c v using eq eqs
    by (auto intro: Posix.intros)
  next
    case ineq
    have s ∈ der c (CHAR d) → v by fact
    then have s ∈ ZERO → v using ineq by simp
    then have False by cases
    then show (c # s) ∈ CHAR d → injval (CHAR d) c v by simp
  qed
next
  case (ALT r1 r2)
  have IH1: ⋀s v. s ∈ der c r1 → v ⟹ (c # s) ∈ r1 → injval r1 c v by fact
  have IH2: ⋀s v. s ∈ der c r2 → v ⟹ (c # s) ∈ r2 → injval r2 c v by fact
  have s ∈ der c (ALT r1 r2) → v by fact
  then have s ∈ ALT (der c r1) (der c r2) → v by simp
  then consider (left) v′ where v = Left v′ s ∈ der c r1 → v′
          | (right) v′ where v = Right v′ s ∉ L (der c r1) s ∈ der c r2 → v′
          by cases auto
  then show (c # s) ∈ ALT r1 r2 → injval (ALT r1 r2) c v
  proof (cases)
    case left
    have s ∈ der c r1 → v′ by fact
    then have (c # s) ∈ r1 → injval r1 c v′ using IH1 by simp
     then have (c # s) ∈ ALT r1 r2 → injval (ALT r1 r2) c (Left v′) by (auto intro:
Posix.intros)
    then show (c # s) ∈ ALT r1 r2 → injval (ALT r1 r2) c v using left by simp
  next
    case right
    have s ∉ L (der c r1) by fact
    then have c # s ∉ L r1 by (simp add: der_correctness Der_def)
    moreover
    have s ∈ der c r2 → v′ by fact
    then have (c # s) ∈ r2 → injval r2 c v′ using IH2 by simp
    ultimately have (c # s) ∈ ALT r1 r2 → injval (ALT r1 r2) c (Right v′)
      by (auto intro: Posix.intros)
    then show (c # s) ∈ ALT r1 r2 → injval (ALT r1 r2) c v using right by simp
  qed
next
  case (SEQ r1 r2)
  have IH1: ⋀s v. s ∈ der c r1 → v ⟹ (c # s) ∈ r1 → injval r1 c v by fact
  have IH2: ⋀s v. s ∈ der c r2 → v ⟹ (c # s) ∈ r2 → injval r2 c v by fact
  have s ∈ der c (SEQ r1 r2) → v by fact
```

then consider
(*left_nullable*) *v1 v2 s1 s2* **where**
v = *Left* (*Seq v1 v2*) *s* = *s1* @ *s2*
s1 ∈ *der c r1* → *v1 s2* ∈ *r2* → *v2 nullable r1*
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in L\ (der\ c\ r1) \wedge s_4 \in L\ r2)$
| (*right_nullable*) *v1 s1 s2* **where**
v = *Right v1 s* = *s1* @ *s2*
s ∈ *der c r2* → *v1 nullable r1 s1* @ *s2* ∉ *L* (*SEQ* (*der c r1*) *r2*)
| (*not_nullable*) *v1 v2 s1 s2* **where**
v = *Seq v1 v2 s* = *s1* @ *s2*
s1 ∈ *der c r1* → *v1 s2* ∈ *r2* → *v2* ¬*nullable r1*
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in L\ (der\ c\ r1) \wedge s_4 \in L\ r2)$
by (*force split*: *if_splits elim*!: *Posix_elims simp add*: *Sequ_def der_correctness Der_def*)
then show (*c* # *s*) ∈ *SEQ r1 r2* → *injval* (*SEQ r1 r2*) *c v*
proof (*cases*)
case *left_nullable*
have *s1* ∈ *der c r1* → *v1* **by** *fact*
then have (*c* # *s1*) ∈ *r1* → *injval r1 c v1* **using** *IH1* **by** *simp*
moreover
have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in L\ (der\ c\ r1) \wedge s_4 \in L\ r2)$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in L\ r1 \wedge s_4 \in L\ r2)$ **by** (*simp add*: *der_correctness Der_def*)
ultimately have ((*c* # *s1*) @ *s2*) ∈ *SEQ r1 r2* → *Seq* (*injval r1 c v1*) *v2* **using** *left_nullable* **by** (*rule_tac Posix.intros*)
then show (*c* # *s*) ∈ *SEQ r1 r2* → *injval* (*SEQ r1 r2*) *c v* **using** *left_nullable* **by** *simp*
next
case *right_nullable*
have *nullable r1* **by** *fact*
then have [] ∈ *r1* → (*mkeps r1*) **by** (*rule Posix_mkeps*)
moreover
have *s* ∈ *der c r2* → *v1* **by** *fact*
then have (*c* # *s*) ∈ *r2* → (*injval r2 c v1*) **using** *IH2* **by** *simp*
moreover
have *s1* @ *s2* ∉ *L* (*SEQ* (*der c r1*) *r2*) **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = c \# s \wedge [] @ s_3 \in L\ r1 \wedge s_4 \in L\ r2)$ **using** *right_nullable*
by(*auto simp add*: *der_correctness Der_def append_eq_Cons_conv Sequ_def*)
ultimately have ([] @ (*c* # *s*)) ∈ *SEQ r1 r2* → *Seq* (*mkeps r1*) (*injval r2 c v1*)
by(*rule Posix.intros*)
then show (*c* # *s*) ∈ *SEQ r1 r2* → *injval* (*SEQ r1 r2*) *c v* **using** *right_nullable* **by** *simp*
next

case *not_nullable*
have $s1 \in der\ c\ r1 \rightarrow v1$ **by** *fact*
then have $(c \# s1) \in r1 \rightarrow injval\ r1\ c\ v1$ **using** *IH1* **by** *simp*
moreover
have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in L\ (der\ c\ r1) \wedge s_4 \in L\ r2)$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in L\ r1 \wedge s_4 \in L\ r2)$ **by** (*simp add*: *der_correctness Der_def*)
ultimately have $((c \# s1) @ s2) \in SEQ\ r1\ r2 \rightarrow Seq\ (injval\ r1\ c\ v1)\ v2$ **using** *not_nullable*
by (*rule_tac Posix.intros*) (*simp_all*)
then show $(c \# s) \in SEQ\ r1\ r2 \rightarrow injval\ (SEQ\ r1\ r2)\ c\ v$ **using** *not_nullable* **by** *simp*
qed
next
case (*STAR r*)
have *IH*: $\bigwedge s\ v.\ s \in der\ c\ r \rightarrow v \Longrightarrow (c \# s) \in r \rightarrow injval\ r\ c\ v$ **by** *fact*
have $s \in der\ c\ (STAR\ r) \rightarrow v$ **by** *fact*
then consider
(*cons*) *v1 vs s1 s2* **where**
$v = Seq\ v1\ (Stars\ vs)\ s = s1 @ s2$
$s1 \in der\ c\ r \rightarrow v1\ s2 \in (STAR\ r) \rightarrow (Stars\ vs)$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in L\ (der\ c\ r) \wedge s_4 \in L\ (STAR\ r))$
apply(*auto elim*!: *Posix_elims*(*1*−*5*) *simp add*: *der_correctness Der_def intro*: *Posix.intros*)
apply(*rotate_tac 3*)
apply(*erule_tac Posix_elims*(*6*))
apply (*simp add*: *Posix.intros*(*6*))
using *Posix.intros*(*7*) **by** *blast*
then show $(c \# s) \in STAR\ r \rightarrow injval\ (STAR\ r)\ c\ v$
proof (*cases*)
case *cons*
have $s1 \in der\ c\ r \rightarrow v1$ **by** *fact*
then have $(c \# s1) \in r \rightarrow injval\ r\ c\ v1$ **using** *IH* **by** *simp*
moreover
have $s2 \in STAR\ r \rightarrow Stars\ vs$ **by** *fact*
moreover
have $(c \# s1) \in r \rightarrow injval\ r\ c\ v1$ **by** *fact*
then have $flat\ (injval\ r\ c\ v1) = (c \# s1)$ **by** (*rule Posix1*)
then have $flat\ (injval\ r\ c\ v1) \neq []$ **by** *simp*
moreover
have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in L\ (der\ c\ r) \wedge s_4 \in L\ (STAR\ r))$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in L\ r \wedge s_4 \in L\ (STAR\ r))$

```
        by (simp add: der_correctness Der_def)
      ultimately
     have ((c # s1) @ s2) ∈ STAR r → Stars (injval r c v1 # vs) by (rule Posix.intros)
      then show (c # s) ∈ STAR r → injval (STAR r) c v using cons by(simp)
   qed
qed
```

15 Lexer Correctness

```
lemma lexer_correct_None:
  shows s ∉ L r ⟷ lexer r s = None
  apply(induct s arbitrary: r)
  apply(simp)
  apply(simp add: nullable_correctness)
  apply(simp)
  apply(drule_tac x=der a r in meta_spec)
  apply(auto)
  apply(auto simp add: der_correctness Der_def)
done

lemma lexer_correct_Some:
  shows s ∈ L r ⟷ (∃ v. lexer r s = Some(v) ∧ s ∈ r → v)
  apply(induct s arbitrary : r)
  apply(simp only: lexer.simps)
  apply(simp)
  apply(simp add: nullable_correctness Posix_mkeps)
  apply(drule_tac x=der a r in meta_spec)
  apply(simp (no_asm_use) add: der_correctness Der_def del: lexer.simps)
  apply(simp del: lexer.simps)
  apply(simp only: lexer.simps)
  apply(case_tac lexer (der a r) s = None)
   apply(auto)[1]
  apply(simp)
  apply(erule exE)
  apply(simp)
  apply(rule iffI)
  apply(simp add: Posix_injval)
  apply(simp add: Posix1(1))
done

lemma lexer_correctness:
  shows (lexer r s = Some v) ⟷ s ∈ r → v
  and   (lexer r s = None) ⟷ ¬(∃ v. s ∈ r → v)
using Posix1(1) Posix_determ lexer_correct_None lexer_correct_Some apply fastforce
using Posix1(1) lexer_correct_None lexer_correct_Some by blast
```

fun *flex* :: *rexp* ⇒ (*val* ⇒ *val*) => *string* ⇒ (*val* ⇒ *val*)
  where
  flex r f [] = *f*
| *flex r f* (*c*#*s*) = *flex* (*der c r*) (λ*v*. *f* (*injval r c v*)) *s*

lemma *flex_fun_apply*:
  shows *g* (*flex r f s v*) = *flex r* (*g o f*) *s v*
  apply(*induct s arbitrary*: *g f r v*)
  apply(*simp_all add*: *comp_def*)
  by *meson*

lemma *flex_append*:
  shows *flex r f* (*s1* @ *s2*) = *flex* (*ders s1 r*) (*flex r f s1*) *s2*
  apply(*induct s1 arbitrary*: *s2 r f*)
  apply(*simp_all*)
  done

lemma *lexer_flex*:
  shows *lexer r s* = (*if nullable* (*ders s r*)
                then Some(*flex r id s* (*mkeps* (*ders s r*))) *else None*)
  apply(*induct s arbitrary*: *r*)
  apply(*simp_all add*: *flex_fun_apply*)
  done

unused-thms

end
theory *Simplifying*
  imports *Lexer*
begin

16 Lexer including simplifications

fun *F_RIGHT* **where**
  F_RIGHT f v = *Right* (*f v*)

fun *F_LEFT* **where**
  F_LEFT f v = *Left* (*f v*)

fun *F_ALT* **where**
  F_ALT f_1 f_2 (*Right v*) = *Right* (f_2 *v*)

