POSIX Lexing with Derivatives of Regular Expressions

Fahad Ausaf Roy Dyckhoff Christian Urban

March 17, 2025

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 [2] cleverly extended this algorithm in order to deal with POSIX matching, which is the underlying disambiguation strategy for regular expressions needed in lexers. In this entry 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 also prove the correctness of an optimised version of the POSIX matching algorithm. Finally we show that (iii) our inductive definition of a POSIX value is equivalent to an alternative definition by Okui and Suzuki [1] which identifies POSIX values as least elements according to an ordering of values. All results are given also for the bounded regular expressions ($r^{\{n\}}$ and $r^{\{..n\}}$).

Contents

theory *Lexer*
imports *Regular−Sets.Derivatives*

begin

1 Values

datatype *'a val* =
  Void
| *Atm 'a*
| *Seq 'a val 'a val*
| *Right 'a val*
| *Left 'a val*
| *Stars ('a val) list*

2 The string behind a value

fun
  flat :: *'a val* ⇒ *'a list*
where
  flat (*Void*) = []
| *flat* (*Atm 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*) = *concat* (*map flat vs*)
by (*induct vs*) (*auto*)

3 Relation between values and regular expressions

inductive
  Prf :: *'a val* ⇒ *'a rexp* ⇒ *bool* (‹⊢ - : -› [*100*, *100*] *100*)
where
 ⟦⊢ *v1* : *r1*; ⊢ *v2* : *r2*⟧ ⟹ ⊢ *Seq v1 v2* : *Times r1 r2*
| ⊢ *v1* : *r1* ⟹ ⊢ *Left v1* : *Plus r1 r2*
| ⊢ *v2* : *r2* ⟹ ⊢ *Right v2* : *Plus r1 r2*
| ⊢ *Void* : *One*
| ⊢ *Atm c* : *Atom c*
| ⟦∀ *v* ∈ *set vs*. ⊢ *v* : *r* ∧ *flat v* ≠ []⟧ ⟹ ⊢ *Stars vs* : *Star r*

inductive-cases *Prf-elims*:
  ⊢ *v* : *Zero*
  ⊢ *v* : *Times r1 r2*
  ⊢ *v* : *Plus r1 r2*

$\vdash v : One$
$\vdash v : Atom\ c$
$\vdash vs : Star\ r$

lemma *Prf-flat-lang*:
assumes $\vdash v : r$ **shows** $flat\ v \in lang\ r$
using *assms*
by (*induct v r rule*: *Prf.induct*)
(*auto simp add*: *concat-in-star subset-eq*)

lemma *Star-string*:
assumes $s \in star\ A$
shows $\exists ss.\ concat\ ss = s \wedge (\forall s \in set\ ss.\ s \in A)$
using *assms*
by (*metis in-star-iff-concat subsetD*)

lemma *Star-val*:
assumes $\forall s{\in}set\ ss.\ \exists v.\ s = flat\ v \wedge \vdash v : r$
shows $\exists vs.\ flats\ vs = concat\ ss \wedge (\forall v{\in}set\ vs.\ \vdash v : r \wedge flat\ v \neq [])$
using *assms*
apply(*induct ss*)
apply(*auto*)
apply (*metis empty-iff list.set*(*1*))
by (*metis append.simps*(*1*) *flat.simps*(*7*) *flat-Stars set-ConsD*)

lemma *L-flat-Prf1*:
assumes $\vdash v : r$ **shows** $flat\ v \in lang\ r$
using *assms*
apply (*induct*)
apply(*auto*)
by (*metis Prf.intros*(*6*) *Prf-flat-lang flat-Stars lang.simps*(*6*))

lemma *L-flat-Prf2*:
assumes $s \in lang\ r$ **shows** $\exists v.\ \vdash v : r \wedge flat\ v = s$
using *assms*
apply(*induct r arbitrary*: *s*)
apply(*auto intro*: *Prf.intros*)
using *Prf.intros*(*2*) *flat.simps*(*3*) **apply** *blast*
using *Prf.intros*(*3*) *flat.simps*(*4*) **apply** *blast*
apply (*metis Prf.intros*(*1*) *concE flat.simps*(*5*))
apply(*subgoal-tac* $\exists vs{::}('a\ val)\ list.\ concat\ (map\ flat\ vs) = s \wedge (\forall v \in set\ vs.\ \vdash v : r \wedge flat\ v \neq [])$)
apply(*auto*)[*1*]
apply(*rule-tac x=Stars vs* **in** *exI*)
apply(*simp*)
apply(*drule Star-string*)
apply(*auto*)

using *Prf.intros(6)* **apply** *blast*
by (*smt* (*verit*) *Star-val in-star-iff-concat subset-iff*)

lemma *L-flat-Prf*:
  lang r = {*flat v* | *v*. ⊢ *v* : *r*}
using *L-flat-Prf1 L-flat-Prf2* **by** *blast*

4 Sulzmann and Lu functions

fun
  mkeps :: *'a rexp* ⇒ *'a val*
where
  mkeps(*One*) = *Void*
| *mkeps*(*Times r1 r2*) = *Seq* (*mkeps r1*) (*mkeps r2*)
| *mkeps*(*Plus r1 r2*) = (*if nullable*(*r1*) *then Left* (*mkeps r1*) *else Right* (*mkeps r2*))
| *mkeps*(*Star r*) = *Stars* []

fun *injval* :: *'a rexp* ⇒ *'a* ⇒ *'a val* ⇒ *'a val*
where
  injval (*Atom d*) *c Void* = *Atm c*
| *injval* (*Plus r1 r2*) *c* (*Left v1*) = *Left*(*injval r1 c v1*)
| *injval* (*Plus r1 r2*) *c* (*Right v2*) = *Right*(*injval r2 c v2*)
| *injval* (*Times r1 r2*) *c* (*Seq v1 v2*) = *Seq* (*injval r1 c v1*) *v2*
| *injval* (*Times r1 r2*) *c* (*Left* (*Seq v1 v2*)) = *Seq* (*injval r1 c v1*) *v2*
| *injval* (*Times 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*)

5 Mkeps, injval

lemma *mkeps-nullable*:
  assumes *nullable r*
  shows ⊢ *mkeps r* : *r*
using *assms*
by (*induct r*)
  (*auto intro*: *Prf.intros*)

lemma *mkeps-flat*:
  assumes *nullable r*
  shows *flat* (*mkeps r*) = []
using *assms*
by (*induct r*) (*auto*)

lemma *Prf-injval-flat*:
  assumes ⊢ *v* : *deriv c r*
  shows *flat* (*injval r c v*) = *c* # (*flat v*)
using *assms*
apply(*induct c r arbitrary*: *v rule*: *deriv.induct*)
apply(*auto elim*!: *Prf-elims intro*: *mkeps-flat split*: *if-splits*)

done

lemma *Prf-injval*:
assumes $\vdash v : \mathit{deriv}\ c\ r$
shows $\vdash (\mathit{injval}\ r\ c\ v) : r$
using *assms*
apply(*induct r arbitrary*: *c v rule*: *rexp.induct*)
apply(*auto intro*!: *Prf.intros mkeps-nullable elim*!: *Prf-elims split*: *if-splits*)

by (*simp add*: *Prf-injval-flat*)

6 Our Alternative Posix definition

inductive
$\mathit{Posix} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{rexp} \Rightarrow {}'a\ \mathit{val} \Rightarrow \mathit{bool}\ (\langle\text{-} \in \text{-} \rightarrow \text{-}\rangle\ [100,\ 100,\ 100]\ 100)$
where
$\mathit{Posix\text{-}One}$: $[] \in \mathit{One} \rightarrow \mathit{Void}$
| $\mathit{Posix\text{-}Atom}$: $[c] \in (\mathit{Atom}\ c) \rightarrow (\mathit{Atm}\ c)$
| $\mathit{Posix\text{-}Plus1}$: $s \in r1 \rightarrow v \Longrightarrow s \in (\mathit{Plus}\ r1\ r2) \rightarrow (\mathit{Left}\ v)$
| $\mathit{Posix\text{-}Plus2}$: $[\![s \in r2 \rightarrow v;\ s \notin \mathit{lang}\ r1]\!] \Longrightarrow s \in (\mathit{Plus}\ r1\ r2) \rightarrow (\mathit{Right}\ v)$
| $\mathit{Posix\text{-}Times}$: $[\![s1 \in r1 \rightarrow v1;\ s2 \in r2 \rightarrow v2;$
$\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (s1 @ s_3) \in \mathit{lang}\ r1 \wedge s_4 \in \mathit{lang}\ r2)]\!] \Longrightarrow$
$(s1 @ s2) \in (\mathit{Times}\ r1\ r2) \rightarrow (\mathit{Seq}\ v1\ v2)$
| $\mathit{Posix\text{-}Star1}$: $[\![s1 \in r \rightarrow v;\ s2 \in \mathit{Star}\ r \rightarrow \mathit{Stars}\ vs;\ \mathit{flat}\ v \neq [];$
$\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (s1 @ s_3) \in \mathit{lang}\ r \wedge s_4 \in \mathit{lang}\ (\mathit{Star}\ r))]\!]$
$\Longrightarrow (s1 @ s2) \in \mathit{Star}\ r \rightarrow \mathit{Stars}\ (v \# vs)$
| $\mathit{Posix\text{-}Star2}$: $[] \in \mathit{Star}\ r \rightarrow \mathit{Stars}\ []$

inductive-cases *Posix-elims*:
$s \in \mathit{Zero} \rightarrow v$
$s \in \mathit{One} \rightarrow v$
$s \in \mathit{Atom}\ c \rightarrow v$
$s \in \mathit{Plus}\ r1\ r2 \rightarrow v$
$s \in \mathit{Times}\ r1\ r2 \rightarrow v$
$s \in \mathit{Star}\ r \rightarrow v$

lemma *Posix1*:
assumes $s \in r \rightarrow v$
shows $s \in \mathit{lang}\ r\ \mathit{flat}\ v = s$
using *assms*
by (*induct s r v rule*: *Posix.induct*) (*auto*)

lemma *Posix1a*:
assumes $s \in r \rightarrow v$
shows $\vdash v : r$
using *assms*
apply(*induct s r v rule*: *Posix.induct*)
apply(*auto intro*: *Prf.intros*)

by (*metis Prf.intros(6) Prf-elims(6) set-ConsD val.inject(5)*)

lemma *Posix-mkeps*:
  assumes *nullable r*
  shows $[] \in r \rightarrow mkeps\ r$
using *assms*
apply(*induct r*)
apply(*auto intro*: *Posix.intros simp add*: *nullable-iff*)
apply(*subst append.simps(1)[symmetric]*)
apply(*rule Posix.intros*)
apply(*auto*)
done

lemma *Posix-determ*:
  assumes $s \in r \rightarrow v1$ $s \in r \rightarrow v2$
  shows $v1 = v2$
using *assms*
proof (*induct s r v1 arbitrary*: *v2 rule*: *Posix.induct*)
  case (*Posix-One v2*)
  have $[] \in One \rightarrow v2$ **by** *fact*
  then show $Void = v2$ **by** *cases auto*
next
  case (*Posix-Atom c v2*)
  have $[c] \in Atom\ c \rightarrow v2$ **by** *fact*
  then show $Atm\ c = v2$ **by** *cases auto*
next
  case (*Posix-Plus1 s r1 v r2 v2*)
  have $s \in Plus\ r1\ r2 \rightarrow v2$ **by** *fact*
  moreover
  have $s \in r1 \rightarrow v$ **by** *fact*
  then have $s \in lang\ r1$ **by** (*simp add*: *Posix1*)
  ultimately obtain v' **where** *eq*: $v2 = Left\ v'$ $s \in r1 \rightarrow v'$ **by** *cases auto*
  moreover
  have *IH*: $\bigwedge v2.\ s \in r1 \rightarrow v2 \Longrightarrow v = v2$ **by** *fact*
  ultimately have $v = v'$ **by** *simp*
  then show $Left\ v = v2$ **using** *eq* **by** *simp*
next
  case (*Posix-Plus2 s r2 v r1 v2*)
  have $s \in Plus\ r1\ r2 \rightarrow v2$ **by** *fact*
  moreover
  have $s \notin lang\ r1$ **by** *fact*
  ultimately obtain v' **where** *eq*: $v2 = Right\ v'$ $s \in r2 \rightarrow v'$
    by *cases* (*auto simp add*: *Posix1*)
  moreover
  have *IH*: $\bigwedge v2.\ s \in r2 \rightarrow v2 \Longrightarrow v = v2$ **by** *fact*
  ultimately have $v = v'$ **by** *simp*
  then show $Right\ v = v2$ **using** *eq* **by** *simp*

next
  case $(Posix\text{-}Times\ s1\ r1\ v1\ s2\ r2\ v2\ v')$
  have $(s1\ @\ s2) \in Times\ r1\ r2 \rightarrow v'$
    $s1 \in r1 \rightarrow v1\ s2 \in r2 \rightarrow v2$
    $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge s1\ @\ s_3 \in lang\ r1 \wedge s_4 \in lang\ r2)$ **by** *fact+*
  then obtain $v1'\ v2'$ **where** $v' = Seq\ v1'\ v2'\ s1 \in r1 \rightarrow v1'\ s2 \in r2 \rightarrow v2'$
  apply(*cases*) **apply** (*auto simp add*: *append-eq-append-conv2*)
  using *Posix1*(*1*) **by** *fastforce+*
  moreover
  have *IHs*: $\bigwedge v1'.\ s1 \in r1 \rightarrow v1' \Longrightarrow v1 = v1'$
    $\bigwedge v2'.\ s2 \in r2 \rightarrow v2' \Longrightarrow v2 = v2'$ **by** *fact+*
  ultimately show $Seq\ v1\ v2 = v'$ **by** *simp*
next
  case $(Posix\text{-}Star1\ s1\ r\ v\ s2\ vs\ v2)$
  have $(s1\ @\ s2) \in Star\ r \rightarrow v2$
    $s1 \in r \rightarrow v\ s2 \in Star\ r \rightarrow Stars\ vs\ flat\ v \neq []$
    $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge s1\ @\ s_3 \in lang\ r \wedge s_4 \in lang\ (Star\ r))$ **by** *fact+*
  then obtain $v'\ vs'$ **where** $v2 = Stars\ (v'\ \#\ vs')\ s1 \in r \rightarrow v'\ s2 \in (Star\ r) \rightarrow (Stars\ vs')$
  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*: $\bigwedge v2.\ s1 \in r \rightarrow v2 \Longrightarrow v = v2$
    $\bigwedge v2.\ s2 \in Star\ r \rightarrow v2 \Longrightarrow Stars\ vs = v2$ **by** *fact+*
  ultimately show $Stars\ (v\ \#\ vs) = v2$ **by** *auto*
next
  case $(Posix\text{-}Star2\ r\ v2)$
  have $[] \in Star\ r \rightarrow v2$ **by** *fact*
  then show $Stars\ [] = v2$ **by** *cases* (*auto simp add*: *Posix1*)
qed

lemma *Posix-injval*:
  assumes $s \in (deriv\ c\ r) \rightarrow v$
  shows $(c\ \#\ s) \in r \rightarrow (injval\ r\ c\ v)$
using *assms*
proof(*induct r arbitrary*: *s v rule*: *rexp.induct*)
  case *Zero*
  have $s \in deriv\ c\ Zero \rightarrow v$ **by** *fact*
  then have $s \in Zero \rightarrow v$ **by** *simp*
  then have *False* **by** *cases*
  then show $(c\ \#\ s) \in Zero \rightarrow (injval\ Zero\ c\ v)$ **by** *simp*
next
  case *One*
  have $s \in deriv\ c\ One \rightarrow v$ **by** *fact*

then have $s \in \textit{Zero} \rightarrow v$ **by** *simp*
then have *False* **by** *cases*
then show $(c \ \# \ s) \in \textit{One} \rightarrow (\textit{injval One } c \ v)$ **by** *simp*
next
case (*Atom d*)
consider $(\textit{eq}) \ c = d \mid (\textit{ineq}) \ c \neq d$ **by** *blast*
then show $(c \ \# \ s) \in (\textit{Atom } d) \rightarrow (\textit{injval } (\textit{Atom } d) \ c \ v)$
proof (*cases*)
case *eq*
have $s \in \textit{deriv } c \ (\textit{Atom } d) \rightarrow v$ **by** *fact*
then have $s \in \textit{One} \rightarrow v$ **using** *eq* **by** *simp*
then have *eqs*: $s = [] \wedge v = \textit{Void}$ **by** *cases simp*
show $(c \ \# \ s) \in \textit{Atom } d \rightarrow \textit{injval } (\textit{Atom } d) \ c \ v$ **using** *eq eqs*
by (*auto intro*: *Posix.intros*)
next
case *ineq*
have $s \in \textit{deriv } c \ (\textit{Atom } d) \rightarrow v$ **by** *fact*
then have $s \in \textit{Zero} \rightarrow v$ **using** *ineq* **by** *simp*
then have *False* **by** *cases*
then show $(c \ \# \ s) \in \textit{Atom } d \rightarrow \textit{injval } (\textit{Atom } d) \ c \ v$ **by** *simp*
qed
next
case (*Plus r1 r2*)
have *IH1*: $\bigwedge s \ v. \ s \in \textit{deriv } c \ \textit{r1} \rightarrow v \Longrightarrow (c \ \# \ s) \in \textit{r1} \rightarrow \textit{injval r1 } c \ v$ **by** *fact*
have *IH2*: $\bigwedge s \ v. \ s \in \textit{deriv } c \ \textit{r2} \rightarrow v \Longrightarrow (c \ \# \ s) \in \textit{r2} \rightarrow \textit{injval r2 } c \ v$ **by** *fact*
have $s \in \textit{deriv } c \ (\textit{Plus r1 r2}) \rightarrow v$ **by** *fact*
then have $s \in \textit{Plus } (\textit{deriv } c \ \textit{r1}) \ (\textit{deriv } c \ \textit{r2}) \rightarrow v$ **by** *simp*
then consider (*left*) v' **where** $v = \textit{Left } v' \ s \in \textit{deriv } c \ \textit{r1} \rightarrow v'$
| (*right*) v' **where** $v = \textit{Right } v' \ s \notin \textit{lang } (\textit{deriv } c \ \textit{r1}) \ s \in \textit{deriv } c \ \textit{r2} \rightarrow$
v'
by *cases auto*
then show $(c \ \# \ s) \in \textit{Plus r1 r2} \rightarrow \textit{injval } (\textit{Plus r1 r2}) \ c \ v$
proof (*cases*)
case *left*
have $s \in \textit{deriv } c \ \textit{r1} \rightarrow v'$ **by** *fact*
then have $(c \ \# \ s) \in \textit{r1} \rightarrow \textit{injval r1 } c \ v'$ **using** *IH1* **by** *simp*
then have $(c \ \# \ s) \in \textit{Plus r1 r2} \rightarrow \textit{injval } (\textit{Plus r1 r2}) \ c \ (\textit{Left } v')$ **by** (*auto*
intro: *Posix.intros*)
then show $(c \ \# \ s) \in \textit{Plus r1 r2} \rightarrow \textit{injval } (\textit{Plus r1 r2}) \ c \ v$ **using** *left* **by** *simp*
next
case *right*
have $s \notin \textit{lang } (\textit{deriv } c \ \textit{r1})$ **by** *fact*
then have $c \ \# \ s \notin \textit{lang r1}$ **by** (*simp add*: *lang-deriv Deriv-def*)
moreover
have $s \in \textit{deriv } c \ \textit{r2} \rightarrow v'$ **by** *fact*
then have $(c \ \# \ s) \in \textit{r2} \rightarrow \textit{injval r2 } c \ v'$ **using** *IH2* **by** *simp*
ultimately have $(c \ \# \ s) \in \textit{Plus r1 r2} \rightarrow \textit{injval } (\textit{Plus r1 r2}) \ c \ (\textit{Right } v')$
by (*auto intro*: *Posix.intros*)
then show $(c \ \# \ s) \in \textit{Plus r1 r2} \rightarrow \textit{injval } (\textit{Plus r1 r2}) \ c \ v$ **using** *right* **by**

simp
qed
next
case (*Times r1 r2*)
have *IH1*: $\bigwedge s\ v.\ s \in deriv\ c\ r1 \rightarrow v \Longrightarrow (c\ \#\ s) \in r1 \rightarrow injval\ r1\ c\ v$ **by** *fact*
have *IH2*: $\bigwedge s\ v.\ s \in deriv\ c\ r2 \rightarrow v \Longrightarrow (c\ \#\ s) \in r2 \rightarrow injval\ r2\ c\ v$ **by** *fact*
have $s \in deriv\ c\ (Times\ r1\ r2) \rightarrow v$ **by** *fact*
then consider
(*left-nullable*) *v1 v2 s1 s2* **where**
$v = Left\ (Seq\ v1\ v2)\quad s = s1\ @\ s2$
$s1 \in deriv\ c\ r1 \rightarrow v1\ s2 \in r2 \rightarrow v2\ nullable\ r1$
$\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ s1\ @\ s_3 \in lang\ (deriv\ c\ r1)\ \wedge\ s_4 \in lang\ r2)$
| (*right-nullable*) *v1 s1 s2* **where**
$v = Right\ v1\ s = s1\ @\ s2$
$s \in deriv\ c\ r2 \rightarrow v1\ nullable\ r1\ s1\ @\ s2 \notin lang\ (Times\ (deriv\ c\ r1)\ r2)$
| (*not-nullable*) *v1 v2 s1 s2* **where**
$v = Seq\ v1\ v2\ s = s1\ @\ s2$
$s1 \in deriv\ c\ r1 \rightarrow v1\ s2 \in r2 \rightarrow v2\ \neg nullable\ r1$
$\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ s1\ @\ s_3 \in lang\ (deriv\ c\ r1)\ \wedge\ s_4 \in lang\ r2)$
by (*force split*: *if-splits elim*!: *Posix-elims simp add*: *lang-deriv Deriv-def*)
then show $(c\ \#\ s) \in Times\ r1\ r2 \rightarrow injval\ (Times\ r1\ r2)\ c\ v$
proof (*cases*)
case *left-nullable*
have $s1 \in deriv\ c\ r1 \rightarrow v1$ **by** *fact*
then have $(c\ \#\ s1) \in r1 \rightarrow injval\ r1\ c\ v1$ **using** *IH1* **by** *simp*
moreover
have $\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ s1\ @\ s_3 \in lang\ (deriv\ c\ r1)\ \wedge\ s_4 \in lang\ r2)$ **by** *fact*
then have $\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ (c\ \#\ s1)\ @\ s_3 \in lang\ r1\ \wedge\ s_4 \in lang\ r2)$
by (*simp add*: *lang-deriv Deriv-def*)
ultimately have $((c\ \#\ s1)\ @\ s2) \in Times\ r1\ r2 \rightarrow Seq\ (injval\ r1\ c\ v1)\ v2$ **using** *left-nullable* **by** (*rule-tac Posix.intros*)
then show $(c\ \#\ s) \in Times\ r1\ r2 \rightarrow injval\ (Times\ r1\ r2)\ c\ v$ **using** *left-nullable* **by** *simp*
next
case *right-nullable*
have *nullable r1* **by** *fact*
then have $[] \in r1 \rightarrow (mkeps\ r1)$ **by** (*rule Posix-mkeps*)
moreover
have $s \in deriv\ c\ r2 \rightarrow v1$ **by** *fact*
then have $(c\ \#\ s) \in r2 \rightarrow (injval\ r2\ c\ v1)$ **using** *IH2* **by** *simp*
moreover
have $s1\ @\ s2 \notin lang\ (Times\ (deriv\ c\ r1)\ r2)$ **by** *fact*
then have $\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = c\ \#\ s\ \wedge\ []\ @\ s_3 \in lang\ r1\ \wedge\ s_4 \in lang\ r2)$
using *right-nullable*

apply (*auto simp add*: *lang-deriv Deriv-def append-eq-Cons-conv*)
by (*metis concI mem-Collect-eq*)
ultimately have ($[] @ (c \# s)) \in Times\ r1\ r2 \to Seq\ (mkeps\ r1)\ (injval\ r2\ c\ v1)$
by(*rule Posix.intros*)
then show $(c \# s) \in Times\ r1\ r2 \to injval\ (Times\ r1\ r2)\ c\ v$ **using** *right-nullable* **by** *simp*
next
case *not-nullable*
have $s1 \in deriv\ c\ r1 \to v1$ **by** *fact*
then have $(c \# s1) \in r1 \to injval\ r1\ c\ v1$ **using** *IH1* **by** *simp*
moreover
have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r1) \wedge s_4 \in lang\ r2)$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r1 \wedge s_4 \in lang\ r2)$ **by** (*simp add*: *lang-deriv Deriv-def*)
ultimately have $((c \# s1) @ s2) \in Times\ r1\ r2 \to Seq\ (injval\ r1\ c\ v1)\ v2$ **using** *not-nullable*
by (*rule-tac Posix.intros*) (*simp-all*)
then show $(c \# s) \in Times\ r1\ r2 \to injval\ (Times\ r1\ r2)\ c\ v$ **using** *not-nullable* **by** *simp*
qed
next
case (*Star r*)
have *IH*: $\bigwedge s\ v.\ s \in deriv\ c\ r \to v \Longrightarrow (c \# s) \in r \to injval\ r\ c\ v$ **by** *fact*
have $s \in deriv\ c\ (Star\ r) \to v$ **by** *fact*
then consider
(*cons*) *v1 vs s1 s2* **where**
$v = Seq\ v1\ (Stars\ vs)\ s = s1 @ s2$
$s1 \in deriv\ c\ r \to v1\ s2 \in (Star\ r) \to (Stars\ vs)$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Star\ r))$
apply(*auto elim*!: *Posix-elims*(*1−5*) *simp add*: *lang-deriv Deriv-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 $(c \# s) \in Star\ r \to injval\ (Star\ r)\ c\ v$
proof (*cases*)
case *cons*
have $s1 \in deriv\ c\ r \to v1$ **by** *fact*
then have $(c \# s1) \in r \to injval\ r\ c\ v1$ **using** *IH* **by** *simp*
moreover
have $s2 \in Star\ r \to Stars\ vs$ **by** *fact*
moreover
have $(c \# s1) \in r \to injval\ r\ c\ v1$ **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 $\neg$ ($\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Star\ r)$) **by** *fact*
then have $\neg$ ($\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r \wedge s_4 \in lang\ (Star\ r)$)
by (*simp add*: *lang-deriv Deriv-def*)
ultimately
have $((c \# s1) @ s2) \in Star\ r \to Stars\ (injval\ r\ c\ v1 \# vs)$ **by** (*rule Posix.intros*)
then show $(c \# s) \in Star\ r \to injval\ (Star\ r)\ c\ v$ **using** *cons* **by**(*simp*)
qed
qed

7 The Lexer by Sulzmann and Lu

fun
lexer :: *'a rexp* ⇒ *'a list* ⇒ (*'a val*) *option*
where
lexer r [] = (*if nullable r then Some*(*mkeps r*) *else None*)
| *lexer r* (*c#s*) = (*case* (*lexer* (*deriv c r*) *s*) *of*
None ⇒ *None*
| *Some*(*v*) ⇒ *Some*(*injval r c v*))

lemma *lexer-correct-None*:
shows $s \notin lang\ r \longleftrightarrow lexer\ r\ s = None$
apply(*induct s arbitrary*: *r*)
apply(*simp add*: *nullable-iff*)
apply(*drule-tac x=deriv a r* **in** *meta-spec*)
apply(*auto simp add*: *lang-deriv Deriv-def*)
done

lemma *lexer-correct-Some*:
shows $s \in lang\ r \longleftrightarrow (\exists v.\ lexer\ r\ s = Some(v) \wedge s \in r \to v)$
apply(*induct s arbitrary*: *r*)
apply(*auto simp add*: *Posix-mkeps nullable-iff*)[*1*]
apply(*drule-tac x=deriv a r* **in** *meta-spec*)
apply(*simp add*: *lang-deriv Deriv-def*)
apply(*rule iffI*)
apply(*auto intro*: *Posix-injval simp add*: *Posix1*(*1*))
done

lemma *lexer-correctness*:
shows $(lexer\ r\ s = Some\ v) \longleftrightarrow s \in r \to v$
and $(lexer\ r\ s = None) \longleftrightarrow \neg(\exists v.\ s \in r \to v)$
apply(*auto*)
using *lexer-correct-None lexer-correct-Some* **apply** *fastforce*
using *Posix1*(*1*) *Posix-determ lexer-correct-Some* **apply** *blast*
using *Posix1*(*1*) *lexer-correct-None* **apply** *blast*

using *lexer-correct-None lexer-correct-Some* **by** *blast*

end
theory *LexicalVals*
imports *Lexer HOL−Library.Sublist*
begin

8 Sets of Lexical Values

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

definition
$LV :: 'a\ rexp \Rightarrow 'a\ list \Rightarrow ('a\ val)\ set$
where $LV\ r\ s \equiv \{v.\ \vdash v : r \wedge flat\ v = s\}$

lemma *LV-simps*:
shows $LV\ Zero\ s = \{\}$
and $LV\ One\ s = (if\ s = []\ then\ \{Void\}\ else\ \{\})$
and $LV\ (Atom\ c)\ s = (if\ s = [c]\ then\ \{Atm\ c\}\ else\ \{\})$
and $LV\ (Plus\ r1\ r2)\ s = Left\ `\ LV\ r1\ s \cup Right\ `\ LV\ r2\ s$
unfolding *LV-def*
by (*auto intro*: *Prf.intros elim*: *Prf.cases*)

abbreviation
$Prefixes\ s \equiv \{s'.\ prefix\ s'\ s\}$

abbreviation
$Suffixes\ s \equiv \{s'.\ suffix\ s'\ s\}$

abbreviation
$SSuffixes\ s \equiv \{s'.\ strict\text{-}suffix\ s'\ s\}$

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 $SSuffixes\ s \subseteq Suffixes\ s$
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::'a 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::'a val, 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 (Atom c s)
  show finite (LV (Atom c) s) by (simp add: LV-simps)
next
  case (Plus r1 r2 s)
  then show finite (LV (Plus r1 r2) s) by (simp add: LV-simps)
next
  case (Times r1 r2 s)
  define f where f ≡ λ(v1::'a val, 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 (Times 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 (Times 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
```

Our POSIX values are lexical values.

```
lemma Posix-LV:
```

assumes $s \in r \to v$
shows $v \in LV\ r\ s$
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 $s \in r \to v$
shows $\vdash v : r$
using *assms Posix-LV LV-def*
by *blast*

end

theory *Simplifying*
imports *Lexer*
begin

9 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-Plus* **where**
F-Plus f_1 f_2 (*Right v*) = *Right* (f_2 *v*)
| *F-Plus* f_1 f_2 (*Left v*) = *Left* (f_1 *v*)
| *F-Plus f1 f2 v* = *v*

fun *F-Times1* **where**
F-Times1 f_1 f_2 *v* = *Seq* (f_1 *Void*) (f_2 *v*)

fun *F-Times2* **where**
F-Times2 f_1 f_2 *v* = *Seq* (f_1 *v*) (f_2 *Void*)

fun *F-Times* **where**
F-Times f_1 f_2 (*Seq* v_1 v_2) = *Seq* (f_1 v_1) (f_2 v_2)
| *F-Times f1 f2 v* = *v*

fun *simp-Plus* **where**
simp-Plus (*Zero*, f_1) (r_2, f_2) = (r_2, *F-RIGHT* f_2)
| *simp-Plus* (r_1, f_1) (*Zero*, f_2) = (r_1, *F-LEFT* f_1)
| *simp-Plus* (r_1, f_1) (r_2, f_2) =

(*if* $r_1 = r_2$ *then* $(r_1,$ *F-LEFT* $f_1)$ *else* (*Plus* r_1 r_2, *F-Plus* f_1 f_2))

fun *simp-Times* **where**
  simp-Times (*Zero*, f_1) (r_2, f_2) = (*Zero*, *undefined*)
| *simp-Times* (r_1, f_1) (*Zero*, f_2) = (*Zero*, *undefined*)
| *simp-Times* (*One*, f_1) (r_2, f_2) = (r_2, *F-Times1* f_1 f_2)
| *simp-Times* (r_1, f_1) (*One*, f_2) = (r_1, *F-Times2* f_1 f_2)
| *simp-Times* (r_1, f_1) (r_2, f_2) = (*Times* r_1 r_2, *F-Times* f_1 f_2)

lemma *simp-Times-simps*[*simp*]:
  simp-Times p1 p2 = (*if* (*fst p1* = *Zero*) *then* (*Zero*, *undefined*)
    else (*if* (*fst p2* = *Zero*) *then* (*Zero*, *undefined*)
    else (*if* (*fst p1* = *One*) *then* (*fst p2*, *F-Times1* (*snd p1*) (*snd p2*))
    else (*if* (*fst p2* = *One*) *then* (*fst p1*, *F-Times2* (*snd p1*) (*snd p2*))
    else (*Times* (*fst p1*) (*fst p2*), *F-Times* (*snd p1*) (*snd p2*))))))
 by (*induct p1 p2 rule*: *simp-Times.induct*)(*auto*)

lemma *simp-Plus-simps*[*simp*]:
  simp-Plus 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 (*if* (*fst p1* = *fst p2*) *then* (*fst p1*, *F-LEFT* (*snd p1*))
    else (*Plus* (*fst p1*) (*fst p2*), *F-Plus* (*snd p1*) (*snd p2*)))))
by (*induct p1 p2 rule*: *simp-Plus.induct*) (*auto*)

fun
  simp :: *'a rexp* ⇒ *'a rexp* ∗ (*'a val* ⇒ *'a val*)
where
  simp (*Plus r1 r2*) = *simp-Plus* (*simp r1*) (*simp r2*)
| *simp* (*Times r1 r2*) = *simp-Times* (*simp r1*) (*simp r2*)
| *simp r* = (*r*, *id*)

fun
  slexer :: *'a rexp* ⇒ *'a list* ⇒ (*'a val*) *option*
where
  slexer r [] = (*if nullable r then Some*(*mkeps r*) *else None*)
| *slexer r* (*c#s*) = (*let* (*rs*, *fr*) = *simp* (*deriv 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* (*deriv c r*))) *s*) *of*
      None ⇒ *None*
    | *Some*(*v*) ⇒ *Some*(*injval r c* ((*snd* (*simp* (*deriv c r*))) *v*)))
by (*auto split*: *prod.split option.split*)

lemma *L-fst-simp*:
  shows *lang r* = *lang* (*fst* (*simp r*))

by $(induct\ r)$ $(auto)$

lemma *Posix-simp*:
  assumes $s \in (fst\ (simp\ r)) \to v$
  shows $s \in r \to ((snd\ (simp\ r))\ v)$
using *assms*
proof$(induct\ r\ arbitrary{:}\ s\ v\ rule{:}\ rexp.induct)$
  case $(Plus\ r1\ r2\ s\ v)$
  have *IH1*: $\bigwedge s\ v.\ s \in fst\ (simp\ r1) \to v \Longrightarrow s \in r1 \to snd\ (simp\ r1)\ v$ **by** *fact*
  have *IH2*: $\bigwedge s\ v.\ s \in fst\ (simp\ r2) \to v \Longrightarrow s \in r2 \to snd\ (simp\ r2)\ v$ **by** *fact*
  have *as*: $s \in fst\ (simp\ (Plus\ r1\ r2)) \to v$ **by** *fact*
  consider $(Zero\text{-}Zero)\ fst\ (simp\ r1) = Zero\ fst\ (simp\ r2) = Zero$
    $|\ (Zero\text{-}NZero)\ fst\ (simp\ r1) = Zero\ fst\ (simp\ r2) \neq Zero$
    $|\ (NZero\text{-}Zero)\ fst\ (simp\ r1) \neq Zero\ fst\ (simp\ r2) = Zero$
    $|\ (NZero\text{-}NZero1)\ fst\ (simp\ r1) \neq Zero\ fst\ (simp\ r2) \neq Zero\ fst\ (simp\ r1) = fst\ (simp\ r2)$
    $|\ (NZero\text{-}NZero2)\ fst\ (simp\ r1) \neq Zero\ fst\ (simp\ r2) \neq Zero\ fst\ (simp\ r1) \neq fst\ (simp\ r2)$ **by** *auto*
  then show $s \in Plus\ r1\ r2 \to snd\ (simp\ (Plus\ r1\ r2))\ v$
    proof$(cases)$
      case $(Zero\text{-}Zero)$
      with *as* **have** $s \in Zero \to v$ **by** *simp*
    then show $s \in Plus\ r1\ r2 \to snd\ (simp\ (Plus\ r1\ r2))\ v$ **by** $(rule\ Posix\text{-}elims(1))$
    next
      case $(Zero\text{-}NZero)$
      with *as* **have** $s \in fst\ (simp\ r2) \to v$ **by** *simp*
      with *IH2* **have** $s \in r2 \to snd\ (simp\ r2)\ v$ **by** *simp*
      moreover
      from *Zero-NZero* **have** $fst\ (simp\ r1) = Zero$ **by** *simp*
      then have $lang\ (fst\ (simp\ r1)) = \{\}$ **by** *simp*
      then have $lang\ r1 = \{\}$ **using** *L-fst-simp* **by** *auto*
      then have $s \notin lang\ r1$ **by** *simp*
      ultimately have $s \in Plus\ r1\ r2 \to Right\ (snd\ (simp\ r2)\ v)$ **by** $(rule\ Posix\text{-}Plus2)$
      then show $s \in Plus\ r1\ r2 \to snd\ (simp\ (Plus\ r1\ r2))\ v$
      using *Zero-NZero* **by** *simp*
    next
      case $(NZero\text{-}Zero)$
      with *as* **have** $s \in fst\ (simp\ r1) \to v$ **by** *simp*
      with *IH1* **have** $s \in r1 \to snd\ (simp\ r1)\ v$ **by** *simp*
      then have $s \in Plus\ r1\ r2 \to Left\ (snd\ (simp\ r1)\ v)$ **by** $(rule\ Posix\text{-}Plus1)$
      then show $s \in Plus\ r1\ r2 \to snd\ (simp\ (Plus\ r1\ r2))\ v$ **using** *NZero-Zero* **by** *simp*
    next
      case $(NZero\text{-}NZero1)$
      with *as* **have** *a*: $s \in fst\ (simp\ r1) \to v$ **by** *simp*
        then show $s \in Plus\ r1\ r2 \to snd\ (simp\ (Plus\ r1\ r2))\ v$
        using *IH1 NZero-NZero1 Posix-Plus1 a* **by** *fastforce*
    next

case (*NZero-NZero2*)
with *as* **have** $s \in$ *Plus* (*fst* (*simp r1*)) (*fst* (*simp r2*)) $\to$ *v* **by** *simp*
then consider (*Left*) *v1* **where** v = *Left v1* $s \in$ (*fst* (*simp r1*)) $\to$ *v1*
| (*Right*) *v2* **where** v = *Right v2* $s \in$ (*fst* (*simp r2*)) $\to$ *v2* $s \notin$ *lang* (*fst* (*simp r1*))
by (*erule-tac Posix-elims*(*4*))
then show $s \in$ *Plus r1 r2* $\to$ *snd* (*simp* (*Plus r1 r2*)) *v*
proof(*cases*)
case (*Left*)
then have v = *Left v1* $s \in$ *r1* $\to$ (*snd* (*simp r1*) *v1*) **using** *IH1* **by** *simp-all*
then show $s \in$ *Plus r1 r2* $\to$ *snd* (*simp* (*Plus r1 r2*)) *v* **using** *NZero-NZero2*
by (*simp-all add*: *Posix-Plus1*)
next
case (*Right*)
then have v = *Right v2* $s \in$ *r2* $\to$ (*snd* (*simp r2*) *v2*) $s \notin$ *lang r1* **using** *IH2 L-fst-simp* **by** *auto*
then show $s \in$ *Plus r1 r2* $\to$ *snd* (*simp* (*Plus r1 r2*)) *v* **using** *NZero-NZero2*
by (*simp-all add*: *Posix-Plus2*)
qed
qed
next
case (*Times r1 r2 s v*)
have *IH1*: $\bigwedge s\ v.\ s \in$ *fst* (*simp r1*) $\to v \Longrightarrow s \in$ *r1* $\to$ *snd* (*simp r1*) *v* **by** *fact*
have *IH2*: $\bigwedge s\ v.\ s \in$ *fst* (*simp r2*) $\to v \Longrightarrow s \in$ *r2* $\to$ *snd* (*simp r2*) *v* **by** *fact*
have *as*: $s \in$ *fst* (*simp* (*Times r1 r2*)) $\to$ *v* **by** *fact*
consider (*Zero*) *fst* (*simp r1*) = *Zero* $\vee$ *fst* (*simp r2*) = *Zero*
| (*One-One*) *fst* (*simp r1*) = *One fst* (*simp r2*) = *One*
| (*One-NOne*) *fst* (*simp r1*) = *One fst* (*simp r2*) $\neq$ *One fst* (*simp r2*) $\neq$ *Zero*
| (*NOne-One*) *fst* (*simp r1*) $\neq$ *One fst* (*simp r2*) = *One fst* (*simp r1*) $\neq$ *Zero*
| (*NOne-NOne*) *fst* (*simp r1*) $\neq$ *One fst* (*simp r2*) $\neq$ *One*
fst (*simp r1*) $\neq$ *Zero fst* (*simp r2*) $\neq$ *Zero* **by** *auto*
then show $s \in$ *Times r1 r2* $\to$ *snd* (*simp* (*Times r1 r2*)) *v*
proof(*cases*)
case (*Zero*)
with *as* **have** *False*
by (*metis Posix-elims*(*1*) *fst-conv simp.simps*(*2*) *simp-Times-simps*)
then show $s \in$ *Times r1 r2* $\to$ *snd* (*simp* (*Times r1 r2*)) *v* **by** *simp*
next
case (*One-One*)
with *as* **have** *b*: $s \in$ *One* $\to$ *v* **by** *simp*
from *b* **have** $s \in$ *r1* $\to$ *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 [] $\in$ *One* $\to$ *Void* **by** (*simp add*: *Posix-One*)
then have [] $\in$ *fst* (*simp r2*) $\to$ *Void* **using** *One-One* **by** *simp*
then have [] $\in$ *r2* $\to$ *snd* (*simp r2*) *Void* **using** *IH2* **by** *simp*

ultimately have $([] @ []) \in$ *Times r1 r2* $\rightarrow$ *Seq* (*snd* (*simp r1*) *Void*) (*snd* (*simp r2*) *Void*)
using *Posix-Times* **by** *blast*
then show $s \in$ *Times r1 r2* $\rightarrow$ *snd* (*simp* (*Times r1 r2*)) *v* **using** *c One-One* **by** *simp*
next
case (*One-NOne*)
with *as* **have** *b*: $s \in$ *fst* (*simp r2*) $\rightarrow$ *v* **by** *simp*
from *b* **have** $s \in$ *r2* $\rightarrow$ *snd* (*simp r2*) *v* **using** *IH2 One-NOne* **by** *simp*
moreover
have $[] \in$ *One* $\rightarrow$ *Void* **by** (*simp add*: *Posix-One*)
then have $[] \in$ *fst* (*simp r1*) $\rightarrow$ *Void* **using** *One-NOne* **by** *simp*
then have $[] \in$ *r1* $\rightarrow$ *snd* (*simp r1*) *Void* **using** *IH1* **by** *simp*
moreover
from *One-NOne*(*1*) **have** *lang* (*fst* (*simp r1*)) = {[]} **by** *simp*
then have *lang r1* = {[]} **by** (*simp add*: *L-fst-simp*[*symmetric*])
ultimately have $([] @ s) \in$ *Times r1 r2* $\rightarrow$ *Seq* (*snd* (*simp r1*) *Void*) (*snd* (*simp r2*) *v*)
by(*rule-tac Posix-Times*) *auto*
then show $s \in$ *Times r1 r2* $\rightarrow$ *snd* (*simp* (*Times r1 r2*)) *v* **using** *One-NOne* **by** *simp*
next
case (*NOne-One*)
with *as* **have** $s \in$ *fst* (*simp r1*) $\rightarrow$ *v* **by** *simp*
with *IH1* **have** $s \in$ *r1* $\rightarrow$ *snd* (*simp r1*) *v* **by** *simp*
moreover
have $[] \in$ *One* $\rightarrow$ *Void* **by** (*simp add*: *Posix-One*)
then have $[] \in$ *fst* (*simp r2*) $\rightarrow$ *Void* **using** *NOne-One* **by** *simp*
then have $[] \in$ *r2* $\rightarrow$ *snd* (*simp r2*) *Void* **using** *IH2* **by** *simp*
ultimately have $(s @ []) \in$ *Times r1 r2* $\rightarrow$ *Seq* (*snd* (*simp r1*) *v*) (*snd* (*simp r2*) *Void*)
by(*rule-tac Posix-Times*) *auto*
then show $s \in$ *Times r1 r2* $\rightarrow$ *snd* (*simp* (*Times r1 r2*)) *v* **using** *NOne-One* **by** *simp*
next
case (*NOne-NOne*)
with *as* **have** $s \in$ *Times* (*fst* (*simp r1*)) (*fst* (*simp r2*)) $\rightarrow$ *v* **by** *simp*
then obtain *s1 s2 v1 v2* **where** *eqs*: *s* = *s1* @ *s2* *v* = *Seq v1 v2*
s1 $\in$ (*fst* (*simp r1*)) $\rightarrow$ *v1* *s2* $\in$ (*fst* (*simp r2*)) $\rightarrow$ *v2*
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ r1 \wedge s_4 \in lang\ r2)$
by (*erule-tac Posix-elims*(*5*)) (*auto simp add*: *L-fst-simp*[*symmetric*])

then have *s1* $\in$ *r1* $\rightarrow$ (*snd* (*simp r1*) *v1*) *s2* $\in$ *r2* $\rightarrow$ (*snd* (*simp r2*) *v2*)
using *IH1 IH2* **by** *auto*
then show $s \in$ *Times r1 r2* $\rightarrow$ *snd* (*simp* (*Times r1 r2*)) *v* **using** *eqs NOne-NOne*
by(*auto intro*: *Posix-Times*)
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*: $\bigwedge r$. *slexer r s* = *lexer r s* **by** *fact*
show *slexer r* (*c* # *s*) = *lexer r* (*c* # *s*)
proof (*cases s* $\in$ *lang* (*deriv c r*))
case *True*
assume *a1*: *s* $\in$ *lang* (*deriv c r*)
then obtain *v1* **where** *a2*: *lexer* (*deriv c r*) *s* = *Some v1 s* $\in$ *deriv c r* $\to$ *v1*
using *lexer-correct-Some* **by** *auto*
from *a1* **have** *s* $\in$ *lang* (*fst* (*simp* (*deriv c r*))) **using** *L-fst-simp*[*symmetric*] **by** *auto*
then obtain *v2* **where** *a3*: *lexer* (*fst* (*simp* (*deriv c r*))) *s* = *Some v2 s* $\in$ (*fst* (*simp* (*deriv c r*))) $\to$ *v2*
using *lexer-correct-Some* **by** *auto*
then have *a4*: *slexer* (*fst* (*simp* (*deriv c r*))) *s* = *Some v2* **using** *IH* **by** *simp*
from *a3*(*2*) **have** *s* $\in$ *deriv c r* $\to$ (*snd* (*simp* (*deriv c r*))) *v2* **using** *Posix-simp* **by** *auto*
with *a2*(*2*) **have** *v1* = (*snd* (*simp* (*deriv c r*))) *v2* **using** *Posix-determ* **by** *auto*
with *a2*(*1*) *a4* **show** *slexer r* (*c* # *s*) = *lexer r* (*c* # *s*) **by** (*auto split*: *prod.split*)
next
case *False*
assume *b1*: *s* $\notin$ *lang* (*deriv c r*)
then have *lexer* (*deriv c r*) *s* = *None* **using** *lexer-correct-None* **by** *auto*
moreover
from *b1* **have** *s* $\notin$ *lang* (*fst* (*simp* (*deriv c r*))) **using** *L-fst-simp*[*symmetric*] **by** *auto*
then have *lexer* (*fst* (*simp* (*deriv c r*))) *s* = *None* **using** *lexer-correct-None* **by** *auto*
then have *slexer* (*fst* (*simp* (*deriv 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 *Lexer LexicalVals*
begin

10 An alternative definition for POSIX values by Okui & Suzuki

11 Positions in Values

fun
  $at :: 'a\ val \Rightarrow nat\ list \Rightarrow 'a\ val$
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 $Pos :: 'a\ val \Rightarrow (nat\ list)\ set$
where
  $Pos\ (Void) = \{[]\}$
$|\ Pos\ (Atm\ c) = \{[]\}$
$|\ Pos\ (Left\ v) = \{[]\} \cup \{0\#ps \mid ps.\ ps \in Pos\ v\}$
$|\ Pos\ (Right\ v) = \{[]\} \cup \{1\#ps \mid ps.\ ps \in Pos\ v\}$
$|\ Pos\ (Seq\ v1\ v2) = \{[]\} \cup \{0\#ps \mid ps.\ ps \in Pos\ v1\} \cup \{1\#ps \mid ps.\ ps \in Pos\ v2\}$
$|\ Pos\ (Stars\ []) = \{[]\}$
$|\ Pos\ (Stars\ (v\#vs)) = \{[]\} \cup \{0\#ps \mid ps.\ ps \in Pos\ v\} \cup \{Suc\ n\#ps \mid n\ ps.\ n\#ps \in Pos\ (Stars\ vs)\}$

lemma *Pos-stars*:
  $Pos\ (Stars\ vs) = \{[]\} \cup (\bigcup n < length\ vs.\ \{n\#ps \mid ps.\ ps \in Pos\ (vs\ !\ n)\})$
apply(*induct vs*)
apply(*auto simp add*: *insert-ident less-Suc-eq-0-disj*)
done

lemma *Pos-empty*:
  shows $[] \in Pos\ v$
by (*induct v rule*: *Pos.induct*)(*auto*)

abbreviation
  $intlen\ vs \equiv int\ (length\ vs)$

definition *pflat-len* $:: 'a\ val \Rightarrow nat\ list => int$

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*)

12 Orderings

definition *prefix-list*:: ′*a list* ⇒ ′*a list* ⇒ *bool* (‹- ⊑*pre* -› [*60*,*59*] *60*)
where
ps1 ⊑*pre ps2* ≡ ∃ *ps*′. *ps1* @*ps*′ = *ps2*

definition *sprefix-list*:: ′*a list* ⇒ ′*a list* ⇒ *bool* (‹- ⊏*spre* -› [*60*,*59*] *60*)
where
ps1 ⊏*spre ps2* ≡ *ps1* ⊑*pre ps2* ∧ *ps1* ≠ *ps2*

inductive *lex-list* :: *nat list* ⇒ *nat list* ⇒ *bool* (‹- ⊏*lex* -› [*60*,*59*] *60*)
where
[] ⊏*lex* (*p#ps*)
| *ps1* ⊏*lex ps2* ⟹ (*p#ps1*) ⊏*lex* (*p#ps2*)
| *p1* < *p2* ⟹ (*p1#ps1*) ⊏*lex* (*p2#ps2*)

lemma *lex-irrfl*:
fixes *ps1 ps2* :: *nat list*
assumes *ps1* ⊏*lex ps2*
shows *ps1* ≠ *ps2*
using *assms*

by(*induct rule*: *lex-list.induct*)(*auto*)

lemma *lex-simps* [*simp*]:
  fixes *xs ys* :: *nat list*
  shows [] ⊏*lex ys* ⟷ *ys* ≠ []
  and *xs* ⊏*lex* [] ⟷ *False*
  and (*x* # *xs*) ⊏*lex* (*y* # *ys*) ⟷ (*x* < *y* ∨ (*x* = *y* ∧ *xs* ⊏*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* ⊏*lex ps2 ps2* ⊏*lex ps3*
  shows *ps1* ⊏*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* ∨ *p* ⊏*lex q* ∨ *q* ⊏*lex p*
apply(*induct p arbitrary*: *q*)
apply(*auto elim*: *lex-list.cases*)
apply(*case-tac q*)
apply(*auto*)
done

13 POSIX Ordering of Values According to Okui & Suzuki

definition *PosOrd*:: ′*a val* ⇒ *nat list* ⇒ ′*a 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*:: ′*a val* ⇒ ′*a val* ⇒ *bool* (‹- :⊏*val* -› [*60*, *59*] *60*)