```
| F_ALT f_1 f_2 (Left v) = Left (f_1 v)
| F_ALT f1 f2 v = v


fun F_SEQ1 where
  F_SEQ1 f_1 f_2 v = Seq (f_1 Void) (f_2 v)

fun F_SEQ2 where
  F_SEQ2 f_1 f_2 v = Seq (f_1 v) (f_2 Void)

fun F_SEQ where
  F_SEQ f_1 f_2 (Seq v_1 v_2) = Seq (f_1 v_1) (f_2 v_2)
| F_SEQ f1 f2 v = v

fun simp_ALT where
  simp_ALT (ZERO, f_1) (r_2, f_2) = (r_2, F_RIGHT f_2)
| simp_ALT (r_1, f_1) (ZERO, f_2) = (r_1, F_LEFT f_1)
| simp_ALT (r_1, f_1) (r_2, f_2) = (ALT r_1 r_2, F_ALT f_1 f_2)


fun simp_SEQ where
  simp_SEQ (ONE, f_1) (r_2, f_2) = (r_2, F_SEQ1 f_1 f_2)
| simp_SEQ (r_1, f_1) (ONE, f_2) = (r_1, F_SEQ2 f_1 f_2)
| simp_SEQ (ZERO, f_1) (r_2, f_2) = (ZERO, undefined)
| simp_SEQ (r_1, f_1) (ZERO, f_2) = (ZERO, undefined)
| simp_SEQ (r_1, f_1) (r_2, f_2) = (SEQ r_1 r_2, F_SEQ f_1 f_2)

lemma simp_SEQ_simps[simp]:
  simp_SEQ p1 p2 = (if (fst p1 = ONE) then (fst p2, F_SEQ1 (snd p1) (snd p2))
                  else (if (fst p2 = ONE) then (fst p1, F_SEQ2 (snd p1) (snd p2))
                  else (if (fst p1 = ZERO) then (ZERO, undefined)
                  else (if (fst p2 = ZERO) then (ZERO, undefined)
                  else (SEQ (fst p1) (fst p2), F_SEQ (snd p1) (snd p2))))))
by (induct p1 p2 rule: simp_SEQ.induct) (auto)

lemma simp_ALT_simps[simp]:
  simp_ALT p1 p2 = (if (fst p1 = ZERO) then (fst p2, F_RIGHT (snd p2))
                  else (if (fst p2 = ZERO) then (fst p1, F_LEFT (snd p1))
                  else (ALT (fst p1) (fst p2), F_ALT (snd p1) (snd p2))))
by (induct p1 p2 rule: simp_ALT.induct) (auto)

fun
  simp :: rexp ⇒ rexp ∗ (val ⇒ val)
where
  simp (ALT r1 r2) = simp_ALT (simp r1) (simp r2)
```

```
| simp (SEQ r1 r2) = simp_SEQ (simp r1) (simp r2)
| simp r = (r, id)

fun
  slexer :: rexp ⇒ string ⇒ val option
where
  slexer r [] = (if nullable r then Some(mkeps r) else None)
| slexer r (c#s) = (let (rs, fr) = simp (der c r) in
                      (case (slexer rs s) of
                        None ⇒ None
                      | Some(v) ⇒ Some(injval r c (fr v))))


lemma slexer_better_simp:
  slexer r (c#s) = (case (slexer (fst (simp (der c r))) s) of
                        None ⇒ None
                      | Some(v) ⇒ Some(injval r c ((snd (simp (der c r))) v)))
by (auto split: prod.split option.split)


lemma L_fst_simp:
  shows L(r) = L(fst (simp r))
by (induct r) (auto)

lemma Posix_simp:
  assumes s ∈ (fst (simp r)) → v
  shows s ∈ r → ((snd (simp r)) v)
using assms
proof(induct r arbitrary: s v rule: rexp.induct)
  case (ALT r1 r2 s v)
  have IH1: ⋀s v. s ∈ fst (simp r1) → v ⟹ s ∈ r1 → snd (simp r1) v by fact
  have IH2: ⋀s v. s ∈ fst (simp r2) → v ⟹ s ∈ r2 → snd (simp r2) v by fact
  have as: s ∈ fst (simp (ALT r1 r2)) → v by fact
  consider (ZERO_ZERO) fst (simp r1) = ZERO fst (simp r2) = ZERO
        | (ZERO_NZERO) fst (simp r1) = ZERO fst (simp r2) ≠ ZERO
        | (NZERO_ZERO) fst (simp r1) ≠ ZERO fst (simp r2) = ZERO
        | (NZERO_NZERO) fst (simp r1) ≠ ZERO fst (simp r2) ≠ ZERO by auto
  then show s ∈ ALT r1 r2 → snd (simp (ALT r1 r2)) v
    proof(cases)
      case (ZERO_ZERO)
      with as have s ∈ ZERO → v by simp
      then show s ∈ ALT r1 r2 → snd (simp (ALT r1 r2)) v by (rule Posix_elims(1))
    next
      case (ZERO_NZERO)
      with as have s ∈ fst (simp r2) → v by simp
```

```
      with IH2 have s ∈ r2 → snd (simp r2) v by simp
      moreover
      from ZERO_NZERO have fst (simp r1) = ZERO by simp
      then have L (fst (simp r1)) = {} by simp
      then have L r1 = {} using L_fst_simp by simp
      then have s ∉ L r1 by simp
      ultimately have s ∈ ALT r1 r2 → Right (snd (simp r2) v) by (rule Posix_ALT2)
      then show s ∈ ALT r1 r2 → snd (simp (ALT r1 r2)) v
      using ZERO_NZERO by simp
    next
      case (NZERO_ZERO)
      with as have s ∈ fst (simp r1) → v by simp
      with IH1 have s ∈ r1 → snd (simp r1) v by simp
      then have s ∈ ALT r1 r2 → Left (snd (simp r1) v) by (rule Posix_ALT1)
       then show s ∈ ALT r1 r2 → snd (simp (ALT r1 r2)) v using NZERO_ZERO by
simp
    next
      case (NZERO_NZERO)
      with as have s ∈ ALT (fst (simp r1)) (fst (simp r2)) → v by simp
      then consider (Left) v1 where v = Left v1 s ∈ (fst (simp r1)) → v1
                  | (Right) v2 where v = Right v2 s ∈ (fst (simp r2)) → v2 s ∉ L (fst (simp
r1))
                 by (erule_tac Posix_elims(4))
      then show s ∈ ALT r1 r2 → snd (simp (ALT r1 r2)) v
      proof(cases)
        case (Left)
        then have v = Left v1 s ∈ r1 → (snd (simp r1) v1) using IH1 by simp_all
        then show s ∈ ALT r1 r2 → snd (simp (ALT r1 r2)) v using NZERO_NZERO
          by (simp_all add: Posix_ALT1)
      next
        case (Right)
       then have v = Right v2 s ∈ r2 → (snd (simp r2) v2) s ∉ L r1 using IH2 L_fst_simp
by simp_all
        then show s ∈ ALT r1 r2 → snd (simp (ALT r1 r2)) v using NZERO_NZERO
          by (simp_all add: Posix_ALT2)
      qed
    qed
next
  case (SEQ r1 r2 s v)
  have IH1: ⋀s v. s ∈ fst (simp r1) → v ⟹ s ∈ r1 → snd (simp r1) v by fact
  have IH2: ⋀s v. s ∈ fst (simp r2) → v ⟹ s ∈ r2 → snd (simp r2) v by fact
  have as: s ∈ fst (simp (SEQ r1 r2)) → v by fact
  consider (ONE_ONE) fst (simp r1) = ONE fst (simp r2) = ONE
       | (ONE_NONE) fst (simp r1) = ONE fst (simp r2) ≠ ONE
       | (NONE_ONE) fst (simp r1) ≠ ONE fst (simp r2) = ONE
```

```
      | (NONE_NONE) fst (simp r1) ≠ ONE fst (simp r2) ≠ ONE
      by auto
  then show s ∈ SEQ r1 r2 → snd (simp (SEQ r1 r2)) v
  proof(cases)
    case (ONE_ONE)
    with as have b: s ∈ ONE → v by simp
    from b have s ∈ r1 → snd (simp r1) v using IH1 ONE_ONE by simp
    moreover
    from b have c: s = [] v = Void using Posix_elims(2) by auto
    moreover
    have [] ∈ ONE → Void by (simp add: Posix_ONE)
    then have [] ∈ fst (simp r2) → Void using ONE_ONE by simp
    then have [] ∈ r2 → snd (simp r2) Void using IH2 by simp
    ultimately have ([] @ []) ∈ SEQ r1 r2 → Seq (snd (simp r1) Void) (snd (simp r2) Void)
      using Posix_SEQ by blast
    then show s ∈ SEQ r1 r2 → snd (simp (SEQ r1 r2)) v using c ONE_ONE by simp
   next
    case (ONE_NONE)
    with as have b: s ∈ fst (simp r2) → v by simp
    from b have s ∈ r2 → snd (simp r2) v using IH2 ONE_NONE by simp
    moreover
    have [] ∈ ONE → Void by (simp add: Posix_ONE)
    then have [] ∈ fst (simp r1) → Void using ONE_NONE by simp
    then have [] ∈ r1 → snd (simp r1) Void using IH1 by simp
    moreover
    from ONE_NONE(1) have L (fst (simp r1)) = {[]} by simp
    then have L r1 = {[]} by (simp add: L_fst_simp[symmetric])
    ultimately have ([] @ s) ∈ SEQ r1 r2 → Seq (snd (simp r1) Void) (snd (simp r2) v)
      by(rule_tac Posix_SEQ) auto
    then show s ∈ SEQ r1 r2 → snd (simp (SEQ r1 r2)) v using ONE_NONE by simp
   next
    case (NONE_ONE)
      with as have s ∈ fst (simp r1) → v by simp
      with IH1 have s ∈ r1 → snd (simp r1) v by simp
    moreover
      have [] ∈ ONE → Void by (simp add: Posix_ONE)
      then have [] ∈ fst (simp r2) → Void using NONE_ONE by simp
      then have [] ∈ r2 → snd (simp r2) Void using IH2 by simp
     ultimately have (s @ []) ∈ SEQ r1 r2 → Seq (snd (simp r1) v) (snd (simp r2) Void)
      by(rule_tac Posix_SEQ) auto
    then show s ∈ SEQ r1 r2 → snd (simp (SEQ r1 r2)) v using NONE_ONE by simp
   next
```