```
where
  v1 :⊏val v2 ≡ ∃ p. v1 ⊏val p v2

definition PosOrd-ex-eq:: 'a val ⇒ 'a 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 (metis (full-types) not-int-zless-negative[of length (flat (at v2 p))] zero-less-one
        verit-comp-simplify1(1)[of − 1] pos-int-cases[of 1])
      (metis PosOrd-def as(2) int-ops(2) not-int-zless-negative pflat-len-def verit-comp-simplify1(1))
  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 (verit, best) UnCI)
      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 (verit, best) UnCI 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 (verit, best) UnCI lex-trans pflat-len-outside)
      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 $\neg(v2 :\sqsubset val\ v1)$
using *assms*
using *PosOrd-irrefl PosOrd-trans* **by** *blast*

lemma *PosOrd-ordering*:
shows *ordering* $(\lambda v1\ v2.\ v1 :\sqsubseteq val\ v2)$ $(\lambda\ v1\ v2.\ v1 :\sqsubset val\ v2)$
unfolding *ordering-def PosOrd-ex-eq-def*
apply(*auto*)
using *PosOrd-trans partial-preordering-def* **apply** *blast*
using *PosOrd-assym ordering-axioms-def* **by** *blast*

lemma *PosOrd-order*:
shows *class.order* $(\lambda v1\ v2.\ v1 :\sqsubseteq val\ v2)$ $(\lambda\ v1\ v2.\ v1 :\sqsubset val\ v2)$
using *PosOrd-ordering*
apply(*simp add*: *class.order-def class.preorder-def class.order-axioms-def*)
by (*smt* (*verit*) *PosOrd-ex-eq-def PosOrd-irrefl PosOrd-trans*)

lemma *PosOrd-ex-eq2*:
shows $v1 :\sqsubset val\ v2 \longleftrightarrow (v1 :\sqsubseteq val\ v2 \wedge v1 \neq v2)$
using *PosOrd-ordering*
using *PosOrd-ex-eq-def PosOrd-irrefl* **by** *blast*

lemma *PosOrdeq-trans*:
assumes $v1 :\sqsubseteq val\ v2\ v2 :\sqsubseteq val\ v3$
shows $v1 :\sqsubseteq val\ v3$
using *assms PosOrd-ordering*
unfolding *ordering-def*
by (*metis partial-preordering.trans*)

lemma *PosOrdeq-antisym*:
assumes $v1 :\sqsubseteq val\ v2\ v2 :\sqsubseteq val\ v1$
shows $v1 = v2$
using *assms PosOrd-ordering*
by (*metis ordering.eq-iff*)

lemma *PosOrdeq-refl*:
shows $v :\sqsubseteq val\ v$
unfolding *PosOrd-ex-eq-def*
by *auto*

lemma *PosOrd-shorterE*:
assumes $v1 :\sqsubset val\ v2$
shows *length* (*flat* $v2$) $\leq$ *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 (verit) Collect-disj-eq lex-list.intros(2) mem-Collect-eq pflat-len-simps(2))
apply(smt (verit) 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 $v1 :\sqsubset val\ v2 \lor v2 :\sqsubset val\ v1 \lor (length\ (flat\ v1) = length\ (flat\ v2))$
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

14 The Posix Value is smaller than any other lexical value

lemma *Posix-PosOrd*:
  assumes $s \in r \to v1\ v2 \in LV\ r\ s$
  shows $v1 :\sqsubseteq val\ v2$
using *assms*
proof (*induct arbitrary*: *v2 rule*: *Posix.induct*)
  case (*Posix-One v*)
  have $v \in LV\ One\ []$ **by** *fact*
  then have $v = Void$
    by (*simp add*: *LV-simps*)
  then show $Void :\sqsubseteq val\ v$
    by (*simp add*: *PosOrd-ex-eq-def*)
next
  case (*Posix-Atom c v*)
  have $v \in LV\ (Atom\ c)\ [c]$ **by** *fact*
  then have $v = Atm\ c$
    by (*simp add*: *LV-simps*)
  then show $Atm\ c :\sqsubseteq val\ v$
    by (*simp add*: *PosOrd-ex-eq-def*)
next
  case (*Posix-Plus1 s r1 v r2 v2*)
  have *as1*: $s \in r1 \to v$ **by** *fact*
  have *IH*: $\bigwedge v2.\ v2 \in LV\ r1\ s \Longrightarrow v :\sqsubseteq val\ v2$ **by** *fact*
  have $v2 \in LV\ (Plus\ r1\ r2)\ s$ **by** *fact*
  then have $\vdash v2 : Plus\ r1\ r2\ flat\ v2 = s$
    by(*auto simp add*: *LV-def prefix-list-def*)
  then consider
    (*Left*) *v3* **where** $v2 = Left\ v3 \vdash v3 : r1\ flat\ v3 = s$
  | (*Right*) *v3* **where** $v2 = Right\ v3 \vdash v3 : r2\ flat\ v3 = s$
  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-Plus2 s r2 v r1 v2)
  have as1: s ∈ r2 → v by fact
  have as2: s ∉ lang r1 by fact
  have IH: ⋀v2. v2 ∈ LV r2 s ⟹ v :⊑val v2 by fact
  have v2 ∈ LV (Plus r1 r2) s by fact
  then have ⊢ v2 : Plus 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-Times 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: ¬ (∃ s₃ s₄. s₃ ≠ [] ∧ s₃ @ s₄ = s2 ∧ s1 @ s₃ ∈ lang r1 ∧ s₄ ∈ lang
r2) by fact
  have v3 ∈ LV (Times 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 (verit) 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 ∈ lang r ∧ s₄ ∈ lang
(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-elims)
  by (metis NonEmpty Prf.intros(6) list.set-intros(1) list.set-intros(2) neq-Nil-conv)
  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 (verit) Prf-flat-lang append.right-neutral 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 (metis flat.simps(7) flat-Stars val.inject(5))
      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 $v1 \in LV\ r\ s\ \forall v_2 \in LV\ r\ s.\ \neg\ v_2 :\sqsubset val\ v1$
shows $s \in r \to v1$
proof −
have $s \in lang\ r$ **using** *assms*(*1*) **unfolding** *LV-def*
using *L-flat-Prf1* **by** *blast*
then obtain *vposix* **where** *vp*: $s \in r \to vposix$
using *lexer-correct-Some* **by** *blast*
with *assms*(*1*) **have** $vposix :\sqsubseteq val\ v1$ **by** (*simp add*: *Posix-PosOrd*)
then have $vposix = v1 \lor vposix :\sqsubset val\ v1$ **unfolding** *PosOrd-ex-eq2* **by** *auto*
moreover
{ assume $vposix :\sqsubset val\ v1$
moreover
have $vposix \in LV\ r\ s$ **using** *vp*
using *Posix-LV* **by** *blast*
ultimately have *False* **using** *assms*(*2*) **by** *blast*
}
ultimately show $s \in r \to v1$ **using** *vp* **by** *blast*
qed

lemma *Least-existence*:
assumes $LV\ r\ s \neq \{\}$
shows $\exists\, vmin \in LV\ r\ s.\ \forall\, v \in LV\ r\ s.\ vmin :\sqsubseteq val\ v$
proof −
from *assms*
obtain *vposix* **where** $s \in r \to vposix$
unfolding *LV-def*
using *L-flat-Prf1 lexer-correct-Some* **by** *blast*
then have $\forall\, v \in LV\ r\ s.\ vposix :\sqsubseteq val\ v$
by (*simp add*: *Posix-PosOrd*)
then show $\exists\, vmin \in LV\ r\ s.\ \forall\, v \in LV\ r\ s.\ vmin :\sqsubseteq val\ v$
using *Posix-LV* $\langle s \in r \to vposix \rangle$ **by** *blast*
qed

lemma *Least-existence1*:
assumes $LV\ r\ s \neq \{\}$
shows $\exists!\ v_{min} \in LV\ r\ s.\ \forall\, v \in LV\ r\ s.\ v_{min} :\sqsubseteq val\ v$
using *Least-existence*[*OF assms*] *assms*
using *PosOrdeq-antisym* **by** *blast*

lemma *Least-existence2*:
assumes $LV\ r\ s \neq \{\}$
shows $\exists! vmin \in LV\ r\ s.\ lexer\ r\ s = Some\ vmin \land (\forall\, v \in LV\ r\ s.\ vmin :\sqsubseteq val\ v)$
using *Least-existence*[*OF assms*] *assms*
using *PosOrdeq-antisym*
using *PosOrd-Posix PosOrd-ex-eq2 lexer-correctness*(*1*)

by (*metis* (*mono-tags*, *lifting*))

lemma *Least-existence1-pre*:
  assumes $LV\ r\ s \neq \{\}$
  shows $\exists!vmin \in LV\ r\ s.\ \forall v \in (LV\ r\ s \cup \{v'.\ flat\ v' \sqsubset spre\ s\}).\ vmin :\sqsubseteq val\ v$
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 *PosOrd-partial*:
  shows *partial-order-on UNIV* $\{(v1, v2).\ v1 :\sqsubseteq val\ v2\}$
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 *PosOrd-wf*:
  shows *wf* $\{(v1, v2).\ v1 :\sqsubset val\ v2 \wedge v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s\}$
proof −
  have *finite* $\{(v1, v2).\ v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s\}$
    by (*simp add*: *LV-finite*)
  moreover
  have $\{(v1, v2).\ v1 :\sqsubset val\ v2 \wedge v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s\} \subseteq \{(v1, v2).\ v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s\}$
    by *auto*
  ultimately have *finite* $\{(v1, v2).\ v1 :\sqsubset val\ v2 \wedge v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s\}$
    using *finite-subset* **by** *blast*
  moreover
  have *acyclicP* $(\lambda v1\ v2.\ v1 :\sqsubset val\ v2 \wedge v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s)$
    unfolding *acyclic-def*
    by (*smt* (*verit*, *ccfv-threshold*) *PosOrd-irrefl PosOrd-trans tranclp-trans-induct tranclp-unfold*)
  ultimately show *wf* $\{(v1, v2).\ v1 :\sqsubset val\ v2 \wedge v1 \in LV\ r\ s \wedge v2 \in LV\ r\ s\}$
    using *finite-acyclic-wf* **by** *blast*
qed

unused-thms

end

15 Extended Regular Expressions 3

theory *Regular-Exps3*
imports *Regular−Sets.Regular-Set*
begin

datatype (*atoms*: *'a*) *rexp* =
  is-Zero: *Zero* |
  is-One: *One* |
  Atom 'a |
  Plus (*'a rexp*) (*'a rexp*) |
  Times (*'a rexp*) (*'a rexp*) |
  Star (*'a rexp*) |
  NTimes (*'a rexp*) *nat* |
  Upto (*'a rexp*) *nat* |
  From (*'a rexp*) *nat* |
  Rec string (*'a rexp*) |
  Charset (*'a set*)

fun *lang* :: *'a rexp* => *'a lang* **where**
lang Zero = {} |
lang One = {[]} |
lang (*Atom a*) = {[*a*]} |
lang (*Plus r s*) = (*lang r*) *Un* (*lang s*) |
lang (*Times r s*) = *conc* (*lang r*) (*lang s*) |
lang (*Star r*) = *star*(*lang r*) |
lang (*NTimes r n*) = ((*lang r*) ^^ *n*) |
lang (*Upto r n*) = ($\bigcup$ *i* ∈ {..*n*}. (*lang r*) ^^ *i*) |
lang (*From r n*) = ($\bigcup$ *i* ∈ {*n*..}. (*lang r*) ^^ *i*) |
lang (*Rec l r*) = *lang r* |
lang (*Charset cs*) = {[*c*] | *c* . *c* ∈ *cs*}

primrec *nullable* :: *'a rexp* ⇒ *bool* **where**
nullable Zero = *False* |
nullable One = *True* |
nullable (*Atom c*) = *False* |
nullable (*Plus r1 r2*) = (*nullable r1* ∨ *nullable r2*) |
nullable (*Times r1 r2*) = (*nullable r1* ∧ *nullable r2*) |
nullable (*Star r*) = *True* |
nullable (*NTimes r n*) = (*if n* = *0 then True else nullable r*) |
nullable (*Upto r n*) = *True* |
nullable (*From r n*) = (*if n* = *0 then True else nullable r*) |
nullable (*Rec l r*) = *nullable r* |
nullable (*Charset cs*) = *False*

lemma *pow-empty-iff*:
shows $[] \in (lang\ r)$ ^^ $n \longleftrightarrow (\textit{if } n = 0 \textit{ then True else } [] \in (lang\ r))$
by (*induct n*)(*auto*)

lemma *nullable-iff*:
shows *nullable r* $\longleftrightarrow [] \in$ *lang r*
by (*induct r*) (*auto simp add*: *conc-def pow-empty-iff split*: *if-splits*)

end

16 Derivatives of Extended Regular Expressions

theory *Derivatives3*
imports *Regular-Exps3*
begin

This theory is based on work by Brozowski.

16.1 Brzozowski's derivatives of regular expressions

fun
deriv :: *'a* ⇒ *'a rexp* ⇒ *'a rexp*
where
deriv c (*Zero*) = *Zero*
| *deriv c* (*One*) = *Zero*
| *deriv c* (*Atom c'*) = (*if c* = *c' then One else Zero*)
| *deriv c* (*Plus r1 r2*) = *Plus* (*deriv c r1*) (*deriv c r2*)
| *deriv c* (*Times r1 r2*) =
(*if nullable r1 then Plus* (*Times* (*deriv c r1*) *r2*) (*deriv c r2*) *else Times* (*deriv c r1*) *r2*)
| *deriv c* (*Star r*) = *Times* (*deriv c r*) (*Star r*)
| *deriv c* (*NTimes r n*) = (*if n* = *0 then Zero else Times* (*deriv c r*) (*NTimes r* (*n* − *1*)))
| *deriv c* (*Upto r n*) = (*if n* = *0 then Zero else Times* (*deriv c r*) (*Upto r* (*n* − *1*)))
| *deriv c* (*From r n*) = (*if n* = *0 then Times* (*deriv c r*) (*Star r*) *else Times* (*deriv c r*) (*From r* (*n* − *1*)))
| *deriv c* (*Rec l r*) = *deriv c r*
| *deriv c* (*Charset cs*) = (*if c* ∈ *cs then One else Zero*)

fun
derivs :: *'a list* ⇒ *'a rexp* ⇒ *'a rexp*
where
derivs [] *r* = *r*
| *derivs* (*c* # *s*) *r* = *derivs s* (*deriv c r*)

lemma *deriv-pow* [*simp*]:

shows *Deriv c* $(A \frown\frown n) = (if\ n = 0\ then\ \{\}\ else\ (Deriv\ c\ A)\ @@\ (A \frown\frown (n - 1)))$
apply(*induct n arbitrary*: *A*)
apply(*auto*)
by (*metis Suc-pred concI-if-Nil2 conc-assoc conc-pow-comm lang-pow.simps*(*2*))

lemma *lang-deriv*: *lang* (*deriv c r*) = *Deriv c* (*lang r*)
apply (*induct r rule*: *lang.induct*)
apply(*auto simp add*: *nullable-iff conc-UNION-distrib*)
apply (*metis IntI Suc-pred atMost-iff diff-Suc-1 mem-Collect-eq not-less-eq-eq zero-less-Suc*)
apply(*auto*)
apply(*simp add*: *conc-def*)
apply(*metis diff-Suc-Suc minus-nat.diff-0 star-pow zero-less-Suc*)
apply(*metis IntI Suc-le-mono Suc-pred atLeast-iff diff-Suc-1 mem-Collect-eq zero-less-Suc*)
apply(*auto simp add*: *Deriv-def*)
done

lemma *lang-derivs*: *lang* (*derivs s r*) = *Derivs s* (*lang r*)
by (*induct s arbitrary*: *r*) (*simp-all add*: *lang-deriv*)

A regular expression matcher:

definition *matcher* :: *'a rexp* ⇒ *'a list* ⇒ *bool* **where**
matcher r s = *nullable* (*derivs s r*)

lemma *matcher-correctness*: *matcher r s* ⟷ *s* ∈ *lang r*
by (*induct s arbitrary*: *r*)
(*simp-all add*: *nullable-iff lang-deriv matcher-def Deriv-def*)

end

theory *Lexer3*
imports *Derivatives3*
begin

17 Values

datatype *'a val* =
Void
| *Atm 'a*
| *Seq 'a val 'a val*
| *Right 'a val*
| *Left 'a val*
| *Stars* (*'a val*) *list*
| *Recv string 'a val*

18 The string behind a value

fun
flat :: *'a val* ⇒ *'a list*
where
flat (*Void*) = []
| *flat* (*Atm 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*))
| *flat* (*Recv l v*) = *flat v*

abbreviation
flats vs ≡ *concat* (*map flat vs*)

lemma *flat-Stars* [*simp*]:
flat (*Stars vs*) = *concat* (*map flat vs*)
by (*induct vs*) (*auto*)

lemma *flats-empty*:
assumes (∀ *v*∈*set vs*. *flat v* = [])
shows *flats vs* = []
using *assms*
by(*induct vs*) (*simp-all*)

19 Relation between values and regular expressions

inductive
Prf :: *'a val* ⇒ *'a rexp* ⇒ *bool* (‹⊢ - : -› [*100*, *100*] *100*)
where
⟦⊢ *v1* : *r1*; ⊢ *v2* : *r2*⟧ ⟹ ⊢ *Seq v1 v2* : *Times r1 r2*
| ⊢ *v1* : *r1* ⟹ ⊢ *Left v1* : *Plus r1 r2*
| ⊢ *v2* : *r2* ⟹ ⊢ *Right v2* : *Plus r1 r2*
| ⊢ *Void* : *One*
| ⊢ *Atm c* : *Atom c*
| ⟦∀ *v* ∈ *set vs*. ⊢ *v* : *r* ∧ *flat v* ≠ []⟧ ⟹ ⊢ *Stars vs* : *Star r*
| ⟦∀ *v* ∈ *set vs1*. ⊢ *v* : *r* ∧ *flat v* ≠ [];
∀ *v* ∈ *set vs2*. ⊢ *v* : *r* ∧ *flat v* = [];
length (*vs1* @ *vs2*) = *n*⟧ ⟹ ⊢ *Stars* (*vs1* @ *vs2*) : *NTimes r n*
| ⟦∀ *v* ∈ *set vs*. ⊢ *v* : *r* ∧ *flat v* ≠ []; *length vs* ≤ *n*⟧ ⟹ ⊢ *Stars vs* : *Upto r n*
| ⟦∀ *v* ∈ *set vs1*. ⊢ *v* : *r* ∧ *flat v* ≠ [];
∀ *v* ∈ *set vs2*. ⊢ *v* : *r* ∧ *flat v* = [];
length (*vs1* @ *vs2*) = *n*⟧ ⟹ ⊢ *Stars* (*vs1* @ *vs2*) : *From r n*
| ⟦∀ *v* ∈ *set vs*. ⊢ *v* : *r* ∧ *flat v* ≠ []; *length vs* > *n*⟧ ⟹ ⊢ *Stars vs* : *From r n*
| ⊢ *v* : *r* ⟹ ⊢ *Recv l v* : *Rec l r*

| $c \in cs \Longrightarrow \vdash Atm\ c : Charset\ cs$

inductive-cases *Prf-elims*:
$\vdash v : Zero$
$\vdash v : Times\ r1\ r2$
$\vdash v : Plus\ r1\ r2$
$\vdash v : One$
$\vdash v : Atom\ c$
$\vdash v : Star\ r$
$\vdash v : NTimes\ r\ n$
$\vdash v : Upto\ r\ n$
$\vdash v : From\ r\ n$
$\vdash v : Rec\ l\ r$
$\vdash v : Charset\ cs$

lemma *Prf-NTimes-empty*:
assumes $\forall v \in set\ vs.\ \vdash v : r \wedge flat\ v = []$
and $length\ vs = n$
shows $\vdash Stars\ vs : NTimes\ r\ n$
using *assms*
by (*metis Prf.intros*(*7*) *empty-iff eq-Nil-appendI list.set*(*1*))

lemma *Times-decomp*:
assumes $s \in A\ @@\ B$
shows $\exists s1\ s2.\ s = s1\ @\ s2 \wedge s1 \in A \wedge s2 \in B$
using *assms*
by *blast*

lemma *pow-string*:
assumes $s \in A\ \widehat{}\widehat{}\ n$
shows $\exists ss.\ concat\ ss = s \wedge (\forall s \in set\ ss.\ s \in A) \wedge length\ ss = n$
using *assms*
apply(*induct n arbitrary*: *s*)
apply(*auto dest*!: *Times-decomp*)
apply(*drule-tac x=s2* **in** *meta-spec*)
apply(*auto*)
apply(*rule-tac x=s1 # ss* **in** *exI*)
apply(*simp*)
done

lemma *pow-Prf*:
assumes $\forall v \in set\ vs.\ \vdash v : r \wedge flat\ v \in A$
shows $flats\ vs \in A\ \widehat{}\widehat{}\ (length\ vs)$
using *assms*
by (*induct vs*) (*auto*)

lemma *Star-string*:
assumes $s \in star\ A$

```
  shows ∃ss. concat ss = s ∧ (∀s ∈ set ss. s ∈ A)
using assms
by (metis in-star-iff-concat subsetD)

lemma Star-val:
  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 (metis empty-iff list.set(1))
by (metis append.simps(1) flat.simps(7) flat-Stars set-ConsD)

lemma Aux:
  assumes ∀s∈set ss. s = []
  shows concat ss = []
using assms
by (induct ss) (auto)

lemma pow-cstring:
  assumes s ∈ A ⌢⌢ n
  shows ∃ss1 ss2. concat (ss1 @ ss2) = s ∧ length (ss1 @ ss2) = n ∧
        (∀s ∈ set ss1. s ∈ A ∧ s ≠ []) ∧ (∀s ∈ set ss2. s ∈ A ∧ s = [])
using assms
apply(induct n arbitrary: s)
  apply(auto)[1]
  apply(auto dest!: Times-decomp simp add: Seq-def)
  apply(drule-tac x=s2 in meta-spec)
  apply(simp)
apply(erule exE)+
  apply(clarify)
apply(case-tac s1 = [])
apply(simp)
apply(rule-tac x=ss1 in exI)
apply(rule-tac x=s1 # ss2 in exI)
apply(simp)
apply(rule-tac x=s1 # ss1 in exI)
apply(rule-tac x=ss2 in exI)
  apply(simp)
  done

lemma flats-cval:
  assumes ∀s∈set ss. ∃v. s = flat v ∧ ⊢ v : r
  shows ∃vs1 vs2. flats vs1 = concat ss ∧ length (vs1 @ vs2) = length ss ∧
         (∀v∈set vs1. ⊢ v : r ∧ flat v ≠ []) ∧
         (∀v∈set vs2. ⊢ v : r ∧ flat v = [])
using assms
apply(induct ss rule: rev-induct)
apply(rule-tac x=[] in exI)+
```

apply(*simp*)
apply(*simp*)
  apply(*clarify*)
  apply(*case-tac flat v* = [])
  apply(*rule-tac x*=*vs1* **in** *exI*)
  apply(*simp*)
apply(*rule-tac x*=*v*#*vs2* **in** *exI*)
apply(*simp*)
  apply(*rule-tac x*=*vs1* @ [*v*] **in** *exI*)
  apply(*simp*)
apply(*rule-tac x*=*vs2* **in** *exI*)
apply(*simp*)
  done

lemma *flats-cval2*:
  assumes ∀ *s*∈*set ss*. ∃ *v*. *s* = *flat v* ∧ ⊢ *v* : *r*
  shows ∃ *vs*. *flats vs* = *concat ss* ∧ *length vs* ≤ *length ss* ∧ (∀ *v*∈*set vs*. ⊢ *v* : *r* ∧ *flat v* ≠ [])
  using *assms*
  apply −
  apply(*drule flats-cval*)
  apply(*auto*)
  done

lemma *Prf-flat-lang*:
  assumes ⊢ *v* : *r* **shows** *flat v* ∈ *lang r*
using *assms*
  apply(*induct v r rule*: *Prf.induct*)
  apply(*auto simp add*: *concat-in-star subset-eq lang-pow-add*)
  apply(*meson concI pow-Prf*)
  apply(*meson atMost-iff pow-Prf*)
  apply(*subgoal-tac flats vs1* @ *flats vs2* ∈ *lang r* ^^ *length vs1*)
  apply (*metis add-diff-cancel-left′ atLeast-iff diff-is-0-eq empty-pow-add last-in-set length-0-conv order-refl*)
  apply (*metis* (*no-types*, *opaque-lifting*) *Aux imageE list.set-map pow-Prf self-append-conv*)
  apply (*meson atLeast-iff less-imp-le-nat pow-Prf*)
  done

lemma *L-flat-Prf2*:
  assumes *s* ∈ *lang r*
  shows ∃ *v*. ⊢ *v* : *r* ∧ *flat v* = *s*
using *assms*
proof(*induct r arbitrary*: *s*)
  case (*Star r s*)
  have *IH*: ⋀*s*. *s* ∈ *lang r* ⟹ ∃ *v*.⊢ *v* : *r* ∧ *flat v* = *s* **by** *fact*
  have *s* ∈ *lang* (*Star r*) **by** *fact*
  then obtain *ss* **where** *concat ss* = *s* ∀ *s* ∈ *set ss*. *s* ∈ *lang r* ∧ *s* ≠ []
    by (*smt* (*verit*) *Nil-eq-concat-conv concat-append lang.simps*(*6*) *pow-cstring*

```
self-append-conv
      star-pow)
  then obtain vs where flats vs = s ∀ v∈set vs. ⊢ v : r ∧ flat v ≠ []
  using IH by (metis Star-val)
  then show ∃ v. ⊢ v : Star r ∧ flat v = s
  using Prf.intros(6) flat-Stars by blast
next
  case (Times r1 r2 s)
  then show ∃ v. ⊢ v : Times r1 r2 ∧ flat v = s
  unfolding Seq-def lang.simps by (fastforce intro: Prf.intros)
next
  case (Plus r1 r2 s)
  then show ∃ v. ⊢ v : Plus r1 r2 ∧ flat v = s
  unfolding lang.simps by (fastforce intro: Prf.intros)
next
  case (NTimes r n)
  have IH: ⋀s. s ∈ lang r ⟹ ∃ v. ⊢ v : r ∧ flat v = s by fact
  have s ∈ lang (NTimes r n) by fact
  then obtain ss1 ss2 where concat (ss1 @ ss2) = s length (ss1 @ ss2) = n
    ∀ s ∈ set ss1. s ∈ lang r ∧ s ≠ [] ∀ s ∈ set ss2. s ∈ lang r ∧ s = []
  using pow-cstring by force
  then obtain vs1 vs2 where flats (vs1 @ vs2) = s length (vs1 @ vs2) = n
      ∀ v∈set vs1. ⊢ v : r ∧ flat v ≠ [] ∀ v∈set vs2. ⊢ v : r ∧ flat v = []
    using IH flats-cval
  apply −
  apply(drule-tac x=ss1 @ ss2 in meta-spec)
  apply(drule-tac x=r in meta-spec)
  apply(drule meta-mp)
  apply(simp)
  apply (metis Un-iff)
  apply(clarify)
  apply(drule-tac x=vs1 in meta-spec)
  apply(drule-tac x=vs2 in meta-spec)
  apply(simp)
  done
  then show ∃ v. ⊢ v : NTimes r n ∧ flat v = s
    using Prf.intros(7) flat-Stars by blast
next
  case (Upto r n)
  have IH: ⋀s. s ∈ lang r ⟹ ∃ v.⊢ v : r ∧ flat v = s by fact
  have s ∈ lang (Upto r n) by fact
  then obtain ss where concat ss = s ∀ s ∈ set ss. s ∈ lang r ∧ s ≠ [] length ss
≤ n
    apply(auto)
   by (smt (verit) Nil-eq-concat-conv pow-cstring concat-append le0 le-add-same-cancel1
le-trans length-append self-append-conv)
  then obtain vs where flats vs = s ∀ v∈set vs. ⊢ v : r ∧ flat v ≠ [] length vs ≤
n
  using IH flats-cval2
```

by (*smt* (*verit*, *best*) *le-trans*)
then show $\exists v.\ \vdash v : Upto\ r\ n \wedge flat\ v = s$
by (*meson Prf.intros*(*8*) *flat-Stars*)
next
case (*From r n*)
have *IH*: $\bigwedge s.\ s \in lang\ r \Longrightarrow \exists v.\ \vdash v : r \wedge flat\ v = s$ **by** *fact*
have $s \in lang\ (From\ r\ n)$ **by** *fact*
then obtain *ss1 ss2 k* **where** $concat\ (ss1\ @\ ss2) = s\ length\ (ss1\ @\ ss2) = k\ \ n \leq k$
$\forall s \in set\ ss1.\ s \in lang\ r \wedge s \neq []\ \forall s \in set\ ss2.\ s \in lang\ r \wedge s = []$
using *pow-cstring* **by** *force*
then obtain *vs1 vs2* **where** $flats\ (vs1\ @\ vs2) = s\ length\ (vs1\ @\ vs2) = k\ n \leq k$
$\forall v{\in}set\ vs1.\ \vdash v : r \wedge flat\ v \neq []\ \forall v{\in}set\ vs2.\ \vdash v : r \wedge flat\ v = []$
using *IH flats-cval*
apply −
apply(*drule-tac x=ss1* @ *ss2* **in** *meta-spec*)
apply(*drule-tac x=r* **in** *meta-spec*)
apply(*drule meta-mp*)
apply(*simp*)
apply (*metis Un-iff*)
apply(*clarify*)
apply(*drule-tac x=vs1* **in** *meta-spec*)
apply(*drule-tac x=vs2* **in** *meta-spec*)
apply(*simp*)
done
then show $\exists v.\ \vdash v : From\ r\ n \wedge flat\ v = s$
apply(*case-tac* $length\ vs1 \leq n$)
apply(*rule-tac x=Stars* ($vs1\ @\ take\ (n - length\ vs1)\ vs2$) **in** *exI*)
apply(*simp*)
apply(*subgoal-tac* $flats\ (take\ (n - length\ vs1)\ vs2) = []$)
apply(*auto*)
apply(*rule Prf.intros*(*9*))
apply(*auto*)
apply (*meson in-set-takeD*)
apply (*simp add*: *Aux*)
apply (*meson in-set-takeD*)
apply(*rule-tac x=Stars vs1* **in** *exI*)
by (*simp add*: *Prf.intros*(*10*))
next
case (*Rec l r*)
then show *?case* **apply**(*auto*)
using *Prf.intros*(*11*) *flat.simps*(*8*) **by** *blast*
qed (*auto intro*: *Prf.intros*)

lemma *L-flat-Prf*:
$lang\ r = \{flat\ v \mid v.\ \vdash v : r\}$
using *L-flat-Prf2 Prf-flat-lang* **by** *blast*

20 Sulzmann and Lu functions

fun
mkeps :: *'a rexp* ⇒ *'a val*
where
mkeps(*One*) = *Void*
| *mkeps*(*Times r1 r2*) = *Seq* (*mkeps r1*) (*mkeps r2*)
| *mkeps*(*Plus r1 r2*) = (*if nullable*(*r1*) *then Left* (*mkeps r1*) *else Right* (*mkeps r2*))
| *mkeps*(*Star r*) = *Stars* []
| *mkeps*(*Upto r n*) = *Stars* []
| *mkeps*(*NTimes r n*) = *Stars* (*replicate n* (*mkeps r*))
| *mkeps*(*From r n*) = *Stars* (*replicate n* (*mkeps r*))
| *mkeps*(*Rec l r*) = *Recv l* (*mkeps r*)

fun *injval* :: *'a rexp* ⇒ *'a* ⇒ *'a val* ⇒ *'a val*
where
injval (*Atom d*) *c Void* = *Atm c*
| *injval* (*Plus r1 r2*) *c* (*Left v1*) = *Left*(*injval r1 c v1*)
| *injval* (*Plus r1 r2*) *c* (*Right v2*) = *Right*(*injval r2 c v2*)
| *injval* (*Times r1 r2*) *c* (*Seq v1 v2*) = *Seq* (*injval r1 c v1*) *v2*
| *injval* (*Times r1 r2*) *c* (*Left* (*Seq v1 v2*)) = *Seq* (*injval r1 c v1*) *v2*
| *injval* (*Times 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*)
| *injval* (*NTimes r n*) *c* (*Seq v* (*Stars vs*)) = *Stars* ((*injval r c v*) # *vs*)
| *injval* (*Upto r n*) *c* (*Seq v* (*Stars vs*)) = *Stars* ((*injval r c v*) # *vs*)
| *injval* (*From r n*) *c* (*Seq v* (*Stars vs*)) = *Stars* ((*injval r c v*) # *vs*)
| *injval* (*Rec l r*) *c v* = *Recv l* (*injval r c v*)
| *injval* (*Charset cs*) *c Void* = *Atm c*

21 Mkeps, injval

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

lemma *mkeps-nullable*:
assumes *nullable r*
shows ⊢ *mkeps r* : *r*
using *assms*
apply (*induct r*)
apply (*auto intro*: *Prf.intros split*: *if-splits*)
apply (*metis Prf.intros*(*7*) *append-Nil2 in-set-replicate list.size*(*3*) *replicate-0*)
apply(*rule Prf-NTimes-empty*)
apply(*auto simp add*: *mkeps-flat*)
apply (*metis Prf.intros*(*9*) *append-Nil empty-iff list.set*(*1*) *list.size*(*3*))
by (*metis Prf.intros*(*9*) *append-Nil empty-iff in-set-replicate length-replicate list.set*(*1*) *mkeps-flat*)

lemma *Prf-injval-flat*:
  assumes ⊢ *v* : *deriv c r*
  shows *flat* (*injval r c v*) = *c* # (*flat v*)
using *assms*
apply(*induct c r arbitrary*: *v rule*: *deriv.induct*)
apply(*auto elim*!: *Prf-elims intro*: *mkeps-flat split*: *if-splits*)
done

lemma *Prf-injval*:
  assumes ⊢ *v* : *deriv 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 simp add*: *Prf-injval-flat split*: *if-splits*)[*7*]

apply(*case-tac x2*)
apply(*simp*)
apply(*simp*)
apply(*subst append.simps*(*2*)[*symmetric*])
apply(*rule Prf.intros*)
apply(*auto simp add*: *Prf-injval-flat*)[*4*]

apply(*case-tac x2*)
apply(*simp*)
using *Prf-elims*(*1*) **apply** *blast*
apply(*simp*)
apply(*erule Prf-elims*)
apply(*erule Prf-elims*(*8*))
apply(*simp*)
apply(*rule Prf.intros*(*8*))
apply(*auto simp add*: *Prf-injval-flat*)[*2*]

apply(*simp*)
apply(*case-tac x2*)
apply(*simp*)
apply(*erule Prf-elims*)
apply(*simp*)
apply(*erule Prf-elims*(*6*))
apply(*simp*)
apply (*simp add*: *Prf.intros*(*10*) *Prf-injval-flat*)
apply(*simp*)
apply(*erule Prf-elims*)
apply(*simp*)
apply(*erule Prf-elims*(*9*))
apply(*simp*)
apply (*smt* (*verit*, *best*) *Cons-eq-appendI Prf.intros*(*9*) *Prf-injval-flat length-Cons length-append list.discI set-ConsD*)

apply(*simp add*: *Prf.intros*(*10*) *Prf-injval-flat*)
apply(*simp add*: *Prf.intros*(*11*))
by (*metis Prf.intros*(*12*) *Prf-elims*(*1*) *Prf-elims*(*4*) *deriv.simps*(*11*) *injval.simps*(*12*))

22 Our Alternative Posix definition

inductive
  Posix :: $'a$ *list* $\Rightarrow$ $'a$ *rexp* $\Rightarrow$ $'a$ *val* $\Rightarrow$ *bool* (‹- $\in$ - $\rightarrow$ -› [*100*, *100*, *100*] *100*)
where
  Posix-One: $[] \in One \rightarrow Void$
| *Posix-Atom*: $[c] \in (Atom\ c) \rightarrow (Atm\ c)$
| *Posix-Plus1*: $s \in r1 \rightarrow v \Longrightarrow s \in (Plus\ r1\ r2) \rightarrow (Left\ v)$
| *Posix-Plus2*: $[\![s \in r2 \rightarrow v;\ s \notin lang\ r1]\!] \Longrightarrow s \in (Plus\ r1\ r2) \rightarrow (Right\ v)$
| *Posix-Times*: $[\![s1 \in r1 \rightarrow v1;\ s2 \in r2 \rightarrow v2;$
    $\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge (s1\ @\ s_3) \in lang\ r1 \wedge s_4 \in lang\ r2)]\!] \Longrightarrow$
    $(s1\ @\ s2) \in (Times\ r1\ r2) \rightarrow (Seq\ v1\ v2)$
| *Posix-Star1*: $[\![s1 \in r \rightarrow v;\ s2 \in Star\ r \rightarrow Stars\ vs;\ flat\ v \neq [];$
    $\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge (s1\ @\ s_3) \in lang\ r \wedge s_4 \in lang\ (Star\ r))]\!]$
    $\Longrightarrow (s1\ @\ s2) \in Star\ r \rightarrow Stars\ (v\ \#\ vs)$
| *Posix-Star2*: $[] \in Star\ r \rightarrow Stars\ []$
| *Posix-NTimes1*: $[\![s1 \in r \rightarrow v;\ s2 \in NTimes\ r\ n \rightarrow Stars\ vs;\ flat\ v \neq [];$
    $\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge (s1\ @\ s_3) \in lang\ r \wedge s_4 \in lang\ (NTimes\ r$
$n))]\!]$
    $\Longrightarrow (s1\ @\ s2) \in NTimes\ r\ (n + 1) \rightarrow Stars\ (v\ \#\ vs)$
| *Posix-NTimes2*: $[\![\forall v \in set\ vs.\ [] \in r \rightarrow v;\ length\ vs = n]\!]$
    $\Longrightarrow [] \in NTimes\ r\ n \rightarrow Stars\ vs$
| *Posix-Upto1*: $[\![s1 \in r \rightarrow v;\ s2 \in Upto\ r\ n \rightarrow Stars\ vs;\ flat\ v \neq [];$
    $\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge (s1\ @\ s_3) \in lang\ r \wedge s_4 \in lang\ (Upto\ r\ n))]\!]$
    $\Longrightarrow (s1\ @\ s2) \in Upto\ r\ (n + 1) \rightarrow Stars\ (v\ \#\ vs)$
| *Posix-Upto2*: $[] \in Upto\ r\ n \rightarrow Stars\ []$
| *Posix-From2*: $[\![\forall v \in set\ vs.\ [] \in r \rightarrow v;\ length\ vs = n]\!]$
    $\Longrightarrow [] \in From\ r\ n \rightarrow Stars\ vs$
| *Posix-From1*: $[\![s1 \in r \rightarrow v;\ s2 \in From\ r\ (n - 1) \rightarrow Stars\ vs;\ flat\ v \neq [];\ 0 < n;$
    $\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge (s1\ @\ s_3) \in lang\ r \wedge s_4 \in lang\ (From\ r\ (n$
$- 1)))]\!]$
    $\Longrightarrow (s1\ @\ s2) \in From\ r\ n \rightarrow Stars\ (v\ \#\ vs)$
| *Posix-From3*: $[\![s1 \in r \rightarrow v;\ s2 \in Star\ r \rightarrow Stars\ vs;\ flat\ v \neq [];$
    $\neg(\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge (s1\ @\ s_3) \in lang\ r \wedge s_4 \in lang\ (Star\ r))]\!]$
    $\Longrightarrow (s1\ @\ s2) \in From\ r\ 0 \rightarrow Stars\ (v\ \#\ vs)$
| *Posix-Rec*: $s \in r \rightarrow v \Longrightarrow s \in (Rec\ l\ r) \rightarrow (Recv\ l\ v)$
| *Posix-Cset*: $c \in cs \Longrightarrow [c] \in (Charset\ cs) \rightarrow (Atm\ c)$

inductive-cases *Posix-elims*:
  $s \in Zero \rightarrow v$
  $s \in One \rightarrow v$
  $s \in Atom\ c \rightarrow v$
  $s \in Plus\ r1\ r2 \rightarrow v$
  $s \in Times\ r1\ r2 \rightarrow v$
  $s \in Star\ r \rightarrow v$

$s \in NTimes\ r\ n \to v$
$s \in Upto\ r\ n \to v$
$s \in From\ r\ n \to v$
$s \in Rec\ l\ r \to v$
$s \in Charset\ cs \to v$

lemma *Posix1*:
assumes $s \in r \to v$
shows $s \in lang\ r\ flat\ v = s$
using *assms*
apply (*induct s r v rule*: *Posix.induct*)
apply(*auto simp add*: *pow-empty-iff*)
apply (*meson ex-in-conv set-empty*)
apply(*metis Suc-pred atMost-iff concI lang-pow.simps*(*2*) *not-less-eq-eq*)
apply (*meson atLeast-iff dual-order.refl in-set-conv-nth*)
apply (*metis Suc-le-mono Suc-pred atLeast-iff concI lang-pow.simps*(*2*))
by (*simp add*: *star-pow*)

lemma *Posix1a*:
assumes $s \in r \to v$
shows $\vdash v : r$
using *assms*
apply(*induct s r v rule*: *Posix.induct*)
apply(*auto intro*: *Prf.intros*)
apply (*metis Prf.intros*(*6*) *Prf-elims*(*6*) *set-ConsD val.inject*(*5*))
prefer *2*
using *Posix1*(*2*) *Prf-NTimes-empty* **apply** *blast*
apply(*erule Prf-elims*)
apply(*auto*)
apply(*subst append.simps*(*2*)[*symmetric*])
apply(*rule Prf.intros*)
apply(*auto*)
apply (*metis* (*no-types*, *lifting*) *Prf.intros*(*8*) *Prf-elims*(*8*) *Suc-le-mono length-Cons set-ConsD val.inject*(*5*))
apply (*metis Posix1*(*2*) *Prf.intros*(*9*) *append-Nil empty-iff list.set*(*1*))
apply(*erule Prf-elims*)
apply(*auto*)
apply (*smt* (*verit*, *best*) *Cons-eq-appendI Prf.intros*(*9*) *Suc-pred length-Cons length-append set-ConsD*)
apply (*simp add*: *Prf.intros*(*10*))
apply(*erule Prf-elims*)
apply(*auto*)
by (*simp add*: *Prf.intros*(*10*))

lemma *Posix-mkeps*:
assumes *nullable r*
shows $[] \in r \to mkeps\ r$

```
using assms
apply(induct r)
apply(auto intro: Posix.intros simp add: nullable-iff)
apply(subst append.simps(1)[symmetric])
apply(rule Posix.intros)
apply(auto)
apply(simp add: Posix-NTimes2 pow-empty-iff)
apply(simp add: Posix-From2 pow-empty-iff)
done

lemma List-eq-zipI:
  assumes ∀ (v1, v2) ∈ set (zip vs1 vs2). v1 = v2
  and length vs1 = length vs2
  shows vs1 = vs2
 using assms
  apply(induct vs1 arbitrary: vs2)
   apply(case-tac vs2)
   apply(simp)
   apply(simp)
   apply(case-tac vs2)
   apply(simp)
  apply(simp)
done
```