```
   case (NONE_NONE)
   from as have 00: fst (simp r1) ≠ ZERO fst (simp r2) ≠ ZERO
    apply(auto)
    apply(smt Posix_elims(1) fst_conv)
    by (smt NONE_NONE(2) Posix_elims(1) fstI)
   with NONE_NONE as have s ∈ SEQ (fst (simp r1)) (fst (simp r2)) → v by simp
   then obtain s1 s2 v1 v2 where eqs: s = s1 @ s2 v = Seq v1 v2
          s1 ∈ (fst (simp r1)) → v1 s2 ∈ (fst (simp r2)) → v2
          ¬ (∃s3 s4. s3 ≠ [] ∧ s3 @ s4 = s2 ∧ s1 @ s3 ∈ L r1 ∧ s4 ∈ L r2)
          by (erule_tac Posix_elims(5)) (auto simp add: L_fst_simp[symmetric])
   then have s1 ∈ r1 → (snd (simp r1) v1) s2 ∈ r2 → (snd (simp r2) v2)
    using IH1 IH2 by auto
    then show s ∈ SEQ r1 r2 → snd (simp (SEQ r1 r2)) v using eqs NONE_NONE 00
    by(auto intro: Posix_SEQ)
  qed
qed (simp_all)


lemma slexer_correctness:
 shows slexer r s = lexer r s
proof(induct s arbitrary: r)
 case Nil
 show slexer r [] = lexer r [] by simp
next
 case (Cons c s r)
 have IH: ⋀r. slexer r s = lexer r s by fact
 show slexer r (c # s) = lexer r (c # s)
  proof (cases s ∈ L (der c r))
   case True
    assume a1: s ∈ L (der c r)
    then obtain v1 where a2: lexer (der c r) s = Some v1 s ∈ der c r → v1
     using lexer_correct_Some by auto
    from a1 have s ∈ L (fst (simp (der c r))) using L_fst_simp[symmetric] by simp
    then obtain v2 where a3: lexer (fst (simp (der c r))) s = Some v2 s ∈ (fst (simp (der c r))) → v2
      using lexer_correct_Some by auto
    then have a4: slexer (fst (simp (der c r))) s = Some v2 using IH by simp
     from a3(2) have s ∈ der c r → (snd (simp (der c r))) v2 using Posix_simp by simp
    with a2(2) have v1 = (snd (simp (der c r))) v2 using Posix_determ by simp
    with a2(1) a4 show slexer r (c # s) = lexer r (c # s) by (auto split: prod.split)
   next
   case False
    assume b1: s ∉ L (der c r)
```

```
      then have lexer (der c r) s = None using lexer_correct_None by simp
      moreover
      from b1 have s ∉ L (fst (simp (der c r))) using L_fst_simp[symmetric] by simp
      then have lexer (fst (simp (der c r))) s = None using lexer_correct_None by simp
      then have slexer (fst (simp (der c r))) s = None using IH by simp
      ultimately show slexer r (c # s) = lexer r (c # s)
        by (simp del: slexer.simps add: slexer_better_simp)
  qed
 qed

end

theory Positions
  imports Spec Lexer
begin
```

17 Positions in Values

```
fun
  at :: val ⇒ nat list ⇒ val
where
  at v [] = v
| at (Left v) (0#ps)= at v ps
| at (Right v) (Suc 0#ps)= at v ps
| at (Seq v1 v2) (0#ps)= at v1 ps
| at (Seq v1 v2) (Suc 0#ps)= at v2 ps
| at (Stars vs) (n#ps)= at (nth vs n) ps
```

```
fun Pos :: val ⇒ (nat list) set
where
  Pos (Void) = {[]}
| Pos (Char c) = {[]}
| Pos (Left v) = {[]} ∪ {0#ps | ps. ps ∈ Pos v}
| Pos (Right v) = {[]} ∪ {1#ps | ps. ps ∈ Pos v}
| Pos (Seq v1 v2) = {[]} ∪ {0#ps | ps. ps ∈ Pos v1} ∪ {1#ps | ps. ps ∈ Pos v2}
| Pos (Stars []) = {[]}
| Pos (Stars (v#vs)) = {[]} ∪ {0#ps | ps. ps ∈ Pos v} ∪ {Suc n#ps | n ps. n#ps ∈
Pos (Stars vs)}
```

```
lemma Pos_stars:
  Pos (Stars vs) = {[]} ∪ (⋃n < length vs. {n#ps | ps. ps ∈ Pos (vs ! n)})
apply(induct vs)
```

```
apply(auto simp add: insert_ident less_Suc_eq_0_disj)
done

lemma Pos_empty:
  shows [] ∈ Pos v
by (induct v rule: Pos.induct)(auto)


abbreviation
  intlen vs ≝ int (length vs)


definition pflat_len :: val ⇒ nat list => int
where
  pflat_len v p ≝ (if p ∈ Pos v then intlen (flat (at v p)) else −1)

lemma pflat_len_simps:
  shows pflat_len (Seq v1 v2) (0#p) = pflat_len v1 p
  and   pflat_len (Seq v1 v2) (Suc 0#p) = pflat_len v2 p
  and   pflat_len (Left v) (0#p) = pflat_len v p
  and   pflat_len (Left v) (Suc 0#p) = −1
  and   pflat_len (Right v) (Suc 0#p) = pflat_len v p
  and   pflat_len (Right v) (0#p) = −1
  and   pflat_len (Stars (v#vs)) (Suc n#p) = pflat_len (Stars vs) (n#p)
  and   pflat_len (Stars (v#vs)) (0#p) = pflat_len v p
  and   pflat_len v [] = intlen (flat v)
by (auto simp add: pflat_len_def Pos_empty)

lemma pflat_len_Stars_simps:
  assumes n < length vs
  shows pflat_len (Stars vs) (n#p) = pflat_len (vs!n) p
using assms
apply(induct vs arbitrary: n p)
apply(auto simp add: less_Suc_eq_0_disj pflat_len_simps)
done

lemma pflat_len_outside:
  assumes p ∉ Pos v1
  shows pflat_len v1 p = −1
using assms by (simp add: pflat_len_def)
```

18 Orderings

```
definition prefix_list:: 'a list ⇒ 'a list ⇒ bool (_ ⊑pre _ [60,59] 60)
where
```

$ps1 \sqsubseteq pre\ ps2 \overset{def}{=} \exists ps'.\ ps1\ @ps' = ps2$

definition *sprefix_list*:: $'a\ list \Rightarrow\ 'a\ list \Rightarrow bool\ (\_ \sqsubset spre\ \_\ [60,59]\ 60)$
where
$ps1 \sqsubset spre\ ps2 \overset{def}{=} ps1 \sqsubseteq pre\ ps2 \wedge ps1 \neq ps2$

inductive *lex_list* :: $nat\ list \Rightarrow nat\ list \Rightarrow bool\ (\_ \sqsubset lex\ \_\ [60,59]\ 60)$
where
$[] \sqsubset lex\ (p \# ps)$
$\mid ps1 \sqsubset lex\ ps2 \Longrightarrow (p \# ps1) \sqsubset lex\ (p \# ps2)$
$\mid p1 < p2 \Longrightarrow (p1 \# ps1) \sqsubset lex\ (p2 \# ps2)$

lemma *lex_irrfl*:
fixes *ps1 ps2* :: *nat list*
assumes $ps1 \sqsubset lex\ ps2$
shows $ps1 \neq ps2$
using *assms*
by(*induct rule*: *lex_list.induct*)(*auto*)

lemma *lex_simps* [*simp*]:
fixes *xs ys* :: *nat list*
shows $[] \sqsubset lex\ ys \longleftrightarrow ys \neq []$
and $xs \sqsubset lex\ [] \longleftrightarrow False$
and $(x \# xs) \sqsubset lex\ (y \# ys) \longleftrightarrow (x < y \vee (x = y \wedge xs \sqsubset lex\ ys))$
by (*auto simp add*: *neq_Nil_conv elim*: *lex_list.cases intro*: *lex_list.intros*)

lemma *lex_trans*:
fixes *ps1 ps2 ps3* :: *nat list*
assumes $ps1 \sqsubset lex\ ps2\ ps2 \sqsubset lex\ ps3$
shows $ps1 \sqsubset lex\ ps3$
using *assms*
by (*induct arbitrary*: *ps3 rule*: *lex_list.induct*)
(*auto elim*: *lex_list.cases*)

lemma *lex_trichotomous*:
fixes *p q* :: *nat list*
shows $p = q \vee p \sqsubset lex\ q \vee q \sqsubset lex\ p$
apply(*induct p arbitrary*: *q*)
apply(*auto elim*: *lex_list.cases*)
apply(*case_tac q*)
apply(*auto*)
done

19 POSIX Ordering of Values According to Okui & Suzuki

```
definition PosOrd:: val ⇒ nat list ⇒ val ⇒ bool (_ ⊏val _ _ [60, 60, 59] 60)
where
 v1 ⊏val p v2 ≝ pflat_len v1 p > pflat_len v2 p ∧
         (∀ q ∈ Pos v1 ∪ Pos v2. q ⊏lex p ⟶ pflat_len v1 q = pflat_len v2 q)

lemma PosOrd_def2:
 shows v1 ⊏val p v2 ⟷
     pflat_len v1 p > pflat_len v2 p ∧
     (∀ q ∈ Pos v1. q ⊏lex p ⟶ pflat_len v1 q = pflat_len v2 q) ∧
     (∀ q ∈ Pos v2. q ⊏lex p ⟶ pflat_len v1 q = pflat_len v2 q)
unfolding PosOrd_def
apply(auto)
done


definition PosOrd_ex:: val ⇒ val ⇒ bool (_ :⊏val _ [60, 59] 60)
where
 v1 :⊏val v2 ≝ ∃ p. v1 ⊏val p v2

definition PosOrd_ex_eq:: val ⇒ val ⇒ bool (_ :⊑val _ [60, 59] 60)
where
 v1 :⊑val v2 ≝ v1 :⊏val v2 ∨ v1 = v2


lemma PosOrd_trans:
 assumes v1 :⊏val v2 v2 :⊏val v3
 shows v1 :⊏val v3
proof -
 from assms obtain p p'
  where as: v1 ⊏val p v2 v2 ⊏val p' v3 unfolding PosOrd_ex_def by blast
 then have pos: p ∈ Pos v1 p' ∈ Pos v2 unfolding PosOrd_def pflat_len_def
  by (smt not_int_zless_negative)+
 have p = p' ∨ p ⊏lex p' ∨ p' ⊏lex p
  by (rule lex_trichotomous)
 moreover
  { assume p = p'
   with as have v1 ⊏val p v3 unfolding PosOrd_def pflat_len_def
   by (smt Un_iff)
   then have  v1 :⊏val v3 unfolding PosOrd_ex_def by blast
  }
 moreover
  { assume p ⊏lex p'
   with as have v1 ⊏val p v3 unfolding PosOrd_def pflat_len_def
```

```
    by (smt Un_iff lex_trans)
    then have  v1 :⊏val v3 unfolding PosOrd_ex_def by blast
  }
  moreover
  { assume p′ ⊏lex p
    with as have v1 ⊏val p′ v3 unfolding PosOrd_def
    by (smt Un_iff lex_trans pflat_len_def)
    then have v1 :⊏val v3 unfolding PosOrd_ex_def by blast
  }
  ultimately show v1 :⊏val v3 by blast
qed

lemma PosOrd_irrefl:
  assumes v :⊏val v
  shows False
using assms unfolding PosOrd_ex_def PosOrd_def
by auto

lemma PosOrd_assym:
  assumes v1 :⊏val v2
  shows ¬(v2 :⊏val v1)
using assms
using PosOrd_irrefl PosOrd_trans by blast



lemma PosOrd_ordering:
  shows ordering (λv1 v2. v1 :⊑val v2) (λ v1 v2. v1 :⊏val v2)
unfolding ordering_def PosOrd_ex_eq_def
apply(auto)
using PosOrd_irrefl apply blast
using PosOrd_assym apply blast
using PosOrd_trans by blast

lemma PosOrd_order:
  shows class.order (λv1 v2. v1 :⊑val v2) (λ v1 v2. v1 :⊏val v2)
using PosOrd_ordering
apply(simp add: class.order_def class.preorder_def class.order_axioms_def)
unfolding ordering_def
by blast


lemma PosOrd_ex_eq2:
  shows v1 :⊏val v2 ⟷ (v1 :⊑val v2 ∧ v1 ≠ v2)
using PosOrd_ordering
```

```
unfolding ordering_def
by auto

lemma PosOrdeq_trans:
  assumes v1 :⊑val v2 v2 :⊑val v3
  shows v1 :⊑val v3
using assms PosOrd_ordering
unfolding ordering_def
by blast

lemma PosOrdeq_antisym:
  assumes v1 :⊑val v2 v2 :⊑val v1
  shows v1 = v2
using assms PosOrd_ordering
unfolding ordering_def
by blast

lemma PosOrdeq_refl:
  shows v :⊑val v
unfolding PosOrd_ex_eq_def
by auto


lemma PosOrd_shorterE:
  assumes v1 :⊏val v2
  shows length (flat v2) ≤ length (flat v1)
using assms unfolding PosOrd_ex_def PosOrd_def
apply(auto)
apply(case_tac p)
apply(simp add: pflat_len_simps)
apply(drule_tac x=[] in bspec)
apply(simp add: Pos_empty)
apply(simp add: pflat_len_simps)
done

lemma PosOrd_shorterI:
  assumes length (flat v2) < length (flat v1)
  shows v1 :⊏val v2
unfolding PosOrd_ex_def PosOrd_def pflat_len_def
using assms Pos_empty by force

lemma PosOrd_spreI:
  assumes flat v′ ⊏spre flat v
  shows v :⊏val v′
using assms
```

```
apply(rule_tac PosOrd_shorterI)
unfolding prefix_list_def sprefix_list_def
by (metis append_Nil2 append_eq_conv_conj drop_all le_less_linear)

lemma pflat_len_inside:
  assumes pflat_len v2 p < pflat_len v1 p
  shows p ∈ Pos v1
using assms
unfolding pflat_len_def
by (auto split: if_splits)


lemma PosOrd_Left_Right:
  assumes flat v1 = flat v2
  shows Left v1 :⊏val Right v2
unfolding PosOrd_ex_def
apply(rule_tac x=[0] in exI)
apply(auto simp add: PosOrd_def pflat_len_simps assms)
done

lemma PosOrd_LeftE:
  assumes Left v1 :⊏val Left v2 flat v1 = flat v2
  shows v1 :⊏val v2
using assms
unfolding PosOrd_ex_def PosOrd_def2
apply(auto simp add: pflat_len_simps)
apply(frule pflat_len_inside)
apply(auto simp add: pflat_len_simps)
by (metis lex_simps(3) pflat_len_simps(3))

lemma PosOrd_LeftI:
  assumes v1 :⊏val v2 flat v1 = flat v2
  shows  Left v1 :⊏val Left v2
using assms
unfolding PosOrd_ex_def PosOrd_def2
apply(auto simp add: pflat_len_simps)
by (metis less_numeral_extra(3) lex_simps(3) pflat_len_simps(3))

lemma PosOrd_Left_eq:
  assumes flat v1 = flat v2
  shows Left v1 :⊏val Left v2 ⟷ v1 :⊏val v2
using assms PosOrd_LeftE PosOrd_LeftI
by blast
```

```
lemma PosOrd_RightE:
  assumes Right v1 :⊏val Right v2 flat v1 = flat v2
  shows v1 :⊏val v2
using assms
unfolding PosOrd_ex_def PosOrd_def2
apply(auto simp add: pflat_len_simps)
apply(frule pflat_len_inside)
apply(auto simp add: pflat_len_simps)
by (metis lex_simps(3) pflat_len_simps(5))

lemma PosOrd_RightI:
  assumes v1 :⊏val v2 flat v1 = flat v2
  shows  Right v1 :⊏val Right v2
using assms
unfolding PosOrd_ex_def PosOrd_def2
apply(auto simp add: pflat_len_simps)
by (metis lex_simps(3) nat_neq_iff pflat_len_simps(5))


lemma PosOrd_Right_eq:
  assumes flat v1 = flat v2
  shows Right v1 :⊏val Right v2 ⟷ v1 :⊏val v2
using assms PosOrd_RightE PosOrd_RightI
by blast


lemma PosOrd_SeqI1:
  assumes v1 :⊏val w1 flat (Seq v1 v2) = flat (Seq w1 w2)
  shows Seq v1 v2 :⊏val Seq w1 w2
using assms(1)
apply(subst (asm) PosOrd_ex_def)
apply(subst (asm) PosOrd_def)
apply(clarify)
apply(subst PosOrd_ex_def)
apply(rule_tac x=0#p in exI)
apply(subst PosOrd_def)
apply(rule conjI)
apply(simp add: pflat_len_simps)
apply(rule ballI)
apply(rule impI)
apply(simp only: Pos.simps)
apply(auto)[1]
apply(simp add: pflat_len_simps)
apply(auto simp add: pflat_len_simps)
using assms(2)
```

```
apply(simp)
apply(metis length_append of_nat_add)
done

lemma PosOrd_SeqI2:
  assumes v2 :⊏val w2 flat v2 = flat w2
  shows Seq v v2 :⊏val Seq v w2
using assms(1)
apply(subst (asm) PosOrd_ex_def)
apply(subst (asm) PosOrd_def)
apply(clarify)
apply(subst PosOrd_ex_def)
apply(rule_tac x=Suc 0#p in exI)
apply(subst PosOrd_def)
apply(rule conjI)
apply(simp add: pflat_len_simps)
apply(rule ballI)
apply(rule impI)
apply(simp only: Pos.simps)
apply(auto)[1]
apply(simp add: pflat_len_simps)
using assms(2)
apply(simp)
apply(auto simp add: pflat_len_simps)
done

lemma PosOrd_Seq_eq:
  assumes flat v2 = flat w2
  shows (Seq v v2) :⊏val (Seq v w2) ⟷ v2 :⊏val w2
using assms
apply(auto)
prefer 2
apply(simp add: PosOrd_SeqI2)
apply(simp add: PosOrd_ex_def)
apply(auto)
apply(case_tac p)
apply(simp add: PosOrd_def pflat_len_simps)
apply(case_tac a)
apply(simp add: PosOrd_def pflat_len_simps)
apply(clarify)
apply(case_tac nat)
prefer 2
apply(simp add: PosOrd_def pflat_len_simps pflat_len_outside)
apply(rule_tac x=list in exI)
apply(auto simp add: PosOrd_def2 pflat_len_simps)
```

```
apply(smt Collect_disj_eq lex_list.intros(2) mem_Collect_eq pflat_len_simps(2))
apply(smt Collect_disj_eq lex_list.intros(2) mem_Collect_eq pflat_len_simps(2))
done
```

```
lemma PosOrd_StarsI:
  assumes v1 :⊏val v2 flats (v1#vs1) = flats (v2#vs2)
  shows Stars (v1#vs1) :⊏val Stars (v2#vs2)
using assms(1)
apply(subst (asm) PosOrd_ex_def)
apply(subst (asm) PosOrd_def)
apply(clarify)
apply(subst PosOrd_ex_def)
apply(subst PosOrd_def)
apply(rule_tac x=0#p in exI)
apply(simp add: pflat_len_Stars_simps pflat_len_simps)
using assms(2)
apply(simp add: pflat_len_simps)
apply(auto simp add: pflat_len_Stars_simps pflat_len_simps)
by (metis length_append of_nat_add)

lemma PosOrd_StarsI2:
  assumes Stars vs1 :⊏val Stars vs2 flats vs1 = flats vs2
  shows Stars (v#vs1) :⊏val Stars (v#vs2)
using assms(1)
apply(subst (asm) PosOrd_ex_def)
apply(subst (asm) PosOrd_def)
apply(clarify)
apply(subst PosOrd_ex_def)
apply(subst PosOrd_def)
apply(case_tac p)
apply(simp add: pflat_len_simps)
apply(rule_tac x=Suc a#list in exI)
apply(auto simp add: pflat_len_Stars_simps pflat_len_simps assms(2))
done

lemma PosOrd_Stars_appendI:
  assumes Stars vs1 :⊏val Stars vs2 flat (Stars vs1) = flat (Stars vs2)
  shows Stars (vs @ vs1) :⊏val Stars (vs @ vs2)
using assms
apply(induct vs)
apply(simp)
apply(simp add: PosOrd_StarsI2)
done
```