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-Atom c v2)
  have [c] ∈ Atom c → v2 by fact
  then show Atm c = v2 by cases auto
next
  case (Posix-Plus1 s r1 v r2 v2)
  have s ∈ Plus r1 r2 → v2 by fact
  moreover
  have s ∈ r1 → v by fact
  then have s ∈ lang 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 ($Posix\text{-}Plus2\ s\ r2\ v\ r1\ v2$)
have $s \in Plus\ r1\ r2 \to v2$ **by** *fact*
moreover
have $s \notin lang\ r1$ **by** *fact*
ultimately obtain v' **where** eq: $v2 = Right\ v'\ s \in r2 \to v'$
by *cases* ($auto\ simp\ add$: $Posix1$)
moreover
have IH: $\bigwedge v2.\ s \in r2 \to v2 \Longrightarrow v = v2$ **by** *fact*
ultimately have $v = v'$ **by** *simp*
then show $Right\ v = v2$ **using** *eq* **by** *simp*
next
case ($Posix\text{-}Times\ s1\ r1\ v1\ s2\ r2\ v2\ v'$)
have $(s1\ @\ s2) \in Times\ r1\ r2 \to v'$
$s1 \in r1 \to v1\ s2 \in r2 \to v2$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge s1\ @\ s_3 \in lang\ r1 \wedge s_4 \in lang\ r2)$ **by** *fact+*
then obtain $v1'\ v2'$ **where** $v' = Seq\ v1'\ v2'\ s1 \in r1 \to v1'\ s2 \in r2 \to v2'$
apply(*cases*) **apply** ($auto\ simp\ add$: $append\text{-}eq\text{-}append\text{-}conv2$)
using $Posix1(1)$ **by** *fastforce+*
moreover
have IHs: $\bigwedge v1'.\ s1 \in r1 \to v1' \Longrightarrow v1 = v1'$
$\bigwedge v2'.\ s2 \in r2 \to v2' \Longrightarrow v2 = v2'$ **by** *fact+*
ultimately show $Seq\ v1\ v2 = v'$ **by** *simp*
next
case ($Posix\text{-}Star1\ s1\ r\ v\ s2\ vs\ v2$)
have $(s1\ @\ s2) \in Star\ r \to v2$
$s1 \in r \to v\ s2 \in Star\ r \to Stars\ vs\ flat\ v \neq []$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge s1\ @\ s_3 \in lang\ r \wedge s_4 \in lang\ (Star\ r))$
by *fact+*
then obtain $v'\ vs'$ **where** $v2 = Stars\ (v'\ \#\ vs')\ s1 \in r \to v'\ s2 \in (Star\ r) \to (Stars\ vs')$
apply(*cases*) **apply** ($auto\ simp\ add$: $append\text{-}eq\text{-}append\text{-}conv2$)
using $Posix1(1)$ **apply** *fastforce*
apply ($metis\ Posix1(1)\ Posix\text{-}Star1.hyps(6)\ append\text{-}Nil\ append\text{-}Nil2$)
using $Posix1(2)$ **by** *blast*
moreover
have IHs: $\bigwedge v2.\ s1 \in r \to v2 \Longrightarrow v = v2$
$\bigwedge v2.\ s2 \in Star\ r \to v2 \Longrightarrow Stars\ vs = v2$ **by** *fact+*
ultimately show $Stars\ (v\ \#\ vs) = v2$ **by** *auto*
next
case ($Posix\text{-}Star2\ r\ v2$)
have $[] \in Star\ r \to v2$ **by** *fact*
then show $Stars\ [] = v2$ **by** *cases* ($auto\ simp\ add$: $Posix1$)
next
case ($Posix\text{-}NTimes2\ vs\ r\ n\ v2$)
then show $Stars\ vs = v2$
apply($erule\text{-}tac\ Posix\text{-}elims$)
apply(*auto*)
apply ($simp\ add$: $Posix1(2)$)

apply(*rule List-eq-zipI*)
 apply(*auto*)
by (*meson in-set-zipE*)
next
case (*Posix-NTimes1 s1 r v s2 n vs*)
have $(s1 @ s2) \in NTimes\ r\ (n + 1) \to v2$
$s1 \in r \to v\ s2 \in NTimes\ r\ n \to Stars\ vs\ flat\ v \neq []$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ r \wedge s_4 \in lang\ (NTimes\ r\ n))$ **by** *fact+*
then obtain $v'\ vs'$ **where** $v2 = Stars\ (v' \# vs')\ s1 \in r \to v'\ s2 \in (NTimes\ r\ n) \to (Stars\ vs')$
apply(*cases*) **apply** (*auto simp add*: *append-eq-append-conv2*)
using *Posix1*(*1*) **apply** *fastforce*
apply (*metis Posix1*(*1*) *Posix-NTimes1.hyps*(*6*) *append.right-neutral append-Nil*)
using *Posix1*(*2*) **by** *blast*
moreover
have *IHs*: $\bigwedge v2.\ s1 \in r \to v2 \Longrightarrow v = v2$
$\bigwedge v2.\ s2 \in NTimes\ r\ n \to v2 \Longrightarrow Stars\ vs = v2$ **by** *fact+*
ultimately show $Stars\ (v \# vs) = v2$ **by** *auto*
next
case (*Posix-Upto1 s1 r v s2 n vs*)
have $(s1 @ s2) \in Upto\ r\ (n + 1) \to v2$
$s1 \in r \to v\ s2 \in Upto\ r\ n \to Stars\ vs\ flat\ v \neq []$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ r \wedge s_4 \in lang\ (Upto\ r\ n))$ **by** *fact+*
then obtain $v'\ vs'$ **where** $v2 = Stars\ (v' \# vs')\ s1 \in r \to v'\ s2 \in (Upto\ r\ n) \to (Stars\ vs')$
apply(*cases*) **apply** (*auto simp add*: *append-eq-append-conv2*)
using *Posix1*(*1*) **apply** *fastforce*
apply (*metis Posix1*(*1*) *Posix-Upto1.hyps*(*6*) *append.right-neutral append-Nil*)
using *Posix1*(*2*) **by** *blast*
moreover
have *IHs*: $\bigwedge v2.\ s1 \in r \to v2 \Longrightarrow v = v2$
$\bigwedge v2.\ s2 \in Upto\ r\ n \to v2 \Longrightarrow Stars\ vs = v2$ **by** *fact+*
ultimately show $Stars\ (v \# vs) = v2$ **by** *auto*
next
case (*Posix-Upto2 r n*)
have $[] \in Upto\ r\ n \to v2$ **by** *fact*
then show $Stars\ [] = v2$ **by** *cases* (*auto simp add*: *Posix1*)
next
case (*Posix-From2 vs r n v2*)
then show $Stars\ vs = v2$
apply(*erule-tac Posix-elims*)
 apply(*auto*)
apply(*rule List-eq-zipI*)
 apply(*auto*)
  apply(*meson in-set-zipE*)
 apply (*simp add*: *Posix1*(*2*))
using *Posix1*(*2*) **by** *blast*

next
  case (*Posix-From1 s1 r v s2 n vs*)
  have $(s1\ @\ s2) \in From\ r\ n \rightarrow v2$
    $s1 \in r \rightarrow v\ s2 \in From\ r\ (n - 1) \rightarrow Stars\ vs\ flat\ v \neq []\ 0 < n$
    $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge s1\ @\ s_3 \in lang\ r \wedge s_4 \in lang\ (From\ r$
$(n - 1)))$ **by** *fact+*
  then obtain $v'\ vs'$ **where** $v2 = Stars\ (v' \# vs')\ s1 \in r \rightarrow v'\ s2 \in (From\ r\ (n$
$- 1)) \rightarrow (Stars\ vs')$
  apply(*cases*) **apply** (*auto simp add*: *append-eq-append-conv2*)
   using *Posix1*(*1*) *Posix1*(*2*) **apply** *blast*
    apply(*case-tac n*)
     apply(*simp*)
    apply(*simp*)
    apply (*smt* (*verit*, *ccfv-threshold*) *Posix1*(*1*) *UN-E append-eq-append-conv2*
lang.simps(*9*))
  by (*metis One-nat-def Posix1*(*1*) *Posix-From1.hyps*(*7*) *append-Nil2 append-self-conv2*)
  moreover
  have *IHs*: $\bigwedge v2.\ s1 \in r \rightarrow v2 \Longrightarrow v = v2$
    $\bigwedge v2.\ s2 \in From\ r\ (n - 1) \rightarrow v2 \Longrightarrow Stars\ vs = v2$ **by** *fact+*
  ultimately show $Stars\ (v \# vs) = v2$ **by** *auto*
next
  case (*Posix-From3 s1 r v s2 vs*)
  have $(s1\ @\ s2) \in From\ r\ 0 \rightarrow v2$
    $s1 \in r \rightarrow v\ s2 \in Star\ r \rightarrow Stars\ vs\ flat\ v \neq []$
    $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge s1\ @\ s_3 \in lang\ r \wedge s_4 \in lang\ (Star\ r))$
by *fact+*
  then obtain $v'\ vs'$ **where** $v2 = Stars\ (v' \# vs')\ s1 \in r \rightarrow v'\ s2 \in (Star\ r) \rightarrow$
$(Stars\ vs')$
  apply(*cases*) **apply** (*auto simp add*: *append-eq-append-conv2*)
   using *Posix1*(*2*) **apply** *fastforce*
   using *Posix1*(*1*) **apply** *fastforce*
   by (*metis Posix1*(*1*) *Posix-From3.hyps*(*6*) *append.right-neutral append-Nil*)
  moreover
  have *IHs*: $\bigwedge v2.\ s1 \in r \rightarrow v2 \Longrightarrow v = v2$
    $\bigwedge v2.\ s2 \in Star\ r \rightarrow v2 \Longrightarrow Stars\ vs = v2$ **by** *fact+*
  ultimately show $Stars\ (v \# vs) = v2$ **by** *auto*
next
  case (*Posix-Rec s r v l v2*)
  then show $Recv\ l\ v = v2$ **by** (*metis Posix-elims*(*10*))
next
  case (*Posix-Cset c cs v2*)
  have $[c] \in Charset\ cs \rightarrow v2$ **by** *fact*
  then show $Atm\ c = v2$ **by** *cases auto*
qed

lemma *Posix-injval*:
  assumes $s \in (deriv\ c\ r) \rightarrow v$
  shows $(c \# s) \in r \rightarrow (injval\ r\ c\ v)$

```
using assms
proof(induct r arbitrary: s v rule: rexp.induct)
  case Zero
  have s ∈ deriv 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 ∈ deriv 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 (Atom d)
  consider (eq) c = d | (ineq) c ≠ d by blast
  then show (c # s) ∈ (Atom d) → (injval (Atom d) c v)
  proof (cases)
    case eq
    have s ∈ deriv c (Atom 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) ∈ Atom d → injval (Atom d) c v using eq eqs
    by (auto intro: Posix.intros)
  next
    case ineq
    have s ∈ deriv c (Atom d) → v by fact
    then have s ∈ Zero → v using ineq by simp
    then have False by cases
    then show (c # s) ∈ Atom d → injval (Atom d) c v by simp
  qed
next
  case (Plus r1 r2)
  have IH1: ⋀s v. s ∈ deriv c r1 → v ⟹ (c # s) ∈ r1 → injval r1 c v by fact
  have IH2: ⋀s v. s ∈ deriv c r2 → v ⟹ (c # s) ∈ r2 → injval r2 c v by fact
  have s ∈ deriv c (Plus r1 r2) → v by fact
  then have s ∈ Plus (deriv c r1) (deriv c r2) → v by simp
  then consider (left) v' where v = Left v' s ∈ deriv c r1 → v'
            | (right) v' where v = Right v' s ∉ lang (deriv c r1) s ∈ deriv c r2 →
v'
             by cases auto
  then show (c # s) ∈ Plus r1 r2 → injval (Plus r1 r2) c v
  proof (cases)
    case left
    have s ∈ deriv c r1 → v' by fact
    then have (c # s) ∈ r1 → injval r1 c v' using IH1 by simp
     then have (c # s) ∈ Plus r1 r2 → injval (Plus r1 r2) c (Left v') by (auto
intro: Posix.intros)
    then show (c # s) ∈ Plus r1 r2 → injval (Plus r1 r2) c v using left by simp
```

next
case *right*
have $s \notin lang\ (deriv\ c\ r1)$ **by** *fact*
then have $c \# s \notin lang\ r1$ **by** (*simp add*: *lang-deriv Deriv-def*)
moreover
have $s \in deriv\ c\ r2 \to v'$ **by** *fact*
then have $(c \# s) \in r2 \to injval\ r2\ c\ v'$ **using** *IH2* **by** *simp*
ultimately have $(c \# s) \in Plus\ r1\ r2 \to injval\ (Plus\ r1\ r2)\ c\ (Right\ v')$
by (*auto intro*: *Posix.intros*)
then show $(c \# s) \in Plus\ r1\ r2 \to injval\ (Plus\ r1\ r2)\ c\ v$ **using** *right* **by** *simp*
qed
next
case $(Times\ r1\ r2)$
have *IH1*: $\bigwedge s\ v.\ s \in deriv\ c\ r1 \to v \Longrightarrow (c \# s) \in r1 \to injval\ r1\ c\ v$ **by** *fact*
have *IH2*: $\bigwedge s\ v.\ s \in deriv\ c\ r2 \to v \Longrightarrow (c \# s) \in r2 \to injval\ r2\ c\ v$ **by** *fact*
have $s \in deriv\ c\ (Times\ r1\ r2) \to v$ **by** *fact*
then consider
(*left-nullable*) *v1 v2 s1 s2* **where**
$v = Left\ (Seq\ v1\ v2)\ \ s = s1\ @\ s2$
$s1 \in deriv\ c\ r1 \to v1\ s2 \in r2 \to v2\ nullable\ r1$
$\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ s1\ @\ s_3 \in lang\ (deriv\ c\ r1)\ \wedge\ s_4 \in lang\ r2)$
| (*right-nullable*) *v1 s1 s2* **where**
$v = Right\ v1\ s = s1\ @\ s2$
$s \in deriv\ c\ r2 \to v1\ nullable\ r1\ s1\ @\ s2 \notin lang\ (Times\ (deriv\ c\ r1)\ r2)$
| (*not-nullable*) *v1 v2 s1 s2* **where**
$v = Seq\ v1\ v2\ s = s1\ @\ s2$
$s1 \in deriv\ c\ r1 \to v1\ s2 \in r2 \to v2\ \neg nullable\ r1$
$\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ s1\ @\ s_3 \in lang\ (deriv\ c\ r1)\ \wedge\ s_4 \in lang\ r2)$
by (*force split*: *if-splits elim*!: *Posix-elims simp add*: *lang-deriv Deriv-def*)
then show $(c \# s) \in Times\ r1\ r2 \to injval\ (Times\ r1\ r2)\ c\ v$
proof (*cases*)
case *left-nullable*
have $s1 \in deriv\ c\ r1 \to v1$ **by** *fact*
then have $(c \# s1) \in r1 \to injval\ r1\ c\ v1$ **using** *IH1* **by** *simp*
moreover
have $\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ s1\ @\ s_3 \in lang\ (deriv\ c\ r1)\ \wedge\ s_4 \in lang\ r2)$ **by** *fact*
then have $\neg\ (\exists s_3\ s_4.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = s2\ \wedge\ (c \# s1)\ @\ s_3 \in lang\ r1\ \wedge\ s_4 \in lang\ r2)$
by (*simp add*: *lang-deriv Deriv-def*)
ultimately have $((c \# s1)\ @\ s2) \in Times\ r1\ r2 \to Seq\ (injval\ r1\ c\ v1)\ v2$ **using** *left-nullable* **by** (*rule-tac Posix.intros*)
then show $(c \# s) \in Times\ r1\ r2 \to injval\ (Times\ r1\ r2)\ c\ v$ **using** *left-nullable* **by** *simp*
next
case *right-nullable*

have *nullable r1* **by** *fact*
then have $[] \in r1 \rightarrow (mkeps\ r1)$ **by** (*rule Posix-mkeps*)
moreover
have $s \in deriv\ c\ r2 \rightarrow v1$ **by** *fact*
then have $(c \# s) \in r2 \rightarrow (injval\ r2\ c\ v1)$ **using** *IH2* **by** *simp*
moreover
have $s1 @ s2 \notin lang\ (Times\ (deriv\ c\ r1)\ r2)$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = c \# s \wedge [] @ s_3 \in lang\ r1 \wedge s_4 \in lang\ r2)$
using *right-nullable*
apply (*auto simp add*: *lang-deriv Deriv-def append-eq-Cons-conv*)
by (*metis concI mem-Collect-eq*)
ultimately have $([] @ (c \# s)) \in Times\ r1\ r2 \rightarrow Seq\ (mkeps\ r1)\ (injval\ r2\ c\ v1)$
by(*rule Posix.intros*)
then show $(c \# s) \in Times\ r1\ r2 \rightarrow injval\ (Times\ r1\ r2)\ c\ v$ **using** *right-nullable* **by** *simp*
next
case *not-nullable*
have $s1 \in deriv\ c\ r1 \rightarrow v1$ **by** *fact*
then have $(c \# s1) \in r1 \rightarrow injval\ r1\ c\ v1$ **using** *IH1* **by** *simp*
moreover
have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r1) \wedge s_4 \in lang\ r2)$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r1 \wedge s_4 \in lang\ r2)$ **by** (*simp add*: *lang-deriv Deriv-def*)
ultimately have $((c \# s1) @ s2) \in Times\ r1\ r2 \rightarrow Seq\ (injval\ r1\ c\ v1)\ v2$ **using** *not-nullable*
by (*rule-tac Posix.intros*) (*simp-all*)
then show $(c \# s) \in Times\ r1\ r2 \rightarrow injval\ (Times\ r1\ r2)\ c\ v$ **using** *not-nullable* **by** *simp*
qed
next
case (*Star r*)
have *IH*: $\bigwedge s\ v.\ s \in deriv\ c\ r \rightarrow v \Longrightarrow (c \# s) \in r \rightarrow injval\ r\ c\ v$ **by** *fact*
have $s \in deriv\ c\ (Star\ r) \rightarrow v$ **by** *fact*
then consider
(*cons*) *v1 vs s1 s2* **where**
$v = Seq\ v1\ (Stars\ vs)\ s = s1 @ s2$
$s1 \in deriv\ c\ r \rightarrow v1\ s2 \in (Star\ r) \rightarrow (Stars\ vs)$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Star\ r))$
apply(*auto elim*!: *Posix-elims*($1-5$) *simp add*: *lang-deriv Deriv-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 $s1 \in deriv\ c\ r \rightarrow v1$ **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 $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Star\ r))$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r \wedge s_4 \in lang\ (Star\ r))$
by (*simp add*: *lang-deriv Deriv-def*)
ultimately
have $((c \# s1) @ s2) \in Star\ r \rightarrow Stars\ (injval\ r\ c\ v1 \# vs)$ **by** (*rule Posix.intros*)
then show $(c \# s) \in Star\ r \rightarrow injval\ (Star\ r)\ c\ v$ **using** *cons* **by**(*simp*)
qed
next
case $(NTimes\ r\ n)$
have *IH*: $\bigwedge s\ v.\ s \in deriv\ c\ r \rightarrow v \Longrightarrow (c \# s) \in r \rightarrow injval\ r\ c\ v$ **by** *fact*
have $s \in deriv\ c\ (NTimes\ r\ n) \rightarrow v$ **by** *fact*
then consider
(*cons*) *v1 vs s1 s2* **where**
$v = Seq\ v1\ (Stars\ vs)\ s = s1 @ s2$
$s1 \in deriv\ c\ r \rightarrow v1\ s2 \in (NTimes\ r\ (n - 1)) \rightarrow (Stars\ vs)\ 0 < n$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (NTimes\ r\ (n - 1)))$
apply(*auto elim*: *Posix-elims simp add*: *lang-derivs Deriv-def intro*: *Posix.intros split*: *if-splits*)
apply(*erule Posix-elims*)
apply(*simp*)
apply(*subgoal-tac* $\exists vss.\ v2 = Stars\ vss$)
apply(*clarify*)
apply(*drule-tac x=vss* **in** *meta-spec*)
apply(*drule-tac x=s1* **in** *meta-spec*)
apply(*drule-tac x=s2* **in** *meta-spec*)
apply(*simp add*: *lang-derivs Deriv-def*)
apply(*erule Posix-elims*)
apply(*auto*)
done
then show $(c \# s) \in (NTimes\ r\ n) \rightarrow injval\ (NTimes\ r\ n)\ c\ v$
proof (*cases*)
case *cons*
have $s1 \in deriv\ c\ r \rightarrow v1$ **by** *fact*
then have $(c \# s1) \in r \rightarrow injval\ r\ c\ v1$ **using** *IH* **by** *simp*

moreover
have $s2 \in (NTimes\ r\ (n - 1)) \to Stars\ vs$ **by** *fact*
moreover
have $(c \# s1) \in r \to injval\ r\ c\ v1$ **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 $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (NTimes\ r\ (n - 1)))$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r \wedge s_4 \in lang\ (NTimes\ r\ (n - 1)))$
by (*simp add*: *lang-deriv Deriv-def*)
ultimately
have $((c \# s1) @ s2) \in NTimes\ r\ n \to Stars\ (injval\ r\ c\ v1 \# vs)$
by (*metis One-nat-def Posix-NTimes1 Suc-pred add.commute cons*(*5*) *plus-1-eq-Suc*)
then show $(c \# s) \in NTimes\ r\ n \to injval\ (NTimes\ r\ n)\ c\ v$ **using** *cons* **by**(*simp*)
qed
next
case (*Upto r n*)
have *IH*: $\bigwedge s\ v.\ s \in deriv\ c\ r \to v \Longrightarrow (c \# s) \in r \to injval\ r\ c\ v$ **by** *fact*
have $s \in deriv\ c\ (Upto\ r\ n) \to v$ **by** *fact*
then consider
(*cons*) *v1 vs s1 s2* **where**
$v = Seq\ v1\ (Stars\ vs)\ s = s1 @ s2$
$s1 \in deriv\ c\ r \to v1\ s2 \in (Upto\ r\ (n - 1)) \to (Stars\ vs)$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Upto\ r\ (n - 1)))$
apply(*auto elim*!: *Posix-elims simp add*: *lang-deriv Deriv-def intro*: *Posix.intros*)

apply(*case-tac n*)
apply(*auto*)
using *Posix-elims*(*1*) **apply** *blast*
apply(*erule-tac Posix-elims*)
apply(*auto*)
by (*metis Posix1a Prf-elims*(*8*) *UN-E cons diff-Suc-1 lang.simps*(*8*))
then show $(c \# s) \in Upto\ r\ n \to injval\ (Upto\ r\ n)\ c\ v$
proof (*cases*)
case *cons*
have $s1 \in deriv\ c\ r \to v1$ **by** *fact*
then have $(c \# s1) \in r \to injval\ r\ c\ v1$ **using** *IH* **by** *simp*
moreover
have $s2 \in Upto\ r\ (n - 1) \to Stars\ vs$ **by** *fact*
moreover
have $(c \# s1) \in r \to injval\ r\ c\ v1$ **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 $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Upto\ r\ (n - 1)))$ **by** *fact*
then have $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r \wedge s_4 \in lang\ (Upto\ r\ (n - 1)))$
by (*simp add*: *lang-deriv Deriv-def*)
ultimately
have $((c \# s1) @ s2) \in Upto\ r\ n \rightarrow Stars\ (injval\ r\ c\ v1 \# vs)$
by (*metis One-nat-def Posix-Upto1 Posix-elims*(*1*) *Suc-pred Upto.prems add.commute bot-nat-0.not-eq-extremum deriv.simps*(*8*) *plus-1-eq-Suc*)
then show $(c \# s) \in Upto\ r\ n \rightarrow injval\ (Upto\ r\ n)\ c\ v$ **using** *cons* **by**(*simp*)
qed
next
case (*From r n*)
have *IH*: $\bigwedge s\ v.\ s \in deriv\ c\ r \rightarrow v \Longrightarrow (c \# s) \in r \rightarrow injval\ r\ c\ v$ **by** *fact*
have $s \in deriv\ c\ (From\ r\ n) \rightarrow v$ **by** *fact*
then consider
(*cons*) *v1 vs s1 s2* **where**
$v = Seq\ v1\ (Stars\ vs)\ \ s = s1 @ s2$
$s1 \in deriv\ c\ r \rightarrow v1\ \ s2 \in (From\ r\ (n - 1)) \rightarrow (Stars\ vs)\ \ 0 < n$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (From\ r\ (n - 1)))$
| (*null*) *v1 vs s1 s2* **where**
$v = Seq\ v1\ (Stars\ vs)\ \ s = s1 @ s2\ \ s2 \in (Star\ r) \rightarrow (Stars\ vs)$
$s1 \in deriv\ c\ r \rightarrow v1\ \ n = 0$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Star\ r))$
apply(*auto elim*: *Posix-elims simp add*: *lang-deriv Deriv-def intro*: *Posix.intros split*: *if-splits*)
apply(*erule Posix-elims*)
apply(*auto*)
apply(*auto elim*: *Posix-elims simp add*: *lang-deriv Deriv-def intro*: *Posix.intros split*: *if-splits*)
apply(*metis Posix1a Prf-elims*(*6*))
apply(*erule Posix-elims*)
apply(*auto*)
apply(*erule Posix-elims*(*9*))
apply (*metis* (*no-types*, *lifting*) *Nil-is-append-conv Posix-From2*)
apply (*simp add*: *Posix-From1 that*(*1*))
by (*simp add*: *Posix-From3 that*(*1*))
then show $(c \# s) \in (From\ r\ n) \rightarrow injval\ (From\ r\ n)\ c\ v$
proof (*cases*)
case *cons*
have $s1 \in deriv\ c\ r \rightarrow v1$ **by** *fact*
then have $(c \# s1) \in r \rightarrow injval\ r\ c\ v1$ **using** *IH* **by** *simp*
moreover
have $s2 \in (From\ r\ (n - 1)) \rightarrow Stars\ vs$ **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 $\neg$ ($\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (From\ r\ (n - 1))$) **by** *fact*
then have $\neg$ ($\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r \wedge s_4 \in lang\ (From\ r\ (n - 1))$)
by (*simp add*: *lang-deriv Deriv-def*)
ultimately
have (($c \# s1$) @ $s2$) $\in$ *From r n* $\rightarrow$ *Stars* (*injval r c v1* # *vs*)
by (*meson Posix-From1 cons*(*5*))
then show ($c \# s$) $\in$ *From r n* $\rightarrow$ *injval* (*From r n*) *c v* **using** *cons* **by**(*simp*)
next
case *null*
have *s1* $\in$ *deriv 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 $\neg$ ($\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ (deriv\ c\ r) \wedge s_4 \in lang\ (Star\ r)$) **by** *fact*
then have $\neg$ ($\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge (c \# s1) @ s_3 \in lang\ r \wedge s_4 \in lang\ (Star\ r)$)
by (*simp add*: *lang-deriv Deriv-def*)
ultimately
have (($c \# s1$) @ $s2$) $\in$ *From r 0* $\rightarrow$ *Stars* (*injval r c v1* # *vs*)
by (*metis Posix-From3*)
then show ($c \# s$) $\in$ *From r n* $\rightarrow$ *injval* (*From r n*) *c v* **using** *null* **by** (*simp*)
qed
next
case (*Rec l r*)
then show ($c \# s$) $\in$ *Rec l r* $\rightarrow$ *injval* (*Rec l r*) *c v*
by (*simp add*: *Posix-Rec*)
next
case (*Charset cs*)
consider (*eq*) $c \in cs$ | (*ineq*) $c \notin cs$ **by** *blast*
then show ($c \# s$) $\in$ (*Charset cs*) $\rightarrow$ (*injval* (*Charset cs*) *c v*)
proof (*cases*)
case *eq*
have $s \in$ *deriv c* (*Charset cs*) $\rightarrow v$ **by** *fact*
then have $s \in$ *One* $\rightarrow v$ **using** *eq* **by** *simp*
then have *eqs*: $s = [] \wedge v =$ *Void* **by** *cases simp*
show ($c \# s$) $\in$ *Charset cs* $\rightarrow$ *injval* (*Charset cs*) *c v* **using** *eq eqs*
by (*auto intro*: *Posix.intros*)
next

case *ineq*
have $s \in deriv\ c\ (Charset\ cs) \to v$ **by** *fact*
then have $s \in Zero \to v$ **using** *ineq* **by** *simp*
then have *False* **by** *cases*
then show $(c\ \#\ s) \in Charset\ cs \to injval\ (Charset\ cs)\ c\ v$ **by** *simp*
qed
qed