```
lemma PosOrd_StarsE2:
  assumes Stars (v # vs1) :⊏val Stars (v # vs2)
  shows Stars vs1 :⊏val Stars vs2
using assms
apply(subst (asm) PosOrd_ex_def)
apply(erule exE)
apply(case_tac p)
apply(simp)
apply(simp add: PosOrd_def pflat_len_simps)
apply(subst PosOrd_ex_def)
apply(rule_tac x=[] in exI)
apply(simp add: PosOrd_def pflat_len_simps Pos_empty)
apply(simp)
apply(case_tac a)
apply(clarify)
apply(auto simp add: pflat_len_simps PosOrd_def pflat_len_def split: if_splits)[1]
apply(clarify)
apply(simp add: PosOrd_ex_def)
apply(rule_tac x=nat#list in exI)
apply(auto simp add: PosOrd_def pflat_len_simps)[1]
apply(case_tac q)
apply(simp add: PosOrd_def pflat_len_simps)
apply(clarify)
apply(drule_tac x=Suc a # lista in bspec)
apply(simp)
apply(auto simp add: PosOrd_def pflat_len_simps)[1]
apply(case_tac q)
apply(simp add: PosOrd_def pflat_len_simps)
apply(clarify)
apply(drule_tac x=Suc a # lista in bspec)
apply(simp)
apply(auto simp add: PosOrd_def pflat_len_simps)[1]
done

lemma PosOrd_Stars_appendE:
  assumes Stars (vs @ vs1) :⊏val Stars (vs @ vs2)
  shows Stars vs1 :⊏val Stars vs2
using assms
apply(induct vs)
apply(simp)
apply(simp add: PosOrd_StarsE2)
done

lemma PosOrd_Stars_append_eq:
```

```
  assumes flats vs1 = flats vs2
  shows Stars (vs @ vs1) :⊏val Stars (vs @ vs2) ⟷ Stars vs1 :⊏val Stars vs2
using assms
apply(rule_tac iffI)
apply(erule PosOrd_Stars_appendE)
apply(rule PosOrd_Stars_appendI)
apply(auto)
done

lemma PosOrd_almost_trichotomous:
  shows v1 :⊏val v2 ∨ v2 :⊏val v1 ∨ (length (flat v1) = length (flat v2))
apply(auto simp add: PosOrd_ex_def)
apply(auto simp add: PosOrd_def)
apply(rule_tac x=[] in exI)
apply(auto simp add: Pos_empty pflat_len_simps)
apply(drule_tac x=[] in spec)
apply(auto simp add: Pos_empty pflat_len_simps)
done
```

20 The Posix Value is smaller than any other Value

```
lemma Posix_PosOrd:
  assumes s ∈ r → v1 v2 ∈ LV r s
  shows v1 :⊑val v2
using assms
proof (induct arbitrary: v2 rule: Posix.induct)
  case (Posix_ONE v)
  have v ∈ LV ONE [] by fact
  then have v = Void
    by (simp add: LV_simps)
  then show Void :⊑val v
    by (simp add: PosOrd_ex_eq_def)
next
  case (Posix_CHAR c v)
  have v ∈ LV (CHAR c) [c] by fact
  then have v = Char c
    by (simp add: LV_simps)
  then show Char c :⊑val v
    by (simp add: PosOrd_ex_eq_def)
next
  case (Posix_ALT1 s r1 v r2 v2)
  have as1: s ∈ r1 → v by fact
  have IH: ⋀v2. v2 ∈ LV r1 s ⟹ v :⊑val v2 by fact
  have v2 ∈ LV (ALT r1 r2) s by fact
  then have ⊨ v2 : ALT r1 r2 flat v2 = s
```

```
   by(auto simp add: LV_def prefix_list_def)
  then consider
   (Left) v3 where v2 = Left v3 ⊨ v3 : r1 flat v3 = s
  | (Right) v3 where v2 = Right v3 ⊨ v3 : r2 flat v3 = s
  by (auto elim: Prf.cases)
  then show Left v :⊑val v2
  proof(cases)
    case (Left v3)
    have v3 ∈ LV r1 s using Left(2,3)
      by (auto simp add: LV_def prefix_list_def)
    with IH have v :⊑val v3 by simp
    moreover
    have flat v3 = flat v using as1 Left(3)
      by (simp add: Posix1(2))
    ultimately have Left v :⊑val Left v3
      by (simp add: PosOrd_ex_eq_def PosOrd_Left_eq)
    then show Left v :⊑val v2 unfolding Left .
  next
    case (Right v3)
    have flat v3 = flat v using as1 Right(3)
      by (simp add: Posix1(2))
    then have Left v :⊑val Right v3
      unfolding PosOrd_ex_eq_def
      by (simp add: PosOrd_Left_Right)
    then show Left v :⊑val v2 unfolding Right .
  qed
next
  case (Posix_ALT2 s r2 v r1 v2)
  have as1: s ∈ r2 → v by fact
  have as2: s ∉ L r1 by fact
  have IH: ⋀v2. v2 ∈ LV r2 s ⟹ v :⊑val v2 by fact
  have v2 ∈ LV (ALT r1 r2) s by fact
  then have ⊨ v2 : ALT r1 r2 flat v2 = s
   by(auto simp add: LV_def prefix_list_def)
  then consider
   (Left) v3 where v2 = Left v3 ⊨ v3 : r1 flat v3 = s
  | (Right) v3 where v2 = Right v3 ⊨ v3 : r2 flat v3 = s
  by (auto elim: Prf.cases)
  then show Right v :⊑val v2
  proof (cases)
   case (Right v3)
    have v3 ∈ LV r2 s using Right(2,3)
      by (auto simp add: LV_def prefix_list_def)
    with IH have v :⊑val v3 by simp
    moreover
```

```
    have flat v3 = flat v using as1 Right(3)
     by (simp add: Posix1(2))
    ultimately have Right v :⊑val Right v3
      by (auto simp add: PosOrd_ex_eq_def PosOrd_RightI)
    then show Right v :⊑val v2 unfolding Right .
  next
    case (Left v3)
    have v3 ∈ LV r1 s using Left(2,3) as2
     by (auto simp add: LV_def prefix_list_def)
    then have flat v3 = flat v ∧ ⊨ v3 : r1 using as1 Left(3)
     by (simp add: Posix1(2) LV_def)
    then have False using as1 as2 Left
     by (auto simp add: Posix1(2) L_flat_Prf1)
    then show Right v :⊑val v2 by simp
  qed
next
  case (Posix_SEQ s1 r1 v1 s2 r2 v2 v3)
  have s1 ∈ r1 → v1 s2 ∈ r2 → v2 by fact+
  then have as1: s1 = flat v1 s2 = flat v2 by (simp_all add: Posix1(2))
  have IH1: ⋀v3. v3 ∈ LV r1 s1 ⟹ v1 :⊑val v3 by fact
  have IH2: ⋀v3. v3 ∈ LV r2 s2 ⟹ v2 :⊑val v3 by fact
  have cond: ¬ (∃ s_3 s_4. s_3 ≠ [] ∧ s_3 @ s_4 = s2 ∧ s1 @ s_3 ∈ L r1 ∧ s_4 ∈ L r2) by fact
  have v3 ∈ LV (SEQ r1 r2) (s1 @ s2) by fact
  then obtain v3a v3b where eqs:
   v3 = Seq v3a v3b ⊨ v3a : r1 ⊨ v3b : r2
   flat v3a @ flat v3b = s1 @ s2
   by (force simp add: prefix_list_def LV_def elim: Prf.cases)
  with cond have flat v3a ⊑pre s1 unfolding prefix_list_def
   by (smt L_flat_Prf1 append_eq_append_conv2 append_self_conv)
  then have flat v3a ⊏spre s1 ∨ (flat v3a = s1 ∧ flat v3b = s2) using eqs
   by (simp add: sprefix_list_def append_eq_conv_conj)
  then have q2: v1 :⊏val v3a ∨ (flat v3a = s1 ∧ flat v3b = s2)
   using PosOrd_spreI as1(1) eqs by blast
  then have v1 :⊏val v3a ∨ (v3a ∈ LV r1 s1 ∧ v3b ∈ LV r2 s2) using eqs(2,3)
   by (auto simp add: LV_def)
  then have v1 :⊏val v3a ∨ (v1 :⊑val v3a ∧ v2 :⊑val v3b) using IH1 IH2 by blast
  then have Seq v1 v2 :⊑val Seq v3a v3b using eqs q2 as1
   unfolding  PosOrd_ex_eq_def by (auto simp add: PosOrd_SeqI1 PosOrd_Seq_eq)
  then show Seq v1 v2 :⊑val v3 unfolding eqs by blast
next
  case (Posix_STAR1 s1 r v s2 vs v3)
  have s1 ∈ r → v s2 ∈ STAR r → Stars vs by fact+
  then have as1: s1 = flat v s2 = flat (Stars vs) by (auto dest: Posix1(2))
  have IH1: ⋀v3. v3 ∈ LV r s1 ⟹ v :⊑val v3 by fact
  have IH2: ⋀v3. v3 ∈ LV (STAR r) s2 ⟹ Stars vs :⊑val v3 by fact
```

```
 have cond: ¬ (∃s3 s4. s3 ≠ [] ∧ s3 @ s4 = s2 ∧ s1 @ s3 ∈ L r ∧ s4 ∈ L (STAR r))
by fact
 have cond2: flat v ≠ [] by fact
 have v3 ∈ LV (STAR r) (s1 @ s2) by fact
 then consider
  (NonEmpty) v3a vs3 where v3 = Stars (v3a # vs3)
  ⊨ v3a : r ⊨ Stars vs3 : STAR r
  flat (Stars (v3a # vs3)) = s1 @ s2
 | (Empty) v3 = Stars []
 unfolding LV_def
 apply(auto)
 apply(erule Prf.cases)
 apply(auto)
 apply(case_tac vs)
 apply(auto intro: Prf.intros)
 done
 then show Stars (v # vs) :⊑val v3
  proof (cases)
   case (NonEmpty v3a vs3)
   have flat (Stars (v3a # vs3)) = s1 @ s2 using NonEmpty(4) .
   with cond have flat v3a ⊑pre s1 using NonEmpty(2,3)
    unfolding prefix_list_def
    by (smt L_flat_Prf1 append_Nil2 append_eq_append_conv2 flat.simps(7))
     then have flat v3a ⊏spre s1 ∨ (flat v3a = s1 ∧ flat (Stars vs3) = s2) using
NonEmpty(4)
    by (simp add: sprefix_list_def append_eq_conv_conj)
   then have q2: v :⊏val v3a ∨ (flat v3a = s1 ∧ flat (Stars vs3) = s2)
    using PosOrd_spreI as1(1) NonEmpty(4) by blast
   then have v :⊏val v3a ∨ (v3a ∈ LV r s1 ∧ Stars vs3 ∈ LV (STAR r) s2)
    using NonEmpty(2,3) by (auto simp add: LV_def)
   then have v :⊏val v3a ∨ (v :⊑val v3a ∧ Stars vs :⊑val Stars vs3) using IH1 IH2
by blast
   then have v :⊏val v3a ∨ (v = v3a ∧ Stars vs :⊑val Stars vs3)
     unfolding PosOrd_ex_eq_def by auto
   then have Stars (v # vs) :⊑val Stars (v3a # vs3) using NonEmpty(4) q2 as1
    unfolding  PosOrd_ex_eq_def
    using PosOrd_StarsI PosOrd_StarsI2 by auto
   then show Stars (v # vs) :⊑val v3 unfolding NonEmpty by blast
  next
   case Empty
   have v3 = Stars [] by fact
   then show Stars (v # vs) :⊑val v3
   unfolding PosOrd_ex_eq_def using cond2
   by (simp add: PosOrd_shorterI)
  qed
```

```
next
  case (Posix_STAR2 r v2)
  have v2 ∈ LV (STAR r) [] by fact
  then have v2 = Stars []
    unfolding LV_def by (auto elim: Prf.cases)
  then show Stars [] :⊑val v2
  by (simp add: PosOrd_ex_eq_def)
qed


lemma Posix_PosOrd_reverse:
  assumes s ∈ r → v1
  shows ¬(∃ v2 ∈ LV r s. v2 :⊏val v1)
using assms
by (metis Posix_PosOrd less_irrefl PosOrd_def
    PosOrd_ex_eq_def PosOrd_ex_def PosOrd_trans)

lemma PosOrd_Posix:
  assumes v1 ∈ LV r s ∀ v₂ ∈ LV r s. ¬ v₂ :⊏val v1
  shows s ∈ r → v1
proof –
  have s ∈ L r using assms(1) unfolding LV_def
    using L_flat_Prf1 by blast
  then obtain vposix where vp: s ∈ r → vposix
    using lexer_correct_Some by blast
  with assms(1) have vposix :⊑val v1 by (simp add: Posix_PosOrd)
  then have vposix = v1 ∨ vposix :⊏val v1 unfolding PosOrd_ex_eq2 by auto
  moreover
    { assume vposix :⊏val v1
      moreover
      have vposix ∈ LV r s using vp
         using Posix_LV by blast
      ultimately have False using assms(2) by blast
    }
  ultimately show s ∈ r → v1 using vp by blast
qed

lemma Least_existence:
  assumes LV r s ≠ {}
  shows ∃ vmin ∈ LV r s. ∀ v ∈ LV r s. vmin :⊑val v
proof –
  from assms
  obtain vposix where s ∈ r → vposix
  unfolding LV_def
  using L_flat_Prf1 lexer_correct_Some by blast
```

```
  then have ∀ v ∈ LV r s. vposix :⊑val v
    by (simp add: Posix_PosOrd)
  then show ∃ vmin ∈ LV r s. ∀ v ∈ LV r s. vmin :⊑val v
    using Posix_LV ‹s ∈ r → vposix› by blast
qed

lemma Least_existence1:
  assumes LV r s ≠ {}
  shows ∃!vmin ∈ LV r s. ∀ v ∈ LV r s. vmin :⊑val v
using Least_existence[OF assms] assms
using PosOrdeq_antisym by blast

lemma Least_existence2:
  assumes LV r s ≠ {}
  shows ∃!vmin ∈ LV r s. lexer r s = Some vmin ∧ (∀ v ∈ LV r s. vmin :⊑val v)
using Least_existence[OF assms] assms
using PosOrdeq_antisym
  using PosOrd_Posix PosOrd_ex_eq2 lexer_correctness(1) by auto


lemma Least_existence1_pre:
  assumes LV r s ≠ {}
  shows ∃!vmin ∈ LV r s. ∀ v ∈ (LV r s ∪ {v'. flat v' ⊏spre s}). vmin :⊑val v
using Least_existence[OF assms] assms
apply −
apply(erule bexE)
apply(rule_tac a=vmin in ex1I)
apply(auto)[1]
apply (metis PosOrd_Posix PosOrd_ex_eq2 PosOrd_spreI PosOrdeq_antisym Posix1(2))
apply(auto)[1]
apply(simp add: PosOrdeq_antisym)
done

lemma
  shows partial_order_on UNIV {(v1, v2). v1 :⊑val v2}
apply(simp add: partial_order_on_def)
apply(simp add: preorder_on_def refl_on_def)
apply(simp add: PosOrdeq_refl)
apply(auto)
apply(rule transI)
apply(auto intro: PosOrdeq_trans)[1]
apply(rule antisymI)
apply(simp add: PosOrdeq_antisym)
done
```

lemma
$wf\ \{(v1, v2).\ v1 :\sqsubset val\ v2 \wedge v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s\}$
apply(*rule finite_acyclic_wf*)
prefer *2*
apply(*simp add*: *acyclic_def*)
apply(*induct_tac rule*: *trancl.induct*)
apply(*auto*)[*1*]
oops

unused-thms

end

theory *Sulzmann*
imports *Lexer*
begin

21 Bit-Encodings

datatype $bit = Z \mid S$

fun
$code :: val \Rightarrow bit\ list$
where
$code\ Void = []$
$\mid code\ (Char\ c) = []$
$\mid code\ (Left\ v) = Z\ \#\ (code\ v)$
$\mid code\ (Right\ v) = S\ \#\ (code\ v)$
$\mid code\ (Seq\ v1\ v2) = (code\ v1)\ @\ (code\ v2)$
$\mid code\ (Stars\ []) = [S]$
$\mid code\ (Stars\ (v\ \#\ vs)) = (Z\ \#\ code\ v)\ @\ code\ (Stars\ vs)$

fun
$Stars\_add :: val \Rightarrow val \Rightarrow val$
where
$Stars\_add\ v\ (Stars\ vs) = Stars\ (v\ \#\ vs)$

function
$decode' :: bit\ list \Rightarrow rexp \Rightarrow (val * bit\ list)$
where
$decode'\ ds\ ZERO = (Void, [])$
$\mid decode'\ ds\ ONE = (Void, ds)$
$\mid decode'\ ds\ (CHAR\ d) = (Char\ d, ds)$

```
| 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 ≝ (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)


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

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 (AALT cs r1 r2) = AALT (bs @ cs) r1 r2
| fuse bs (ASEQ cs r1 r2) = ASEQ (bs @ cs) r1 r2
| fuse bs (ASTAR cs r) = ASTAR (bs @ cs) r

fun intern :: rexp ⇒ arexp where
  intern ZERO = AZERO
| intern ONE = AONE []
| intern (CHAR 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 (AALT bs r1 r2) (Left v) = bs @ retrieve r1 v
| retrieve (AALT bs r1 r2) (Right v) = bs @ retrieve r2 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
  erase :: arexp ⇒ rexp
where
  erase AZERO = ZERO
| erase (AONE _) = ONE
| erase (ACHAR _ c) = CHAR c
| erase (AALT _ r1 r2) = ALT (erase r1) (erase r2)
| erase (ASEQ _ r1 r2) = SEQ (erase r1) (erase r2)
| erase (ASTAR _ r) = STAR (erase r)
```

```
fun
 bnullable :: arexp ⇒ bool
where
  bnullable (AZERO) = False
| bnullable (AONE bs) = True
| bnullable (ACHAR bs c) = False
| bnullable (AALT bs r1 r2) = (bnullable r1 ∨ bnullable r2)
| 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(AALT bs r1 r2) = (if bnullable(r1) then bs @ (bmkeps r1) else bs @ (bmkeps
r2))
| 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 (AALT bs r1 r2) = AALT bs (bder c r1) (bder c r2)
| 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)
  apply(simp_all)
  done

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

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)
  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)
  using retrieve_encode_STARS
  apply(auto elim!: Prf_elims)
  apply(case_tac vs)
  apply(simp)
  apply(simp)
  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 bmkeps_retrieve:
  assumes nullable (erase r)
  shows bmkeps r = retrieve r (mkeps (erase r))
  using assms
  apply(induct r)
  apply(auto simp add: bnullable_correctness)
  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)
 apply(auto elim!: Prf_elims simp add: retrieve_fuse2 bnullable_correctness bmkeps_retrieve)
 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 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