23 The Lexer by Sulzmann and Lu

fun
lexer :: $'a\ rexp \Rightarrow 'a\ list \Rightarrow ('a\ val)\ option$
where
$lexer\ r\ [] = (if\ nullable\ r\ then\ Some(mkeps\ r)\ else\ None)$
$|\ lexer\ r\ (c\#s) = (case\ (lexer\ (deriv\ c\ r)\ s)\ of$
$None \Rightarrow None$
$|\ Some(v) \Rightarrow Some(injval\ r\ c\ v))$

lemma *lexer-correct-None*:
shows $s \notin lang\ r \longleftrightarrow lexer\ r\ s = None$
apply(*induct s arbitrary*: *r*)
apply(*simp add*: *nullable-iff*)
apply(*drule-tac x=deriv a r* **in** *meta-spec*)
apply(*auto simp add*: *lang-deriv Deriv-def*)
done

lemma *lexer-correct-Some*:
shows $s \in lang\ r \longleftrightarrow (\exists\, v.\ lexer\ r\ s = Some(v) \wedge s \in r \to v)$
apply(*induct s arbitrary*: *r*)
apply(*auto simp add*: *Posix-mkeps nullable-iff*)[*1*]
apply(*drule-tac x=deriv a r* **in** *meta-spec*)
apply(*simp add*: *lang-deriv Deriv-def*)
apply(*rule iffI*)
apply(*auto intro*: *Posix-injval simp add*: *Posix1*(*1*))
done

lemma *lexer-correctness*:
shows $(lexer\ r\ s = Some\ v) \longleftrightarrow s \in r \to v$
and $(lexer\ r\ s = None) \longleftrightarrow \neg(\exists\, v.\ s \in r \to v)$
apply(*auto*)
using *lexer-correct-None lexer-correct-Some* **apply** *fastforce*
using *Posix1*(*1*) *Posix-determ lexer-correct-Some* **apply** *blast*
using *Posix1*(*1*) *lexer-correct-None* **apply** *blast*
using *lexer-correct-None lexer-correct-Some* **by** *blast*

end

theory *LexicalVals3*
  imports *Lexer3 HOL−Library.Sublist*
begin

24 Sets of Lexical Values

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

definition
  $LV :: 'a\ rexp \Rightarrow 'a\ list \Rightarrow ('a\ val)\ set$
where $LV\ r\ s \equiv \{v.\ \vdash v : r \wedge flat\ v = s\}$

lemma *LV-simps*:
  shows $LV\ Zero\ s = \{\}$
  and $LV\ One\ s = (if\ s = []\ then\ \{Void\}\ else\ \{\})$
  and $LV\ (Atom\ c)\ s = (if\ s = [c]\ then\ \{Atm\ c\}\ else\ \{\})$
  and $LV\ (Plus\ r1\ r2)\ s = Left\ `\ LV\ r1\ s \cup Right\ `\ LV\ r2\ s$
  and $LV\ (NTimes\ r\ 0)\ s = (if\ s = []\ then\ \{Stars\ []\}\ else\ \{\})$
  and $LV\ (Rec\ l\ r)\ s = \{Recv\ l\ v \mid v.\ v \in LV\ r\ s\}$
  and $LV\ (Charset\ cs)\ s = (if\ length\ s = 1 \wedge (hd\ s) \in cs\ then\ \{Atm\ (hd\ s)\}\ else$
$\{\})$
unfolding *LV-def*
  apply(*auto intro*: *Prf.intros elim*: *Prf.cases*)
  apply(*simp add*: *Prf-NTimes-empty*)
  by (*metis Suc-length-conv length-0-conv list.sel*(*1*))

abbreviation
  $Prefixes\ s \equiv \{s'.\ prefix\ s'\ s\}$

abbreviation
  $Suffixes\ s \equiv \{s'.\ suffix\ s'\ s\}$

abbreviation
  $SSuffixes\ s \equiv \{s'.\ strict\text{-}suffix\ s'\ s\}$

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 $SSuffixes\ s \subseteq Suffixes\ s$
   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::'a 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::'a val, 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

definition
  Stars-Cons V Vs ≡ {Stars (v # vs) | v vs. v ∈ V ∧ Stars vs ∈ Vs}

definition
  Stars-Append Vs1 Vs2 ≡ {Stars (vs1 @ vs2) | vs1 vs2. Stars vs1 ∈ Vs1 ∧ Stars
vs2 ∈ Vs2}

fun Stars-Pow :: ('a val) set ⇒ nat ⇒ ('a val) set
where
  Stars-Pow Vs 0 = {Stars []}
| Stars-Pow Vs (Suc n) = Stars-Cons Vs (Stars-Pow Vs n)

lemma finite-Stars-Cons:
  assumes finite V finite Vs
  shows finite (Stars-Cons V Vs)
  using assms
proof −
  from assms(2) have finite (Stars −‘ Vs)
    by(simp add: finite-vimageI inj-on-def)
  with assms(1) have finite (V × (Stars −‘ Vs))
    by(simp)
  then have finite ((λ(v, vs). Stars (v # vs)) ‘ (V × (Stars −‘ Vs)))
    by simp
  moreover have Stars-Cons V Vs = (λ(v, vs). Stars (v # vs)) ‘ (V × (Stars −‘
Vs))
    unfolding Stars-Cons-def by auto
  ultimately show finite (Stars-Cons V Vs)
    by simp
qed

lemma finite-Stars-Append:
  assumes finite Vs1 finite Vs2
  shows finite (Stars-Append Vs1 Vs2)
  using assms
proof −
  define UVs1 where UVs1 ≡ Stars −‘ Vs1
  define UVs2 where UVs2 ≡ Stars −‘ Vs2
  from assms have finite UVs1 finite UVs2
    unfolding UVs1-def UVs2-def
```

by(*simp-all add*: *finite-vimageI inj-on-def*)
then have *finite* $((\lambda(vs1, vs2).\ Stars\ (vs1\ @\ vs2))$ ‘ $(UVs1 \times UVs2))$
by *simp*
moreover
have *Stars-Append Vs1 Vs2* $= (\lambda(vs1, vs2).\ Stars\ (vs1\ @\ vs2))$ ‘ $(UVs1 \times UVs2)$
unfolding *Stars-Append-def UVs1-def UVs2-def* **by** *auto*
ultimately show *finite* (*Stars-Append Vs1 Vs2*)
by *simp*
qed

lemma *finite-Stars-Pow*:
assumes *finite Vs*
shows *finite* (*Stars-Pow Vs n*)
by (*induct n*) (*simp-all add*: *finite-Stars-Cons assms*)

lemma *LV-NTimes-5*:
$LV\ (NTimes\ r\ n)\ s \subseteq Stars\text{-}Append\ (LV\ (Star\ r)\ s)\ (\bigcup i \leq n.\ LV\ (NTimes\ r\ i)\ [])$
apply(*auto simp add*: *LV-def*)
apply(*auto elim*!: *Prf-elims*)
apply(*auto simp add*: *Stars-Append-def*)
apply(*rule-tac x=vs1* **in** *exI*)
apply(*rule-tac x=vs2* **in** *exI*)
apply(*auto*)
using *Prf.intros*(*6*) **apply**(*auto*)
apply(*rule-tac x=length vs2* **in** *bexI*)
thm *Prf.intros*
apply(*subst append.simps*(*1*)[*symmetric*])
apply(*rule Prf.intros*)
apply(*auto*)[*1*]
apply(*auto*)[*1*]
apply(*simp*)
apply(*simp*)
done

lemma *LV-NTIMES-3*:
shows $LV\ (NTimes\ r\ (Suc\ n))\ [] =$
$(\lambda(v, vs).\ Stars\ (v\#vs))$ ‘ $(LV\ r\ [] \times (Stars -\text{‘}\ (LV\ (NTimes\ r\ n)\ [])))$
unfolding *LV-def*
apply(*auto elim*!: *Prf-elims simp add*: *image-def*)
apply(*case-tac vs1*)
apply(*auto*)
apply(*case-tac vs2*)
apply(*auto*)
apply(*subst append.simps*(*1*)[*symmetric*])
apply(*rule Prf.intros*)
apply(*auto*)
apply(*subst append.simps*(*1*)[*symmetric*])

apply(*rule Prf.intros*)
apply(*auto*)
  done

lemma *finite-NTimes-empty*:
  assumes $\bigwedge$*s. finite* (*LV r s*)
  shows *finite* (*LV* (*NTimes r n*) [])
  using *assms*
  apply(*induct n*)
   apply(*auto simp add*: *LV-simps*)
  apply(*subst LV-NTIMES-3*)
  apply(*rule finite-imageI*)
  apply(*rule finite-cartesian-product*)
  using *assms* **apply** *simp*
  apply(*rule finite-vimageI*)
  apply(*simp*)
  apply(*simp add*: *inj-on-def*)
  done

lemma *LV-From-5*:
  shows *LV* (*From r n*) *s* ⊆ *Stars-Append* (*LV* (*Star r*) *s*) ($\bigcup$*i*≤*n*. *LV* (*From r i*) [])
apply(*auto simp add*: *LV-def*)
apply(*auto elim*!: *Prf-elims*)
apply(*auto simp add*: *Stars-Append-def*)
apply(*rule-tac x=vs1* **in** *exI*)
apply(*rule-tac x=vs2* **in** *exI*)
apply(*auto*)
    using *Prf.intros*(*6*) **apply**(*auto*)
      apply(*rule-tac x=length vs2* **in** *bexI*)
    thm *Prf.intros*
      apply(*subst append.simps*(*1*)[*symmetric*])
    apply(*rule Prf.intros*)
      apply(*auto*)[*1*]
      apply(*auto*)[*1*]
     apply(*simp*)
     apply(*simp*)
      apply(*rule-tac x=vs* **in** *exI*)
    apply(*rule-tac x=*[] **in** *exI*)
    apply(*auto*)
   by (*metis Prf.intros*(*9*) *append-Nil atMost-iff empty-iff le-imp-less-Suc less-antisym list.set*(*1*) *nth-mem zero-le*)

lemma *LV-FROMNTIMES-3*:
  shows *LV* (*From r* (*Suc n*)) [] =
    (λ(*v*,*vs*). *Stars* (*v*#*vs*)) ‘ (*LV r* [] × (*Stars* −‘ (*LV* (*From r n*) [])))
unfolding *LV-def*
apply(*auto elim*!: *Prf-elims simp add*: *image-def*)
apply(*case-tac vs1*)

```
apply(auto)
apply(case-tac vs2)
apply(auto)
apply(subst append.simps(1)[symmetric])
apply(rule Prf.intros)
    apply(auto)
 apply (metis le-imp-less-Suc length-greater-0-conv less-antisym list.exhaust list.set-intros(1)
not-less-eq zero-le)
  prefer 2
  using nth-mem apply blast
  apply(case-tac vs1)
   apply (smt (verit, del-insts) Prf.intros(9) append.left-neutral length-Suc-conv
length-append
      set-ConsD)
  apply(auto)
done

lemma LV-From-empty:
 LV (From r n) [] = Stars-Pow (LV r []) n
  apply(induct n)
   apply(simp add: LV-def)
   apply(auto elim: Prf-elims simp add: image-def)[1]
   prefer 2
    apply(subst append.simps[symmetric])
    apply(rule Prf.intros)
      apply(simp-all)
   apply(erule Prf-elims)
    apply(case-tac vs1)
     apply(simp)
    apply(simp)
   apply(case-tac x)
    apply(simp-all)
    apply(simp add: LV-FROMNTIMES-3 image-def Stars-Cons-def)
  apply blast
 done

lemma finite-From-empty:
  assumes ∀ s. finite (LV r s)
  shows finite (LV (From r n) s)
  apply(rule finite-subset)
   apply(rule LV-From-5)
  apply(rule finite-Stars-Append)
    apply(rule LV-STAR-finite)
   apply(rule assms)
  apply(rule finite-UN-I)
   apply(auto)
  by (simp add: assms finite-Stars-Pow LV-From-empty)
```

lemma *subseteq-Upto-Star*:
  shows *LV* (*Upto r n*) *s* $\subseteq$ *LV* (*Star r*) *s*
  apply(*auto simp add*: *LV-def*)
  by (*metis Prf.intros*(*6*) *Prf-elims*(*8*))

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 (*Atom c s*)
  show *finite* (*LV* (*Atom c*) *s*) **by** (*simp add*: *LV-simps*)
next
  case (*Plus r1 r2 s*)
  then show *finite* (*LV* (*Plus r1 r2*) *s*) **by** (*simp add*: *LV-simps*)
next
  case (*Times r1 r2 s*)
  define *f* **where** $f \equiv \lambda(v1::'a\ val,\ v2).\ Seq\ v1\ v2$
  define *S1* **where** $S1 \equiv \bigcup s' \in Prefixes\ s.\ LV\ r1\ s'$
  define *S2* **where** $S2 \equiv \bigcup s' \in Suffixes\ s.\ LV\ r2\ s'$
  have *IHs*: $\bigwedge s.$ *finite* (*LV r1 s*) $\bigwedge s.$ *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* (*Times r1 r2*) *s* $\subseteq$ *f* ‘ (*S1* $\times$ *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* (*Times 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*)
next
  case (*NTimes r n s*)
  have $\bigwedge s.$ *finite* (*LV r s*) **by** *fact*
  then have *finite* (*Stars-Append* (*LV* (*Star r*) *s*) ($\bigcup i \leq n.$ *LV* (*NTimes r i*) []))
    apply(*rule-tac finite-Stars-Append*)
     apply (*simp add*: *LV-STAR-finite*)
    using *finite-NTimes-empty* **by** *blast*
  then show *finite* (*LV* (*NTimes r n*) *s*)
    by (*metis LV-NTimes-5 finite-subset*)

next
  case (*Upto r n s*)
  then have *finite* (*LV* (*Star r*) *s*) **by** (*simp add*: *LV-STAR-finite*)
  moreover
  have *LV* (*Upto r n*) *s* ⊆ *LV* (*Star r*) *s*
    by (*meson subseteq-Upto-Star*)
  ultimately show *finite* (*LV* (*Upto r n*) *s*)
    using *rev-finite-subset* **by** *blast*
next
  case (*From r n*)
  then show *finite* (*LV* (*From r n*) *s*)
    by (*simp add*: *finite-From-empty*)
next
  case (*Rec l r*)
  have ⋀*s*. *finite* (*LV r s*) **by** *fact*
  then show *finite* (*LV* (*Rec l r*) *s*)
    by(*simp add*: *LV-simps*)
next
  case (*Charset cs s*)
  show *finite* (*LV* (*Charset cs*) *s*) **by** (*simp add*: *LV-simps*)
qed

Our POSIX values are lexical values.

lemma *Posix-LV*:
  assumes $s \in r \rightarrow v$
  shows $v \in LV\ r\ s$
  using *assms* **unfolding** *LV-def*
  apply(*induct rule*: *Posix.induct*)
  using *Prf.intros*(*4*) *flat.simps*(*1*) **apply** *blast*
  apply (*simp add*: *Prf.intros*(*5*))
  apply (*simp add*: *Prf.intros*(*2*))
  apply (*simp add*: *Prf.intros*(*3*))
  apply (*simp add*: *Prf.intros*(*1*))
  apply (*smt* (*verit*, *best*) *CollectI Posix1*(*2*) *Posix1a Posix-Star1*)
  apply (*simp add*: *Prf.intros*(*6*))
  apply (*smt* (*verit*, *best*) *Posix1*(*2*) *Posix1a Posix-NTimes1 mem-Collect-eq*)
  using *Posix1a Posix-NTimes2* **apply** *fastforce*
 apply (*smt* (*verit*, *ccfv-threshold*) *Posix1*(*2*) *Posix1a Posix-Upto1 mem-Collect-eq*)
  using *Posix1a Posix-Upto2* **apply** *fastforce*
  using *Posix1a Posix-From2* **apply** *fastforce*
  apply (*smt* (*verit*, *best*) *Posix1*(*2*) *Posix1a Posix-From1 mem-Collect-eq*)
  apply (*smt* (*verit*, *best*) *Posix1a Posix-From3 flat.simps*(*7*) *mem-Collect-eq*)
  apply(*simp add*: *Prf.intros*(*11*))
  by (*simp add*: *Prf.intros*(*12*))

lemma *Posix-Prf*:
  assumes $s \in r \rightarrow v$
  shows $\vdash v : r$

using *assms Posix-LV LV-def*
by *blast*

end

theory *Simplifying3*
imports *Lexer3*
begin

25 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-Plus* **where**
$F\text{-}Plus\ f_1\ f_2\ (Right\ v) = Right\ (f_2\ v)$
$|\ F\text{-}Plus\ f_1\ f_2\ (Left\ v) = Left\ (f_1\ v)$
$|\ F\text{-}Plus\ f1\ f2\ v = v$

fun *F-Times1* **where**
$F\text{-}Times1\ f_1\ f_2\ v = Seq\ (f_1\ Void)\ (f_2\ v)$

fun *F-Times2* **where**
$F\text{-}Times2\ f_1\ f_2\ v = Seq\ (f_1\ v)\ (f_2\ Void)$

fun *F-Times* **where**
$F\text{-}Times\ f_1\ f_2\ (Seq\ v_1\ v_2) = Seq\ (f_1\ v_1)\ (f_2\ v_2)$
$|\ F\text{-}Times\ f1\ f2\ v = v$

fun *simp-Plus* **where**
$simp\text{-}Plus\ (Zero,\ f_1)\ (r_2,\ f_2) = (r_2,\ F\text{-}RIGHT\ f_2)$
$|\ simp\text{-}Plus\ (r_1,\ f_1)\ (Zero,\ f_2) = (r_1,\ F\text{-}LEFT\ f_1)$
$|\ simp\text{-}Plus\ (r_1,\ f_1)\ (r_2,\ f_2) =$
$(if\ r_1 = r_2\ then\ (r_1,\ F\text{-}LEFT\ f_1)\ else\ (Plus\ r_1\ r_2,\ F\text{-}Plus\ f_1\ f_2))$

fun *simp-Times* **where**
$simp\text{-}Times\ (Zero,\ f_1)\ (r_2,\ f_2) = (Zero,\ undefined)$
$|\ simp\text{-}Times\ (r_1,\ f_1)\ (Zero,\ f_2) = (Zero,\ undefined)$
$|\ simp\text{-}Times\ (One,\ f_1)\ (r_2,\ f_2) = (r_2,\ F\text{-}Times1\ f_1\ f_2)$
$|\ simp\text{-}Times\ (r_1,\ f_1)\ (One,\ f_2) = (r_1,\ F\text{-}Times2\ f_1\ f_2)$
$|\ simp\text{-}Times\ (r_1,\ f_1)\ (r_2,\ f_2) = (Times\ r_1\ r_2,\ F\text{-}Times\ f_1\ f_2)$

lemma *simp-Times-simps*[*simp*]:
simp-Times p1 p2 = (if (fst p1 = Zero) then (Zero, undefined)

$\quad\quad$ *else* (*if* (*fst p2* = *Zero*) *then* (*Zero*, *undefined*)
$\quad\quad$ *else* (*if* (*fst p1* = *One*) *then* (*fst p2*, *F-Times1* (*snd p1*) (*snd p2*))
$\quad\quad$ *else* (*if* (*fst p2* = *One*) *then* (*fst p1*, *F-Times2* (*snd p1*) (*snd p2*))
$\quad\quad$ *else* (*Times* (*fst p1*) (*fst p2*), *F-Times* (*snd p1*) (*snd p2*))))))
by (*induct p1 p2 rule*: *simp-Times.induct*)(*auto*)

lemma *simp-Plus-simps*[*simp*]:
simp-Plus p1 p2 = (*if* (*fst p1* = *Zero*) *then* (*fst p2*, *F-RIGHT* (*snd p2*))
$\quad\quad$ *else* (*if* (*fst p2* = *Zero*) *then* (*fst p1*, *F-LEFT* (*snd p1*))
$\quad\quad$ *else* (*if* (*fst p1* = *fst p2*) *then* (*fst p1*, *F-LEFT* (*snd p1*))
$\quad\quad$ *else* (*Plus* (*fst p1*) (*fst p2*), *F-Plus* (*snd p1*) (*snd p2*)))))
by (*induct p1 p2 rule*: *simp-Plus.induct*) (*auto*)

fun
simp :: *'a rexp* ⇒ *'a rexp* ∗ (*'a val* ⇒ *'a val*)
where
simp (*Plus r1 r2*) = *simp-Plus* (*simp r1*) (*simp r2*)
| *simp* (*Times r1 r2*) = *simp-Times* (*simp r1*) (*simp r2*)
| *simp r* = (*r*, *id*)

fun
slexer :: *'a rexp* ⇒ *'a list* ⇒ (*'a val*) *option*
where
slexer r [] = (*if nullable r then Some*(*mkeps r*) *else None*)
| *slexer r* (*c#s*) = (*let* (*rs*, *fr*) = *simp* (*deriv c r*) *in*
$\quad\quad$ (*case* (*slexer rs s*) *of*
$\quad\quad\quad$ *None* ⇒ *None*
$\quad\quad$ | *Some*(*v*) ⇒ *Some*(*injval r c* (*fr v*))))

lemma *slexer-better-simp*:
slexer r (*c#s*) = (*case* (*slexer* (*fst* (*simp* (*deriv c r*))) *s*) *of*
$\quad\quad\quad$ *None* ⇒ *None*
$\quad\quad$ | *Some*(*v*) ⇒ *Some*(*injval r c* ((*snd* (*simp* (*deriv c r*))) *v*)))
by (*auto split*: *prod.split option.split*)

lemma *L-fst-simp*:
shows *lang r* = *lang* (*fst* (*simp r*))
by (*induct r*) (*auto*)

lemma *Posix-simp*:
assumes $s \in (fst\ (simp\ r)) \to v$
shows $s \in r \to ((snd\ (simp\ r))\ v)$
using *assms*
proof(*induct r arbitrary*: *s v rule*: *rexp.induct*)
case (*Plus r1 r2 s v*)
have *IH1*: $\bigwedge s\ v.\ s \in fst\ (simp\ r1) \to v \Longrightarrow s \in r1 \to snd\ (simp\ r1)\ v$ **by** *fact*
have *IH2*: $\bigwedge s\ v.\ s \in fst\ (simp\ r2) \to v \Longrightarrow s \in r2 \to snd\ (simp\ r2)\ v$ **by** *fact*
have *as*: $s \in fst\ (simp\ (Plus\ r1\ r2)) \to v$ **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-NZero1*) *fst* (*simp r1*) ≠ *Zero fst* (*simp r2*) ≠ *Zero fst* (*simp r1*) = *fst* (*simp r2*)
| (*NZero-NZero2*) *fst* (*simp r1*) ≠ *Zero fst* (*simp r2*) ≠ *Zero fst* (*simp r1*) ≠ *fst* (*simp r2*) **by** *auto*
then show *s* ∈ *Plus r1 r2* → *snd* (*simp* (*Plus r1 r2*)) *v*
proof(*cases*)
case (*Zero-Zero*)
with *as* **have** *s* ∈ *Zero* → *v* **by** *simp*
then show *s* ∈ *Plus r1 r2* → *snd* (*simp* (*Plus 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 *lang* (*fst* (*simp r1*)) = {} **by** *simp*
then have *lang r1* = {} **using** *L-fst-simp* **by** *auto*
then have *s* ∉ *lang r1* **by** *simp*
ultimately have *s* ∈ *Plus r1 r2* → *Right* (*snd* (*simp r2*) *v*) **by** (*rule Posix-Plus2*)
then show *s* ∈ *Plus r1 r2* → *snd* (*simp* (*Plus 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* ∈ *Plus r1 r2* → *Left* (*snd* (*simp r1*) *v*) **by** (*rule Posix-Plus1*)
then show *s* ∈ *Plus r1 r2* → *snd* (*simp* (*Plus r1 r2*)) *v* **using** *NZero-Zero* **by** *simp*
next
case (*NZero-NZero1*)
with *as* **have** *a*: *s* ∈ *fst* (*simp r1*) → *v* **by** *simp*
then show *s* ∈ *Plus r1 r2* → *snd* (*simp* (*Plus r1 r2*)) *v*
using *IH1 NZero-NZero1 Posix-Plus1 a* **by** *fastforce*
next
case (*NZero-NZero2*)
with *as* **have** *s* ∈ *Plus* (*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* ∉ *lang* (*fst* (*simp r1*))
by (*erule-tac Posix-elims*(*4*))
then show *s* ∈ *Plus r1 r2* → *snd* (*simp* (*Plus 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* ∈ *Plus r1 r2* → *snd* (*simp* (*Plus r1 r2*)) *v* **using** *NZero-NZero2*

by (*simp-all add*: *Posix-Plus1*)
next
case (*Right*)
then have $v = Right\ v2\ s \in r2 \rightarrow (snd\ (simp\ r2)\ v2)\ s \notin lang\ r1$ **using** *IH2 L-fst-simp* **by** *auto*
then show $s \in Plus\ r1\ r2 \rightarrow snd\ (simp\ (Plus\ r1\ r2))\ v$ **using** *NZero-NZero2*
by (*simp-all add*: *Posix-Plus2*)
qed
qed
next
case (*Times r1 r2 s v*)
have *IH1*: $\bigwedge s\ v.\ s \in fst\ (simp\ r1) \rightarrow v \Longrightarrow s \in r1 \rightarrow snd\ (simp\ r1)\ v$ **by** *fact*
have *IH2*: $\bigwedge s\ v.\ s \in fst\ (simp\ r2) \rightarrow v \Longrightarrow s \in r2 \rightarrow snd\ (simp\ r2)\ v$ **by** *fact*
have *as*: $s \in fst\ (simp\ (Times\ r1\ r2)) \rightarrow v$ **by** *fact*
consider (*Zero*) $fst\ (simp\ r1) = Zero \vee fst\ (simp\ r2) = Zero$
| (*One-One*) $fst\ (simp\ r1) = One\ fst\ (simp\ r2) = One$
| (*One-NOne*) $fst\ (simp\ r1) = One\ fst\ (simp\ r2) \neq One\ fst\ (simp\ r2) \neq Zero$
| (*NOne-One*) $fst\ (simp\ r1) \neq One\ fst\ (simp\ r2) = One\ fst\ (simp\ r1) \neq Zero$
| (*NOne-NOne*) $fst\ (simp\ r1) \neq One\ fst\ (simp\ r2) \neq One$
$fst\ (simp\ r1) \neq Zero\ fst\ (simp\ r2) \neq Zero$ **by** *auto*
then show $s \in Times\ r1\ r2 \rightarrow snd\ (simp\ (Times\ r1\ r2))\ v$
proof(*cases*)
case (*Zero*)
with *as* **have** *False*
by (*metis Posix-elims*(*1*) *fst-conv simp.simps*(*2*) *simp-Times-simps*)
then show $s \in Times\ r1\ r2 \rightarrow snd\ (simp\ (Times\ r1\ r2))\ v$ **by** *simp*
next
case (*One-One*)
with *as* **have** *b*: $s \in One \rightarrow v$ **by** *simp*
from *b* **have** $s \in r1 \rightarrow snd\ (simp\ r1)\ v$ **using** *IH1 One-One* **by** *simp*
moreover
from *b* **have** *c*: $s = []\ v = Void$ **using** *Posix-elims*(*2*) **by** *auto*
moreover
have $[] \in One \rightarrow Void$ **by** (*simp add*: *Posix-One*)
then have $[] \in fst\ (simp\ r2) \rightarrow Void$ **using** *One-One* **by** *simp*
then have $[] \in r2 \rightarrow snd\ (simp\ r2)\ Void$ **using** *IH2* **by** *simp*
ultimately have $([]\ @\ []) \in Times\ r1\ r2 \rightarrow Seq\ (snd\ (simp\ r1)\ Void)\ (snd\ (simp\ r2)\ Void)$
using *Posix-Times* **by** *blast*
then show $s \in Times\ r1\ r2 \rightarrow snd\ (simp\ (Times\ r1\ r2))\ v$ **using** *c One-One* **by** *simp*
next
case (*One-NOne*)
with *as* **have** *b*: $s \in fst\ (simp\ r2) \rightarrow v$ **by** *simp*
from *b* **have** $s \in r2 \rightarrow snd\ (simp\ r2)\ v$ **using** *IH2 One-NOne* **by** *simp*
moreover
have $[] \in One \rightarrow Void$ **by** (*simp add*: *Posix-One*)

then have $[] \in fst\ (simp\ r1) \to Void$ **using** *One-NOne* **by** *simp*
then have $[] \in r1 \to snd\ (simp\ r1)\ Void$ **using** *IH1* **by** *simp*
moreover
from *One-NOne*(*1*) **have** $lang\ (fst\ (simp\ r1)) = \{[]\}$ **by** *simp*
then have $lang\ r1 = \{[]\}$ **by** (*simp add*: *L-fst-simp*[*symmetric*])
ultimately have $([]\ @\ s) \in Times\ r1\ r2 \to Seq\ (snd\ (simp\ r1)\ Void)\ (snd\ (simp\ r2)\ v)$
by(*rule-tac Posix-Times*) *auto*
then show $s \in Times\ r1\ r2 \to snd\ (simp\ (Times\ r1\ r2))\ v$ **using** *One-NOne* **by** *simp*
next
case (*NOne-One*)
with *as* **have** $s \in fst\ (simp\ r1) \to v$ **by** *simp*
with *IH1* **have** $s \in r1 \to snd\ (simp\ r1)\ v$ **by** *simp*
moreover
have $[] \in One \to Void$ **by** (*simp add*: *Posix-One*)
then have $[] \in fst\ (simp\ r2) \to Void$ **using** *NOne-One* **by** *simp*
then have $[] \in r2 \to snd\ (simp\ r2)\ Void$ **using** *IH2* **by** *simp*
ultimately have $(s\ @\ []) \in Times\ r1\ r2 \to Seq\ (snd\ (simp\ r1)\ v)\ (snd\ (simp\ r2)\ Void)$
by(*rule-tac Posix-Times*) *auto*
then show $s \in Times\ r1\ r2 \to snd\ (simp\ (Times\ r1\ r2))\ v$ **using** *NOne-One* **by** *simp*
next
case (*NOne-NOne*)
with *as* **have** $s \in Times\ (fst\ (simp\ r1))\ (fst\ (simp\ r2)) \to v$ **by** *simp*
then obtain *s1 s2 v1 v2* **where** *eqs*: $s = s1\ @\ s2$ $v = Seq\ v1\ v2$
$s1 \in (fst\ (simp\ r1)) \to v1$ $s2 \in (fst\ (simp\ r2)) \to v2$
$\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3\ @\ s_4 = s2 \wedge s1\ @\ s_3 \in lang\ r1 \wedge s_4 \in lang\ r2)$
by (*erule-tac Posix-elims*(*5*)) (*auto simp add*: *L-fst-simp*[*symmetric*])

then have $s1 \in r1 \to (snd\ (simp\ r1)\ v1)$ $s2 \in r2 \to (snd\ (simp\ r2)\ v2)$
using *IH1 IH2* **by** *auto*
then show $s \in Times\ r1\ r2 \to snd\ (simp\ (Times\ r1\ r2))\ v$ **using** *eqs NOne-NOne*
by(*auto intro*: *Posix-Times*)
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*: $\bigwedge r.\ slexer\ r\ s = lexer\ r\ s$ **by** *fact*

show *slexer r* (*c* # *s*) = *lexer r* (*c* # *s*)
proof (*cases s* ∈ *lang* (*deriv c r*))
case *True*
assume *a1*: *s* ∈ *lang* (*deriv c r*)
then obtain *v1* **where** *a2*: *lexer* (*deriv c r*) *s* = *Some v1 s* ∈ *deriv c r* → *v1*
using *lexer-correct-Some* **by** *auto*
from *a1* **have** *s* ∈ *lang* (*fst* (*simp* (*deriv c r*))) **using** *L-fst-simp*[*symmetric*] **by** *auto*
then obtain *v2* **where** *a3*: *lexer* (*fst* (*simp* (*deriv c r*))) *s* = *Some v2 s* ∈ (*fst* (*simp* (*deriv c r*))) → *v2*
using *lexer-correct-Some* **by** *auto*
then have *a4*: *slexer* (*fst* (*simp* (*deriv c r*))) *s* = *Some v2* **using** *IH* **by** *simp*
from *a3*(*2*) **have** *s* ∈ *deriv c r* → (*snd* (*simp* (*deriv c r*))) *v2* **using** *Posix-simp* **by** *auto*
with *a2*(*2*) **have** *v1* = (*snd* (*simp* (*deriv c r*))) *v2* **using** *Posix-determ* **by** *auto*
with *a2*(*1*) *a4* **show** *slexer r* (*c* # *s*) = *lexer r* (*c* # *s*) **by** (*auto split*: *prod.split*)
next
case *False*
assume *b1*: *s* ∉ *lang* (*deriv c r*)
then have *lexer* (*deriv c r*) *s* = *None* **using** *lexer-correct-None* **by** *auto*
moreover
from *b1* **have** *s* ∉ *lang* (*fst* (*simp* (*deriv c r*))) **using** *L-fst-simp*[*symmetric*] **by** *auto*
then have *lexer* (*fst* (*simp* (*deriv c r*))) *s* = *None* **using** *lexer-correct-None* **by** *auto*
then have *slexer* (*fst* (*simp* (*deriv 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 *Positions3*
imports *Lexer3 LexicalVals3*
begin

26 An alternative definition for POSIX values by Okui & Suzuki

27 Positions in Values

fun
at :: ′*a val* ⇒ *nat list* ⇒ ′*a val*

where
  $at\ v\ []\ =\ v$
| $at\ (Left\ v)\ (0\#ps)=\ at\ v\ ps$
| $at\ (Right\ v)\ (Suc\ 0\#ps)=\ at\ v\ ps$
| $at\ (Seq\ v1\ v2)\ (0\#ps)=\ at\ v1\ ps$
| $at\ (Seq\ v1\ v2)\ (Suc\ 0\#ps)=\ at\ v2\ ps$
| $at\ (Stars\ vs)\ (n\#ps)\ =\ at\ (nth\ vs\ n)\ ps$
| $at\ (Recv\ l\ v)\ ps\ =\ at\ v\ ps$

fun $Pos :: 'a\ val \Rightarrow (nat\ list)\ set$
where
  $Pos\ (Void) = \{[]\}$
| $Pos\ (Atm\ c) = \{[]\}$
| $Pos\ (Left\ v) = \{[]\} \cup \{0\#ps \mid ps.\ ps \in Pos\ v\}$
| $Pos\ (Right\ v) = \{[]\} \cup \{1\#ps \mid ps.\ ps \in Pos\ v\}$
| $Pos\ (Seq\ v1\ v2) = \{[]\} \cup \{0\#ps \mid ps.\ ps \in Pos\ v1\} \cup \{1\#ps \mid ps.\ ps \in Pos\ v2\}$
| $Pos\ (Stars\ []) = \{[]\}$
| $Pos\ (Stars\ (v\#vs)) = \{[]\} \cup \{0\#ps \mid ps.\ ps \in Pos\ v\} \cup \{Suc\ n\#ps \mid n\ ps.\ n\#ps$
$\in Pos\ (Stars\ vs)\}$
| $Pos\ (Recv\ l\ v) = \{[]\} \cup \{ps\ .\ ps \in Pos\ v\}$

lemma *Pos-stars*:
  $Pos\ (Stars\ vs) = \{[]\} \cup (\bigcup n < length\ vs.\ \{n\#ps \mid ps.\ ps \in Pos\ (vs\ !\ n)\})$
apply(*induct vs*)
apply(*auto simp add*: *insert-ident less-Suc-eq-0-disj*)
done

lemma *Pos-empty*:
  shows $[] \in Pos\ v$
by (*induct v rule*: *Pos.induct*)(*auto*)

abbreviation
  $intlen\ vs \equiv int\ (length\ vs)$

definition $pflat\text{-}len :: 'a\ val \Rightarrow nat\ list => int$
where
  $pflat\text{-}len\ v\ p \equiv (if\ p \in Pos\ v\ then\ intlen\ (flat\ (at\ v\ p))\ else\ -1)$

lemma *pflat-len-simps*:
  shows $pflat\text{-}len\ (Seq\ v1\ v2)\ (0\#p) = pflat\text{-}len\ v1\ p$
  and $pflat\text{-}len\ (Seq\ v1\ v2)\ (Suc\ 0\#p) = pflat\text{-}len\ v2\ p$
  and $pflat\text{-}len\ (Left\ v)\ (0\#p) = pflat\text{-}len\ v\ p$
  and $pflat\text{-}len\ (Left\ v)\ (Suc\ 0\#p) = -1$
  and $pflat\text{-}len\ (Right\ v)\ (Suc\ 0\#p) = pflat\text{-}len\ v\ p$
  and $pflat\text{-}len\ (Right\ v)\ (0\#p) = -1$

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* (*Recv l v*) *p* = *pflat-len v p*
and *pflat-len v* [] = *intlen* (*flat v*)
apply (*auto simp add*: *pflat-len-def Pos-empty*)
by (*metis at.simps*(*7*) *neq-Nil-conv*)

lemma *pflat-len-Stars-simps*:
assumes $n < \mathit{length}\ vs$
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 $p \notin \mathit{Pos}\ v1$
shows *pflat-len v1 p* = *−1*
using *assms* **by** (*simp add*: *pflat-len-def*)

28 Orderings

definition *prefix-list*:: *'a list* ⇒ *'a list* ⇒ *bool* (‹- ⊑*pre* -› [*60*,*59*] *60*)
where
ps1 ⊑*pre ps2* ≡ ∃ *ps'*. *ps1* @*ps'* = *ps2*

definition *sprefix-list*:: *'a list* ⇒ *'a list* ⇒ *bool* (‹- ⊏*spre* -› [*60*,*59*] *60*)
where
ps1 ⊏*spre ps2* ≡ *ps1* ⊑*pre ps2* ∧ *ps1* ≠ *ps2*

inductive *lex-list* :: *nat list* ⇒ *nat list* ⇒ *bool* (‹- ⊏*lex* -› [*60*,*59*] *60*)
where
[] ⊏*lex* (*p*#*ps*)
| *ps1* ⊏*lex ps2* ⟹ (*p*#*ps1*) ⊏*lex* (*p*#*ps2*)
| *p1* < *p2* ⟹ (*p1*#*ps1*) ⊏*lex* (*p2*#*ps2*)

lemma *lex-irrfl*:
fixes *ps1 ps2* :: *nat list*
assumes *ps1* ⊏*lex ps2*
shows *ps1* ≠ *ps2*
using *assms*
by(*induct rule*: *lex-list.induct*)(*auto*)

lemma *lex-simps* [*simp*]:
fixes *xs ys* :: *nat list*
shows [] ⊏*lex ys* ⟷ *ys* ≠ []
and *xs* ⊏*lex* [] ⟷ *False*
and (*x* # *xs*) ⊏*lex* (*y* # *ys*) ⟷ (*x* < *y* ∨ (*x* = *y* ∧ *xs* ⊏*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

29 POSIX Ordering of Values According to Okui & Suzuki

definition *PosOrd*:: *'a val* ⇒ *nat list* ⇒ *'a val* ⇒ *bool* (‹- $\sqsubset$*val* - -› [*60*, *60*, *59*] *60*)
where
v1 $\sqsubset$*val p v2* ≡ *pflat-len v1 p* > *pflat-len v2 p* ∧
(∀ *q* ∈ *Pos v1* ∪ *Pos v2*. *q* $\sqsubset$*lex p* ⟶ *pflat-len v1 q* = *pflat-len v2 q*)

lemma *PosOrd-def2*:
shows *v1* $\sqsubset$*val p v2* ⟷
pflat-len v1 p > *pflat-len v2 p* ∧
(∀ *q* ∈ *Pos v1*. *q* $\sqsubset$*lex p* ⟶ *pflat-len v1 q* = *pflat-len v2 q*) ∧
(∀ *q* ∈ *Pos v2*. *q* $\sqsubset$*lex p* ⟶ *pflat-len v1 q* = *pflat-len v2 q*)
unfolding *PosOrd-def*
apply(*auto*)
done

definition *PosOrd-ex*:: *'a val* ⇒ *'a val* ⇒ *bool* (‹- :$\sqsubset$*val* -› [*60*, *59*] *60*)
where
v1 :$\sqsubset$*val v2* ≡ ∃ *p*. *v1* $\sqsubset$*val p v2*

definition *PosOrd-ex-eq*:: *'a val* ⇒ *'a val* ⇒ *bool* (‹- :$\sqsubseteq$*val* -› [*60*, *59*] *60*)
where
v1 :$\sqsubseteq$*val v2* ≡ *v1* :$\sqsubset$*val v2* ∨ *v1* = *v2*

lemma *PosOrd-trans*:
  assumes $v1 :\sqsubset val\ v2$ $v2 :\sqsubset val\ v3$
  shows $v1 :\sqsubset val\ v3$
proof −
  from *assms* **obtain** $p\ p'$
    where *as*: $v1 \sqsubset val\ p\ v2$ $v2 \sqsubset val\ p'\ v3$ **unfolding** *PosOrd-ex-def* **by** *blast*
  then have *pos*: $p \in Pos\ v1$ $p' \in Pos\ v2$ **unfolding** *PosOrd-def pflat-len-def*
   by (*metis* (*full-types*) *int-ops*(*2*) *not-int-zless-negative verit-comp-simplify1*(*1*))
      (*metis PosOrd-def2 as*(*2*) *int-ops*(*2*) *not-int-zless-negative order-less-irrefl pflat-len-def*)
  have $p = p' \lor p \sqsubset lex\ p' \lor p' \sqsubset lex\ p$
    by (*rule lex-trichotomous*)
  moreover
    { **assume** $p = p'$
      with *as* **have** $v1 \sqsubset val\ p\ v3$ **unfolding** *PosOrd-def pflat-len-def*
        by (*smt* (*verit*, *best*) *UnCI*)
      then have $v1 :\sqsubset val\ v3$ **unfolding** *PosOrd-ex-def* **by** *blast*
    }
  moreover
    { **assume** $p \sqsubset lex\ p'$
      with *as* **have** $v1 \sqsubset val\ p\ v3$ **unfolding** *PosOrd-def pflat-len-def*
      by (*smt* (*verit*, *best*) *UnCI lex-trans*)
      then have $v1 :\sqsubset val\ v3$ **unfolding** *PosOrd-ex-def* **by** *blast*
    }
  moreover
    { **assume** $p' \sqsubset lex\ p$
      with *as* **have** $v1 \sqsubset val\ p'\ v3$ **unfolding** *PosOrd-def*
        by (*smt* (*verit*, *best*) *Un-iff lex-trans pflat-len-def*)
      then have $v1 :\sqsubset val\ v3$ **unfolding** *PosOrd-ex-def* **by** *blast*
    }
  ultimately show $v1 :\sqsubset val\ v3$ **by** *blast*
qed

lemma *PosOrd-irrefl*:
  assumes $v :\sqsubset val\ v$
  shows *False*
using *assms* **unfolding** *PosOrd-ex-def PosOrd-def*
by *auto*

lemma *PosOrd-assym*:
  assumes $v1 :\sqsubset val\ v2$
  shows $\neg(v2 :\sqsubset val\ v1)$
using *assms*
using *PosOrd-irrefl PosOrd-trans* **by** *blast*

lemma *PosOrd-ordering*:

shows *ordering* (λ*v1 v2*. *v1* :$\sqsubseteq$*val v2*) (λ *v1 v2*. *v1* :$\sqsubset$*val v2*)
unfolding *ordering-def PosOrd-ex-eq-def*
apply(*auto*)
using *PosOrd-trans partial-preordering-def* **apply** *blast*
using *PosOrd-assym ordering-axioms-def* **by** *blast*

lemma *PosOrd-order*:
shows *class.order* (λ*v1 v2*. *v1* :$\sqsubseteq$*val v2*) (λ *v1 v2*. *v1* :$\sqsubset$*val v2*)
using *PosOrd-ordering*
apply(*simp add*: *class.order-def class.preorder-def class.order-axioms-def*)
by (*smt* (*verit*) *PosOrd-ex-eq-def PosOrd-irrefl PosOrd-trans*)

lemma *PosOrd-ex-eq2*:
shows *v1* :$\sqsubset$*val v2* $\longleftrightarrow$ (*v1* :$\sqsubseteq$*val v2* $\wedge$ *v1* $\neq$ *v2*)
using *PosOrd-ordering*
using *PosOrd-ex-eq-def PosOrd-irrefl* **by** *blast*

lemma *PosOrdeq-trans*:
assumes *v1* :$\sqsubseteq$*val v2 v2* :$\sqsubseteq$*val v3*
shows *v1* :$\sqsubseteq$*val v3*
using *assms PosOrd-ordering*
unfolding *ordering-def*
by (*metis partial-preordering.trans*)

lemma *PosOrdeq-antisym*:
assumes *v1* :$\sqsubseteq$*val v2 v2* :$\sqsubseteq$*val v1*
shows *v1* = *v2*
using *assms PosOrd-ordering*
by (*metis ordering.eq-iff*)

lemma *PosOrdeq-refl*:
shows *v* :$\sqsubseteq$*val v*
unfolding *PosOrd-ex-eq-def*
by *auto*

lemma *PosOrd-shorterE*:
assumes *v1* :$\sqsubset$*val v2*
shows *length* (*flat v2*) $\leq$ *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-Rec-eq*:
  assumes *flat v1* = *flat v2*
  shows *Recv l v1* :⊏*val Recv l v2* ⟷ *v1* :⊏*val v2*
unfolding *PosOrd-ex-def PosOrd-def2*
  using *assms*
  apply(*auto*)
  apply (*simp add*: *pflat-len-simps*(*10*))
  apply (*metis pflat-len-simps*(*9*))
  by (*metis pflat-len-simps*(*10*) *pflat-len-simps*(*9*))

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* (*verit*) *Collect-disj-eq lex-list.intros*(*2*) *mem-Collect-eq pflat-len-simps*(*2*))
apply(*smt* (*verit*) *Collect-disj-eq lex-list.intros*(*2*) *mem-Collect-eq pflat-len-simps*(*2*))
done

lemma *PosOrd-StarsI*:
  assumes $v1 :\sqsubset val\ v2$ $flats\ (v1\#vs1) = flats\ (v2\#vs2)$
  shows $Stars\ (v1\#vs1) :\sqsubset val\ Stars\ (v2\#vs2)$
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 $Stars\ vs1 :\sqsubset val\ Stars\ vs2$ $flats\ vs1 = flats\ vs2$
  shows $Stars\ (v\#vs1) :\sqsubset val\ Stars\ (v\#vs2)$
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 $Stars\ vs1 :\sqsubset val\ Stars\ vs2$ $flat\ (Stars\ vs1) = flat\ (Stars\ vs2)$
  shows $Stars\ (vs\ @\ vs1) :\sqsubset val\ Stars\ (vs\ @\ vs2)$
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-Stars-equalsI*:
assumes *flats vs1* = *flats vs2 length vs1* = *length vs2*
and *list-all2* (λ*v1 v2*. *v1* :⊑*val v2*) *vs1 vs2*
shows *Stars vs1* :⊑*val Stars vs2*
using *assms*(*3*) *assms*(*2*,*1*)
apply(*induct rule*: *list-all2-induct*)
apply (*simp add*: *PosOrdeq-refl*)
apply(*case-tac Stars* (*x* # *xs*) = *Stars* (*y* # *ys*))
apply (*simp add*: *PosOrdeq-refl*)
apply(*case-tac x* = *y*)
apply(*subgoal-tac Stars xs* :⊏*val Stars ys*)
apply (*simp add*: *PosOrd-StarsI2 PosOrd-ex-eq-def*)
apply (*simp add*: *PosOrd-ex-eq2*)
by (*meson PosOrd-StarsI PosOrd-ex-eq-def*)

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

30 The Posix Value is smaller than any other lexical 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-Atom c v*)
have $v \in LV$ (*Atom c*) [*c*] **by** *fact*
then have $v = Atm\ c$
by (*simp add*: *LV-simps*)
then show *Atm c* :⊑*val v*
by (*simp add*: *PosOrd-ex-eq-def*)
next
case (*Posix-Plus1 s r1 v r2 v2*)
have *as1*: $s \in r1 \to v$ **by** *fact*
have *IH*: $\bigwedge v2.\ v2 \in LV\ r1\ s \Longrightarrow v$:⊑*val v2* **by** *fact*
have $v2 \in LV$ (*Plus r1 r2*) *s* **by** *fact*
then have $\vdash v2$: *Plus r1 r2 flat v2* $= s$
by(*auto simp add*: *LV-def prefix-list-def*)
then consider
(*Left*) *v3* **where** $v2 =$ *Left v3* $\vdash v3 : r1$ *flat v3* $= s$
| (*Right*) *v3* **where** $v2 =$ *Right v3* $\vdash v3 : r2$ *flat v3* $= s$
by (*auto elim*: *Prf.cases*)
then show *Left v* :⊑*val v2*
proof(*cases*)
case (*Left v3*)
have $v3 \in LV\ r1\ s$ **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-Plus2 s r2 v r1 v2*)
have *as1*: $s \in r2 \to v$ **by** *fact*
have *as2*: $s \notin$ *lang r1* **by** *fact*
have *IH*: $\bigwedge v2.\ v2 \in LV\ r2\ s \Longrightarrow v$:⊑*val v2* **by** *fact*
have $v2 \in LV$ (*Plus r1 r2*) *s* **by** *fact*
then have $\vdash v2$: *Plus r1 r2 flat v2* $= s$
by(*auto simp add*: *LV-def prefix-list-def*)
then consider
(*Left*) *v3* **where** $v2 =$ *Left v3* $\vdash v3 : r1$ *flat v3* $= s$
| (*Right*) *v3* **where** $v2 =$ *Right v3* $\vdash v3 : r2$ *flat v3* $= s$

by (*auto elim*: *Prf.cases*)
then show *Right v* $:\sqsubseteq$*val v2*
proof (*cases*)
case (*Right v3*)
have *v3* $\in$ *LV r2 s* **using** *Right*(*2*,*3*)
by (*auto simp add*: *LV-def prefix-list-def*)
with *IH* **have** *v* $:\sqsubseteq$*val v3* **by** *simp*
moreover
have *flat v3* = *flat v* **using** *as1 Right*(*3*)
by (*simp add*: *Posix1*(*2*))
ultimately have *Right v* $:\sqsubseteq$*val Right v3*
by (*auto simp add*: *PosOrd-ex-eq-def PosOrd-RightI*)
then show *Right v* $:\sqsubseteq$*val v2* **unfolding** *Right* **.**
next
case (*Left v3*)
have *v3* $\in$ *LV r1 s* **using** *Left*(*2*,*3*) *as2*
by (*auto simp add*: *LV-def prefix-list-def*)
then have *flat v3* = *flat v* $\wedge \vdash$ *v3* : *r1* **using** *as1 Left*(*3*)
by (*simp add*: *Posix1*(*2*) *LV-def*)
then have *False* **using** *as1 as2 Left*
using *Prf-flat-lang* **by** *blast*
then show *Right v* $:\sqsubseteq$*val v2* **by** *simp*
qed
next
case (*Posix-Times s1 r1 v1 s2 r2 v2 v3*)
have *s1* $\in$ *r1* $\rightarrow$ *v1 s2* $\in$ *r2* $\rightarrow$ *v2* **by** *fact+*
then have *as1*: *s1* = *flat v1 s2* = *flat v2* **by** (*simp-all add*: *Posix1*(*2*))
have *IH1*: $\bigwedge$*v3*. *v3* $\in$ *LV r1 s1* $\Longrightarrow$ *v1* $:\sqsubseteq$*val v3* **by** *fact*
have *IH2*: $\bigwedge$*v3*. *v3* $\in$ *LV r2 s2* $\Longrightarrow$ *v2* $:\sqsubseteq$*val v3* **by** *fact*
have *cond*: $\neg$ ($\exists\, s_3\ s_4.\ s_3 \neq [] \wedge s_3$ @ s_4 = *s2* $\wedge$ *s1* @ $s_3 \in$ *lang r1* $\wedge\ s_4 \in$ *lang r2*) **by** *fact*
have *v3* $\in$ *LV* (*Times r1 r2*) (*s1* @ *s2*) **by** *fact*
then obtain *v3a v3b* **where** *eqs*:
v3 = *Seq v3a v3b* $\vdash$ *v3a* : *r1* $\vdash$ *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* $\sqsubseteq$*pre s1* **unfolding** *prefix-list-def*
by (*smt* (*verit*, *ccfv-SIG*) *Prf-flat-lang append.right-neutral append-eq-append-conv2*)
then have *flat v3a* $\sqsubset$*spre s1* $\vee$ (*flat v3a* = *s1* $\wedge$ *flat v3b* = *s2*) **using** *eqs*
by (*simp add*: *sprefix-list-def append-eq-conv-conj*)
then have *q2*: *v1* $:\sqsubset$*val v3a* $\vee$ (*flat v3a* = *s1* $\wedge$ *flat v3b* = *s2*)
using *PosOrd-spreI as1*(*1*) *eqs* **by** *blast*
then have *v1* $:\sqsubset$*val v3a* $\vee$ (*v3a* $\in$ *LV r1 s1* $\wedge$ *v3b* $\in$ *LV r2 s2*) **using** *eqs*(*2*,*3*)
by (*auto simp add*: *LV-def*)
then have *v1* $:\sqsubset$*val v3a* $\vee$ (*v1* $:\sqsubseteq$*val v3a* $\wedge$ *v2* $:\sqsubseteq$*val v3b*) **using** *IH1 IH2* **by** *blast*
then have *Seq v1 v2* $:\sqsubseteq$*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*: ¬ (∃ s_3 s_4. s_3 ≠ [] ∧ s_3 @ s_4 = *s2* ∧ *s1* @ s_3 ∈ *lang r* ∧ s_4 ∈ *lang* (*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-elims*)
by (*metis NonEmpty Prf.intros*(*6*) *list.set-intros*(*1*) *list.set-intros*(*2*) *neq-Nil-conv*)
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* (*verit*) *Prf-flat-lang append.right-neutral 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 (*metis flat.simps*(*7*) *flat-Stars val.inject*(*5*))
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*)
next
case (*Posix-NTimes2 vs r n v2*)
have *IH*: ∀ *v*∈*set vs*. [] ∈ *r* → *v* ∧ (∀ *x*. *x* ∈ *LV r* [] ⟶ *v* :⊑*val x*) **by** *fact*
then have *flats vs* = []
by (*metis Posix.Posix-NTimes2 Posix1*(*2*) *flat-Stars*)
have *v2* ∈ *LV* (*NTimes r n*) [] **by** *fact*
then obtain *vs′* **where** *eq*: *v2* = *Stars vs′* **and** *length vs′* = *n* **and** *as*: ∀ *v* ∈ *set vs′*. *v* ∈ *LV r* [] ∧ *flat v* = []
unfolding *LV-def* **by** (*auto elim*!: *Prf-elims*)
then have *Stars vs* :⊑*val Stars vs′*
apply(*rule-tac PosOrd-Stars-equalsI*)
apply (*simp add*: ‹*flats vs* = []›)
using *Posix-NTimes2.hyps*(*2*) **apply** *blast*
using *IH* **apply**(*simp add*: *list-all2-iff*)
apply(*auto*)
using *Posix-NTimes2.hyps*(*2*) **apply** *blast*
by (*meson in-set-zipE*)
then show *Stars vs* :⊑*val v2* **using** *eq* **by** *simp*
next
case (*Posix-NTimes1 s1 r v s2 n vs*)
have *s1* ∈ *r* → *v s2* ∈ *NTimes r n* → *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* (*NTimes r n*) *s2* ⟹ *Stars vs* :⊑*val v3* **by** *fact*
have *cond*: ¬ (∃ s_3 s_4. s_3 ≠ [] ∧ s_3 @ s_4 = *s2* ∧ *s1* @ s_3 ∈ *lang r* ∧ s_4 ∈ *lang* (*NTimes r n*)) **by** *fact*
have *cond2*: *flat v* ≠ [] **by** *fact*
have *v2* ∈ *LV* (*NTimes r* (*n* + *1*)) (*s1* @ *s2*) **by** *fact*
then consider
(*NonEmpty*) *v3a vs3* **where** *v2* = *Stars* (*v3a* # *vs3*)
⊢ *v3a* : *r* ⊢ *Stars vs3* : *NTimes r n*
flat (*Stars* (*v3a* # *vs3*)) = *s1* @ *s2*
| (*Empty*) *v2* = *Stars* []
unfolding *LV-def*
apply(*auto*)
apply(*erule Prf-elims*)
apply(*case-tac vs1*)
apply(*simp add*: *as1*(*1*) *cond2 flats-empty*)
apply(*simp*)

using *Prf.simps* **apply** *fastforce*
done
then show *Stars* (*v* # *vs*) :$\sqsubseteq$*val v2*
proof (*cases*)
case (*NonEmpty v3a vs3*)
have *flat* (*Stars* (*v3a* # *vs3*)) = *s1* @ *s2* **using** *NonEmpty*(*4*) **.**
with *cond* **have** *flat v3a* $\sqsubseteq$*pre s1* **using** *NonEmpty*(*2*,*3*)
unfolding *prefix-list-def*
by (*smt* (*verit*) *Prf-flat-lang append.right-neutral append-eq-append-conv2 flat.simps*(*7*))
then have *flat v3a* $\sqsubset$*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* :$\sqsubset$*val v3a* ∨ (*flat v3a* = *s1* ∧ *flat* (*Stars vs3*) = *s2*)
using *PosOrd-spreI as1*(*1*) *NonEmpty*(*4*) **by** *blast*
then have *v* :$\sqsubset$*val v3a* ∨ (*v3a* ∈ *LV r s1* ∧ *Stars vs3* ∈ *LV* (*NTimes r n*) *s2*)
using *NonEmpty*(*2*,*3*) **by** (*auto simp add*: *LV-def*)
then have *v* :$\sqsubset$*val v3a* ∨ (*v* :$\sqsubseteq$*val v3a* ∧ *Stars vs* :$\sqsubseteq$*val Stars vs3*) **using** *IH1 IH2* **by** *blast*
then have *v* :$\sqsubset$*val v3a* ∨ (*v* = *v3a* ∧ *Stars vs* :$\sqsubseteq$*val Stars vs3*)
unfolding *PosOrd-ex-eq-def* **by** *auto*
then have *Stars* (*v* # *vs*) :$\sqsubseteq$*val Stars* (*v3a* # *vs3*) **using** *NonEmpty*(*4*) *q2 as1*
unfolding *PosOrd-ex-eq-def*
using *PosOrd-StarsI PosOrd-StarsI2*
by (*metis flat.simps*(*7*) *flat-Stars val.inject*(*5*))
then show *Stars* (*v* # *vs*) :$\sqsubseteq$*val v2* **unfolding** *NonEmpty* **by** *blast*
next
case *Empty*
have *v2* = *Stars* [] **by** *fact*
then show *Stars* (*v* # *vs*) :$\sqsubseteq$*val v2*
unfolding *PosOrd-ex-eq-def* **using** *cond2*
by (*simp add*: *PosOrd-shorterI*)
qed
next
case (*Posix-Upto1 s1 r v s2 n vs v3*)
have *s1* ∈ *r* → *v s2* ∈ *Upto r n* → *Stars vs* **by** *fact*+
then have *as1*: *s1* = *flat v s2* = *flat* (*Stars vs*) **by** (*auto dest*: *Posix1*(*2*))
have *IH1*: $\bigwedge$*v3*. *v3* ∈ *LV r s1* $\Longrightarrow$ *v* :$\sqsubseteq$*val v3* **by** *fact*
have *IH2*: $\bigwedge$*v3*. *v3* ∈ *LV* (*Upto r n*) *s2* $\Longrightarrow$ *Stars vs* :$\sqsubseteq$*val v3* **by** *fact*
have *cond*: ¬ (∃ s_3 s_4. s_3 ≠ [] ∧ s_3 @ s_4 = *s2* ∧ *s1* @ s_3 ∈ *lang r* ∧ s_4 ∈ *lang* (*Upto r n*)) **by** *fact*
have *cond2*: *flat v* ≠ [] **by** *fact*
have *v3* ∈ *LV* (*Upto r* (*n* + *1*)) (*s1* @ *s2*) **by** *fact*
then consider
(*NonEmpty*) *v3a vs3* **where** *v3* = *Stars* (*v3a* # *vs3*)
⊢ *v3a* : *r* ⊢ *Stars vs3* : *Upto r n*
flat (*Stars* (*v3a* # *vs3*)) = *s1* @ *s2*