end
```

22 Introduction

Brzozowski [4] 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 [14]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [9]. And another one in Coq is given by Coquand and Siles [5].

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 [6] and the other is POSIX matching [1,10,12,16,17]. 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 [1] 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 [16] 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, [8]). The value for matching *xy* in a single 'iteration', i.e. the POSIX value, would look as follows

$$Stars\ [Seq\ (Char\ x)\ (Char\ y)]$$

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 [16] 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 [6] 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 [16]); 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 [10] who found that nearly all POSIX matching implementations are "buggy" [16, Page 203] and by Grathwohl et al [7, 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 [16]. We have proved the correctness of this algorithm according to our specification of what a POSIX value is (inspired by work of Vansummeren [17]). Sulzmann and Lu sketch in [16] 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 [12].

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

23 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

---

[4] An extended version of [16] 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.

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:

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

In clause *(4)* we use the operation $\_ \mathbin{@} \_$ 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 $s_1 \mathbin{@} s_2$ 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

$$\textit{Der}\ c\ A \overset{\textit{def}}{=} \{s \mid c :: s \in A\}\ .$$

For semantic derivatives we have the following equations (for example mechanically proved in [9]):

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

Brzozowski's derivatives of regular expressions [4] 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) $L(r\backslash c) = \mathit{Der}\ c\ (L(r))$.

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

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

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 [16] is to append another phase to this algorithm in order to calculate a [lexical] value. We will explain the details next.

24 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 [16] 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

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

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 [6], and Sulzmann and Lu for POSIX matching [16]). The string underlying a value can be calculated by the *flat* function, written $|\_|$ and defined as:

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

We will sometimes refer to the underlying string of a value as *flattened value*. We will also overload our notation and use $|\mathit{vs}|$ 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]

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

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

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

where in the clause for *Stars* we use the notation $v \in \mathit{vs}$ 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]

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

---

[5] Note that the rule for *Stars* differs from our earlier paper [3]. There we used the original definition by Sulzmann and Lu which does not require that the values $v \in \mathit{vs}$ 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.

[6] Okui and Suzuki refer to our lexical values as *canonical values* in [12]. The notion of *non-problematic values* by Cardelli and Frisch [6] is related, but not identical to our lexical values.

$$\begin{tikzcd}
r_1 \arrow[r, "\_\backslash a"] & r_2 \arrow[r, "\_\backslash b"] & r_3 \arrow[r, "\_\backslash c"] & r_4\ \mathit{nullable} \arrow[d, "mkeps"] \\
v_1 & v_2 \arrow[l, "inj\ r_1\ a"'] & v_3 \arrow[l, "inj\ r_2\ b"'] & v_4 \arrow[l, "inj\ r_3\ c"']
\end{tikzcd}$$

Fig. 1. The two phases of the algorithm by Sulzmann & Lu [16], 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.

The main property of *LV r s* is that it is alway finite.

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

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, $LV\ (\mathbf{1}^\star)\ []$ would contain infinitely many values, but according to our more restricted definition only a single value, namely $LV\ (\mathbf{1}^\star)\ [] = \{Stars\ []\}$.

If a regular expression r matches a string s, then generally the set *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:

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

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 [16] 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).

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

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:

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

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

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

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

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 [16], 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 [17]. 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* $v = v'$.

---

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

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

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

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

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

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

Fig. 2. Our inductive definition of POSIX values.

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

We claim that our $(s,\ r) \to v$ 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 $s_1 \mathbin{@} s_2$. According to the Longest Match Rule, we want that the s_1 is the longest initial split of $s_1 \mathbin{@} s_2$ 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 $s_1 \mathbin{@} s_3$ 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 $s_1 \mathbin{@} s_2$ and therefore $\mathit{Seq}\ v_1\ v_2$ cannot be a POSIX value for $(s_1 \mathbin{@} s_2,\ r_1 \cdot r_2)$. 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 $s_1 \mathbin{@} s_2$ 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 $\mathit{Stars}\ [\,]$ 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 2. *If* $(s,\ r) \to v$ *then* $v \in \mathit{LV}\ r\ s$.

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

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

Lemma 3. *If* $\mathit{nullable}\ r$ *then* $([], r) \to \mathit{mkeps}\ r$.

Proof. By routine induction on r. □

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

Lemma 4. *If* $(s, r\backslash c) \to v$ *then* $(c :: s, r) \to \mathit{inj}\ r\ c\ v$.

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

- Case $r = r_1 + r_2$. There are two subcases, namely (a) $v = \mathit{Left}\ v'$ and $(s, r_1\backslash c) \to v'$; and (b) $v = \mathit{Right}\ v'$, $s \notin L(r_1\backslash c)$ and $(s, r_2\backslash c) \to v'$. In (a) we know $(s, r_1\backslash c) \to v'$, from which we can infer $(c :: s, r_1) \to \mathit{inj}\ r_1\ c\ v'$ by induction hypothesis and hence $(c :: s, r_1 + r_2) \to \mathit{inj}\ (r_1 + r_2)\ c\ (\mathit{Left}\ v')$ 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) $v = \mathit{Left}\ (\mathit{Seq}\ v_1\ v_2)$ and $\mathit{nullable}\ r_1$
  (b) $v = \mathit{Right}\ v_1$ and $\mathit{nullable}\ r_1$
  (c) $v = \mathit{Seq}\ v_1\ v_2$ and $\neg\ \mathit{nullable}\ r_1$

  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 $(c :: s_1, r_1) \to \mathit{inj}\ r_1\ c\ v_1$. Putting this all together allows us to infer $(c :: s_1 @ s_2, r_1 \cdot r_2) \to \mathit{Seq}\ (\mathit{inj}\ r_1\ c\ v_1)\ v_2$. 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 $(c :: s, r_2) \to \mathit{inj}\ r_2\ c\ v_1$ 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 3 we know $([], r_1) \to \mathit{mkeps}\ r_1$ holds. Putting this all together, we can conclude with $(c :: s, r_1 \cdot r_2) \to \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{inj}\ r_2\ c\ v_1)$, 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 $|\mathit{inj}\ r_1\ c\ v_1| \neq []$. This follows from $(c :: s_1, r_1) \to \mathit{inj}\ r_1\ c\ v_1$ (which in turn follows from $(s_1, r_1\backslash c) \to v_1$ and the induction hypothesis). □

With Lemma 4 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* $\mathit{lexer}\ r\ s = \mathit{None}$
(2) $s \in L(r)$ *if and only if* $\exists v.\ \mathit{lexer}\ r\ s = \mathit{Some}\ v \wedge (s, r) \to v$

Proof. By induction on s using Lemma 3 and 4. □

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) $\mathit{lexer}\ r\ s = \mathit{None}$ *if and only if* $\nexists v.a.\ (s, r) \to v$
(2) $\mathit{lexer}\ r\ s = \mathit{Some}\ v$ *if and only if* $(s, r) \to v$

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.

25 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 [16]: they introduced an ordering for values and identified POSIX values as the maximal elements. An extended version of [16] 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 [12,13] 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

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

---

[8] All deviations we introduced are harmless.

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

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

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 *Pos v*, is given by

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

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:

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

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

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

Both values match x. At position $[0]$ the norm of v is 1 (the subvalue matches x), but the norm of w is -1 (the position is outside w according to how we defined the 'inside' positions of *Left*- and *Right*-values). Of course at position $[1]$, the norms $\|v\|_{[1]}$ and $\|w\|_{[1]}$ are reversed, but the point is that subvalues will be analysed according to lexicographically ordered positions. According to this ordering, the position $[0]$ takes precedence over $[1]$ 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 [12]: 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

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

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 [12]. 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:

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

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 5 (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 -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 [12] 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 $(s, r_1 + r_2) \rightarrow Left\ w_1$: In this case the value v_2 is either of the form $Left\ w_2$ or $Right\ w_2$. 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 $w_2 \in LV\, r_1\, s$ 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 $Left\ w_1 \preccurlyeq Left\ w_2$ using Proposition 5*(2)*.

- Case $P{+}R$ with $(s, r_1 + r_2) \to Right\ w_1$: 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 $Left\ w_2$. 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 $(s_1 @ s_2, r_1 \cdot r_2) \to Seq\ w_1\ w_2$: We can assume $v_2 = Seq\ u_1\ u_2$ with $u_1 : r_1$ and $u_2 : r_2$. We have $s_1 @ s_2 = |u_1| @ |u_2|$. 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 $u_1 \in LV\, r_1\, s_1$ and $u_2 \in LV\, r_2\, s_2$. 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 $LV\ r\ s$. By Proposition 4*(2)* we also know that any value in $LV\ r\ s'$, with s' being a strict prefix, cannot be smaller than v_1. The next theorem shows the opposite—namely any minimal element in $LV\ r\ s$ 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* $v_1 \in LV\ r\ s$ *and* $\forall\, v_2 \in LV\ r\ s.\ v_2 \not\prec v_1$ *then* $(s, r) \to v_1$.

Proof. If $v_1 \in LV\ r\ s$ then $s \in L(r)$ by Proposition 2. Hence by Theorem 2(2) there exists a *POSIX* value v_P with $(s, r) \to v_P$ and by Lemma 2 we also have $v_P \in LV\ r\ s$. 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 $LV\ r\ s$. So we are done in this case too. □

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

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

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 24. This provides an independent confirmation that our ternary relation formalise the informal POSIX rules.

26 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

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

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

$$\begin{array}{rcl}
\mathit{areg} & ::= & \mathit{AZERO} \\
& \mid & \mathit{AONE}\ \mathit{bs} \\
& \mid & \mathit{ACHAR}\ \mathit{bs}\ c \\
& \mid & \mathit{AALT}\ \mathit{bs}\ r_1\ r_2 \\
& \mid & \mathit{ASEQ}\ \mathit{bs}\ r_1\ r_2 \\
& \mid & \mathit{ASTAR}\ \mathit{bs}\ r
\end{array}$$

$$\begin{array}{lcl}
(\mathbf{0})^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{AZERO} \\
(\mathbf{1})^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{AONE}\ [] \\
(c)^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{ACHAR}\ []\ c \\
(r_1 + r_2)^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{AALT}\ []\ (\mathit{fuse}\ [Z]\ (r_1{}^{\uparrow}))\ (\mathit{fuse}\ [S]\ (r_2{}^{\uparrow})) \\
(r_1 \cdot r_2)^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{ASEQ}\ []\ (r_1{}^{\uparrow})\ (r_2{}^{\uparrow}) \\
(r^{\star})^{\uparrow} & \stackrel{\text{def}}{=} & \mathit{ASTAR}\ []\ (r^{\uparrow})
\end{array}$$

$$\begin{array}{lcl}
\mathit{AZERO}^{\downarrow} & \stackrel{\text{def}}{=} & \mathbf{0} \\
(\mathit{AONE}\ \mathit{bs})^{\downarrow} & \stackrel{\text{def}}{=} & \mathbf{1} \\
(\mathit{ACHAR}\ \mathit{bs}\ c)^{\downarrow} & \stackrel{\text{def}}{=} & c \\
(\mathit{AALT}\ \mathit{bs}\ r_1\ r_2)^{\downarrow} & \stackrel{\text{def}}{=} & (r_1{}^{\downarrow}) + (r_2{}^{\downarrow}) \\
(\mathit{ASEQ}\ \mathit{bs}\ r_1\ r_2)^{\downarrow} & \stackrel{\text{def}}{=} & (r_1{}^{\downarrow}) \cdot (r_2{}^{\downarrow}) \\
(\mathit{ASTAR}\ \mathit{bs}\ r)^{\downarrow} & \stackrel{\text{def}}{=} & (r^{\downarrow})^{\star}
\end{array}$$

Some simple facts about erase

Lemma 6.
$(r\backslash\!\backslash a)^{\downarrow} = (r^{\downarrow})\backslash a$
$(r^{\uparrow})^{\downarrow} = r$

$$\begin{array}{lcl}
\mathit{nullable}_b\ \mathit{AZERO} & \stackrel{\text{def}}{=} & \mathit{False}\\
\mathit{nullable}_b\ (\mathit{AONE}\ \mathit{bs}) & \stackrel{\text{def}}{=} & \mathit{True}\\
\mathit{nullable}_b\ (\mathit{ACHAR}\ \mathit{bs}\ c) & \stackrel{\text{def}}{=} & \mathit{False}\\
\mathit{nullable}_b\ (\mathit{AALT}\ \mathit{bs}\ r_1\ r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}_b\ r_1 \vee \mathit{nullable}_b\ r_2\\
\mathit{nullable}_b\ (\mathit{ASEQ}\ \mathit{bs}\ r_1\ r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}_b\ r_1 \wedge \mathit{nullable}_b\ r_2\\
\mathit{nullable}_b\ (\mathit{ASTAR}\ \mathit{bs}\ r) & \stackrel{\text{def}}{=} & \mathit{True}
\end{array}$$

$$\begin{array}{lcl}
\mathit{AZERO}\backslash\!\backslash c & \stackrel{\text{def}}{=} & \mathit{AZERO}\\
\mathit{AONE}\ \mathit{bs}\backslash\!\backslash c & \stackrel{\text{def}}{=} & \mathit{AZERO}\\
\mathit{ACHAR}\ \mathit{bs}\ d\backslash\!\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathit{AONE}\ \mathit{bs}\ \mathit{else}\ \mathit{AZERO}\\
\mathit{AALT}\ r_1\ r_2\ \mathit{r2.0}\backslash\!\backslash \mathit{bs} & \stackrel{\text{def}}{=} & \mathit{AALT}\ r_1\ (r_2\backslash\!\backslash \mathit{bs})\ (\mathit{r2.0}\backslash\!\backslash \mathit{bs})\\
\mathit{ASEQ}\ r_1\ r_2\ \mathit{r2.0}\backslash\!\backslash \mathit{bs} & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}_b\ r_2\ \mathit{then}\ \mathit{AALT}\ r_1\ (\mathit{ASEQ}\ []\ (r_2\backslash\!\backslash \mathit{bs})\ \mathit{r2.0})\ (\mathit{fuse}\ (\mathit{mkeps}_b\ r_2)\ (\mathit{r2.0}\backslash\!\backslash \mathit{bs}))\ \mathit{else}\ A\\
\mathit{ASTAR}\ \mathit{bs}\ r\backslash\!\backslash c & \stackrel{\text{def}}{=} & \mathit{ASEQ}\ \mathit{bs}\ (\mathit{fuse}\ [Z]\ (r\backslash\!\backslash c))\ (\mathit{ASTAR}\ []\ r)
\end{array}$$

$$\begin{array}{lcl}
\mathit{mkeps}_b\ (\mathit{AONE}\ \mathit{bs}) & \stackrel{\text{def}}{=} & \mathit{bs}\\
\mathit{mkeps}_b\ (\mathit{ASEQ}\ \mathit{bs}\ r_1\ r_2) & \stackrel{\text{def}}{=} & \mathit{bs}\ @\ \mathit{mkeps}_b\ r_1\ @\ \mathit{mkeps}_b\ r_2\\
\mathit{mkeps}_b\ (\mathit{AALT}\ \mathit{bs}\ r_1\ r_2) & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}_b\ r_1\ \mathit{then}\ \mathit{bs}\ @\ \mathit{mkeps}_b\ r_1\ \mathit{else}\ \mathit{bs}\ @\ \mathit{mkeps}_b\ r_2\\
\mathit{mkeps}_b\ (\mathit{ASTAR}\ \mathit{bs}\ r) & \stackrel{\text{def}}{=} & \mathit{bs}\ @\ [S]
\end{array}$$

> *If* $v : (r^{\downarrow})\backslash c$ *then* $\mathit{retrieve}\ (r\backslash\!\backslash c)\ v = \mathit{retrieve}\ r\ (\mathit{inj}\ (r^{\downarrow})\ c\ v)$.
> By induction on r

Theorem 5 (Main Lemma).
If $v : r\backslash s$ *then* $\mathit{Some}\ (\mathit{flex}\ r\ \mathit{id}\ s\ v) = \mathit{decode}\ (\mathit{retrieve}\ (r^{\uparrow}\backslash\!\backslash s)\ v)\ r$.

Definition of the bitcoded lexer

> $\mathit{lexer}_b\ r\ s \stackrel{\mathit{def}}{=} \mathit{if}\ \mathit{nullable}_b\ (r^{\uparrow}\backslash\!\backslash s)\ \mathit{then}\ \mathit{decode}\ (\mathit{mkeps}_b\ (r^{\uparrow}\backslash\!\backslash s))\ r\ \mathit{else}\ \mathit{None}$

Theorem 6. $\mathit{lexer}_b\ r\ s = \mathit{lexer}\ r\ s$

27 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 [16]. 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 [16] 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:

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

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

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

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:

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

In the second clause we first calculate the derivative $r\backslash c$ and then 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 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 $(s, fst\ (simp\ r)) \to v$ *then* $(s, r) \to snd\ (simp\ r)\ v$.

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 $\mathbf{0}$ (respectively $\mathbf{1}$). For example for $r = r_1 + r_2$, consider the subcase $fst\ (simp\ r_1) = \mathbf{0}$ and $fst\ (simp\ r_2) \neq \mathbf{0}$. By assumption we know $(s, fst\ (simp\ (r_1 + r_2))) \to v$. From this we can infer $(s, fst\ (simp\ r_2)) \to v$ and by IH also (*) $(s, r_2) \to snd\ (simp\ r_2)\ v$. Given $fst\ (simp\ r_1) = \mathbf{0}$ we know $L(fst\ (simp\ r_1)) = \varnothing$. 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 $(s, r_1 + r_2) \to Right\ (snd\ (simp\ r_2)\ v)$. In turn this gives $(s, r_1 + r_2) \to snd\ (simp\ (r_1 + r_2))\ v$ 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 7. $lexer^+\ r\ s = lexer\ r\ s$

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 $lexer^+\ r\ s = lexer\ r\ s$ 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 $fst\ (simp\ (r\backslash c))$, and f_r be the rectification function, that is $snd\ (simp\ (r\backslash c))$. 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 $lexer\ (r\backslash c)\ s = Some\ v$ 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 $lexer\ r_s\ s = Some\ v'$ 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 $lexer\ (r\backslash c)\ s = None$ by Theorem 2(1). We also know by Lemma 7(1) that $s \notin L(r_s)$. Hence $lexer\ r_s\ s = None$ by Theorem 2(1) and by IH then also $lexer^+\ r_s\ s = None$. With this we can conclude in this case too. □

28 Conclusion

We have implemented the POSIX value calculation algorithm introduced by Sulzmann and Lu [16]. Our implementation is nearly identical to the original and all modifications we introduced are harmless (like our char-clause for *inj*). We have proved this algorithm to be correct, but correct according to our own specification of what POSIX values are. Our specification (inspired from work by Vansummeren [17]) appears to be much simpler than in [16] and our proofs are nearly always straightforward. We have attempted to formalise the original proof by Sulzmann and Lu [16], but we believe it contains unfillable gaps. In the online version of [16], the authors already acknowledge some small problems, but our experience suggests that there are more serious problems.

Having proved the correctness of the POSIX lexing algorithm in [16], which lessons have we learned? Well, this is a perfect example for the importance of the *right* definitions. We have (on and off) explored mechanisations as soon as first versions of [16] appeared, but have made little progress with turning the relatively detailed proof sketch in [16] into a formalisable proof. Having seen [17] and adapted the POSIX definition given there for the algorithm by Sulzmann and Lu made all the difference: the proofs, as said, are nearly straightforward. The question remains whether the original proof idea of [16], potentially using our result as a stepping stone, can be made to work? Alas, we really do not know despite considerable effort.

Closely related to our work is an automata-based lexer formalised by Nipkow [11]. This lexer also splits up strings into longest initial substrings, but Nipkow's algorithm is not completely computational. The algorithm by Sulzmann and Lu, in contrast, can be implemented with ease in any functional language. A bespoke lexer for the Imp-language is formalised in Coq as part of the Software Foundations book by Pierce et al [15]. The disadvantage of such bespoke lexers is that they do not generalise easily to more advanced features. Our formalisation is available from the Archive of Formal Proofs [2] under http://www.isa-afp.org/entries/Posix-Lexing.shtml.

Acknowledgements: We are very grateful to Martin Sulzmann for his comments on our work and moreover for patiently explaining to us the details in [16]. We also received very helpful comments from James Cheney and anonymous referees.

References

1. The Open Group Base Specification Issue 6 IEEE Std 1003.1 2004 Edition, 2004. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html.
2. F. Ausaf, R. Dyckhoff, and C. Urban. POSIX Lexing with Derivatives of Regular Expressions. *Archive of Formal Proofs*, 2016. http://www.isa-afp.org/entries/Posix-Lexing.shtml, Formal proof development.
3. F. Ausaf, R. Dyckhoff, and C. Urban. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Proc. of the 7th International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 69–86, 2016.
4. J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.
5. T. Coquand and V. Siles. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Proc. of the 1st International Conference on Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 119–134, 2011.

6. A. Frisch and L. Cardelli. Greedy Regular Expression Matching. In *Proc. of the 31st International Conference on Automata, Languages and Programming (ICALP)*, volume 3142 of *LNCS*, pages 618–629, 2004.
7. N. B. B. Grathwohl, F. Henglein, and U. T. Rasmussen. A Crash-Course in Regular Expression Parsing and Regular Expressions as Types. Technical report, University of Copenhagen, 2014.
8. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, 2005.
9. A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *Journal of Automated Reasoning*, 49:95–106, 2012.
10. C. Kuklewicz. Regex Posix. https://wiki.haskell.org/Regex_Posix.
11. T. Nipkow. Verified Lexical Analysis. In *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1479 of *LNCS*, pages 1–15, 1998.
12. S. Okui and T. Suzuki. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. In *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA)*, volume 6482 of *LNCS*, pages 231–240, 2010.
13. S. Okui and T. Suzuki. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. Technical report, University of Aizu, 2013.
14. S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.
15. B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2015. http://www.cis.upenn.edu/~bcpierce/sf.
16. M. Sulzmann and K. Lu. POSIX Regular Expression Parsing with Derivatives. In *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*, volume 8475 of *LNCS*, pages 203–220, 2014.
17. S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.