```
  | (Empty) v3 = Stars []
  unfolding LV-def
  apply(auto)
  apply(erule Prf-elims)
  apply(case-tac vs)
   apply(auto)
  by (simp add: Prf.intros(8))
  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 (verit) Prf-flat-lang append.right-neutral 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 (Upto r n) 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 (metis flat.simps(7) flat-Stars val.inject(5))
      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-Upto2 r n v2)
    have v2 ∈ LV (Upto r n) [] 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)
next
  case (Posix-From2 vs r n)
```

```
  then show Stars vs :⊑val v2
    apply(simp add: LV-def)
      apply(auto)
    apply(erule Prf-elims)
     apply(auto)
    apply(rule PosOrd-Stars-equalsI)
    apply (metis Posix1(2) flats-empty)
      apply(simp)
    apply(auto simp add: list-all2-iff)
    apply (meson set-zip-leftD set-zip-rightD)
    done
next
  case (Posix-From1 s1 r v s2 n vs v3)
  have s1 ∈ r → v s2 ∈ From r (n − 1) → Stars vs by fact+
  then have as1: s1 = flat v s2 = flats vs by (auto dest: Posix1(2))
  have IH1: ⋀v3. v3 ∈ LV r s1 ⟹ v :⊑val v3 by fact
  have IH2: ⋀v3. v3 ∈ LV (From r (n − 1)) s2 ⟹ Stars vs :⊑val v3 by fact
  have cond: ¬ (∃ s₃ s₄. s₃ ≠ [] ∧ s₃ @ s₄ = s2 ∧ s1 @ s₃ ∈ lang r ∧ s₄ ∈ lang
(From r (n − 1))) by fact
  have cond2: flat v ≠ [] by fact
  have v3 ∈ LV (From r n) (s1 @ s2) by fact
  then consider
    (NonEmpty) v3a vs3 where v3 = Stars (v3a # vs3)
    ⊢ v3a : r ⊢ Stars vs3 : From r (n − 1)
    flats (v3a # vs3) = s1 @ s2
  | (Empty) v3 = Stars []
  unfolding LV-def
  apply(auto)
  apply(erule Prf.cases)
            apply(auto)
  apply(case-tac vs1)
   apply(auto intro: Prf.intros)
   apply(case-tac vs2)
    apply(auto intro: Prf.intros)
    apply (simp add: as1(1) cond2 flats-empty)
  apply (simp add: Prf.intros)
  apply(case-tac vs)
   apply(auto)
  by (metis Posix-From1.hyps(6) Prf.intros(10) Suc-le-eq Suc-pred less-Suc-eq-le)
  then show Stars (v # vs) :⊑val v3
    proof (cases)
      case (NonEmpty v3a vs3)
      have flats (v3a # vs3) = s1 @ s2 using NonEmpty(4) .
      with cond have flat v3a ⊑pre s1 using NonEmpty(2,3)
        unfolding prefix-list-def
          by (smt (verit) Prf-flat-lang append.right-neutral append-eq-append-conv2
flat.simps(7)
            flat-Stars)
     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*: $v :\sqsubset val\ v3a \vee (flat\ v3a = s1 \wedge flat\ (Stars\ vs3) = s2)$
using *PosOrd-spreI as1*(*1*) *NonEmpty*(*4*) **by** *blast*
then have $v :\sqsubset val\ v3a \vee (v3a \in LV\ r\ s1 \wedge Stars\ vs3 \in LV\ (From\ r\ (n - 1))\ s2)$
using *NonEmpty*(*2*,*3*) **by** (*auto simp add*: *LV-def*)
then have $v :\sqsubset val\ v3a \vee (v :\sqsubseteq val\ v3a \wedge Stars\ vs :\sqsubseteq val\ Stars\ vs3)$ **using** *IH1 IH2* **by** *blast*
then have $v :\sqsubset val\ v3a \vee (v = v3a \wedge Stars\ vs :\sqsubseteq val\ Stars\ vs3)$
unfolding *PosOrd-ex-eq-def* **by** *auto*
then have $Stars\ (v \# vs) :\sqsubseteq val\ Stars\ (v3a \# vs3)$ **using** *NonEmpty*(*4*) *q2 as1*
unfolding *PosOrd-ex-eq-def*
by (*metis PosOrd-StarsI PosOrd-StarsI2 flat.simps*(*7*) *flat-Stars val.inject*(*5*))
then show $Stars\ (v \# vs) :\sqsubseteq val\ v3$ **unfolding** *NonEmpty* **by** *blast*
next
case *Empty*
have $v3 = Stars\ []$ **by** *fact*
then show $Stars\ (v \# vs) :\sqsubseteq val\ v3$
unfolding *PosOrd-ex-eq-def* **using** *cond2*
by (*simp add*: *PosOrd-shorterI*)
qed
next
case (*Posix-From3 s1 r v s2 vs v3*)
have $s1 \in r \rightarrow v\ s2 \in Star\ r \rightarrow Stars\ vs$ **by** *fact+*
then have *as1*: $s1 = flat\ v\ s2 = flat\ (Stars\ vs)$ **by** (*auto dest*: *Posix1*(*2*))
have *IH1*: $\bigwedge v3.\ v3 \in LV\ r\ s1 \Longrightarrow v :\sqsubseteq val\ v3$ **by** *fact*
have *IH2*: $\bigwedge v3.\ v3 \in LV\ (Star\ r)\ s2 \Longrightarrow Stars\ vs :\sqsubseteq val\ v3$ **by** *fact*
have *cond*: $\neg (\exists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s2 \wedge s1 @ s_3 \in lang\ r \wedge s_4 \in lang\ (Star\ r))$ **by** *fact*
have *cond2*: $flat\ v \neq []$ **by** *fact*
have $v3 \in LV\ (From\ r\ 0)\ (s1 @ s2)$ **by** *fact*
then consider
(*NonEmpty*) *v3a vs3* **where** $v3 = Stars\ (v3a \# vs3)$
$\vdash v3a : r \vdash Stars\ vs3 : Star\ r$
$flat\ (Stars\ (v3a \# vs3)) = s1 @ s2$
| (*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 $Stars\ (v \# vs) :\sqsubseteq val\ v3$
proof (*cases*)
case (*NonEmpty v3a vs3*)
have $flat\ (Stars\ (v3a \# vs3)) = s1 @ s2$ **using** *NonEmpty*(*4*) **.**

with *cond* **have** *flat v3a* $\sqsubseteq$*pre s1* **using** *NonEmpty*(*2*,*3*)
unfolding *prefix-list-def*
by (*smt* (*verit*) *Prf-flat-lang append.right-neutral append-eq-append-conv2 flat.simps*(*7*))
then have *flat v3a* $\sqsubset$*spre s1* $\vee$ (*flat v3a* = *s1* $\wedge$ *flat* (*Stars vs3*) = *s2*) **using** *NonEmpty*(*4*)
by (*simp add*: *sprefix-list-def append-eq-conv-conj*)
then have *q2*: *v* :$\sqsubset$*val v3a* $\vee$ (*flat v3a* = *s1* $\wedge$ *flat* (*Stars vs3*) = *s2*)
using *PosOrd-spreI as1*(*1*) *NonEmpty*(*4*) **by** *blast*
then have *v* :$\sqsubset$*val v3a* $\vee$ (*v3a* $\in$ *LV r s1* $\wedge$ *Stars vs3* $\in$ *LV* (*Star r*) *s2*)
using *NonEmpty*(*2*,*3*) **by** (*auto simp add*: *LV-def*)
then have *v* :$\sqsubset$*val v3a* $\vee$ (*v* :$\sqsubseteq$*val v3a* $\wedge$ *Stars vs* :$\sqsubseteq$*val Stars vs3*) **using** *IH1 IH2* **by** *blast*
then have *v* :$\sqsubset$*val v3a* $\vee$ (*v* = *v3a* $\wedge$ *Stars vs* :$\sqsubseteq$*val Stars vs3*)
unfolding *PosOrd-ex-eq-def* **by** *auto*
then have *Stars* (*v* # *vs*) :$\sqsubseteq$*val Stars* (*v3a* # *vs3*) **using** *NonEmpty*(*4*) *q2 as1*
unfolding *PosOrd-ex-eq-def*
by (*metis PosOrd-StarsI PosOrd-StarsI2 flat.simps*(*7*) *flat-Stars val.inject*(*5*))
then show *Stars* (*v* # *vs*) :$\sqsubseteq$*val v3* **unfolding** *NonEmpty* **by** *blast*
next
case *Empty*
have *v3* = *Stars* [] **by** *fact*
then show *Stars* (*v* # *vs*) :$\sqsubseteq$*val v3*
unfolding *PosOrd-ex-eq-def* **using** *cond2*
by (*simp add*: *PosOrd-shorterI*)
qed
next
case (*Posix-Rec s r v l v2*)
then show *Recv l v* :$\sqsubseteq$*val v2*
by (*smt* (*verit*, *del-insts*) *LV-def LV-simps*(*6*) *PosOrd-Rec-eq PosOrd-ex-eq-def Posix1*(*2*) *mem-Collect-eq*)
next
case (*Posix-Cset c cs v*)
have *v* $\in$ *LV* (*Charset cs*) [*c*] **by** *fact*
then have *v* = *Atm c* $\vee$ *False*
apply(*case-tac c* $\in$ *cs*)
by(*auto simp add*: *LV-simps*)
then show *Atm c* :$\sqsubseteq$*val v*
by (*simp add*: *PosOrd-ex-eq-def*)
qed

lemma *Posix-PosOrd-reverse*:
assumes *s* $\in$ *r* $\rightarrow$ *v1*
shows $\neg$($\exists$ *v2* $\in$ *LV r s*. *v2* :$\sqsubset$*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 $v1 \in LV\ r\ s$ $\forall v_2 \in LV\ r\ s.\ \neg\ v_2 :\sqsubset val\ v1$
  shows $s \in r \to v1$
proof −
  have $s \in lang\ r$ **using** *assms*(*1*) **unfolding** *LV-def*
    using *Prf-flat-lang* **by** *blast*
  then obtain *vposix* **where** *vp*: $s \in r \to vposix$
    using *lexer-correct-Some* **by** *blast*
  with *assms*(*1*) **have** $vposix :\sqsubseteq val\ v1$ **by** (*simp add*: *Posix-PosOrd*)
  then have $vposix = v1 \lor vposix :\sqsubset val\ v1$ **unfolding** *PosOrd-ex-eq2* **by** *auto*
  moreover
    { assume $vposix :\sqsubset val\ v1$
      moreover
      have $vposix \in LV\ r\ s$ **using** *vp*
        using *Posix-LV* **by** *blast*
      ultimately have *False* **using** *assms*(*2*) **by** *blast*
    }
  ultimately show $s \in r \to v1$ **using** *vp* **by** *blast*
qed

lemma *Least-existence*:
  assumes $LV\ r\ s \neq \{\}$
  shows $\exists\, vmin \in LV\ r\ s.\ \forall\, v \in LV\ r\ s.\ vmin :\sqsubseteq val\ v$
proof −
  from *assms*
  obtain *vposix* **where** $s \in r \to vposix$
  unfolding *LV-def*
  using *Prf-flat-lang lexer-correct-Some* **by** *blast*
  then have $\forall\, v \in LV\ r\ s.\ vposix :\sqsubseteq val\ v$
    by (*simp add*: *Posix-PosOrd*)
  then show $\exists\, vmin \in LV\ r\ s.\ \forall\, v \in LV\ r\ s.\ vmin :\sqsubseteq val\ v$
    using *Posix-LV* $\langle s \in r \to vposix \rangle$ **by** *blast*
qed

lemma *Least-existence1*:
  assumes $LV\ r\ s \neq \{\}$
  shows $\exists!\ v_{min} \in LV\ r\ s.\ \forall\, v \in LV\ r\ s.\ v_{min} :\sqsubseteq val\ v$
using *Least-existence*[*OF assms*] *assms*
using *PosOrdeq-antisym* **by** *blast*

lemma *Least-existence2*:
  assumes $LV\ r\ s \neq \{\}$
  shows $\exists! vmin \in LV\ r\ s.\ lexer\ r\ s = Some\ vmin \land (\forall\, v \in LV\ r\ s.\ vmin :\sqsubseteq val\ v)$
using *Least-existence*[*OF assms*] *assms*
using *PosOrdeq-antisym*
using *PosOrd-Posix PosOrd-ex-eq2 lexer-correctness*(*1*)
  by (*metis* (*mono-tags*, *lifting*))

lemma *Least-existence1-pre*:
assumes $LV\ r\ s \neq \{\}$
shows $\exists!vmin \in LV\ r\ s.\ \forall v \in (LV\ r\ s \cup \{v'.\ flat\ v' \sqsubset spre\ s\}).\ vmin :\sqsubseteq val\ v$
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 *PosOrd-partial*:
shows *partial-order-on UNIV* $\{(v1, v2).\ v1 :\sqsubseteq val\ v2\}$
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 *PosOrd-wf*:
shows *wf* $\{(v1, v2).\ v1 :\sqsubset val\ v2 \land v1 \in LV\ r\ s \land v2 \in LV\ r\ s\}$
proof −
have *finite* $\{(v1, v2).\ v1 \in LV\ r\ s \land v2 \in LV\ r\ s\}$
by (*simp add*: *LV-finite*)
moreover
have $\{(v1, v2).\ v1 :\sqsubset val\ v2 \land v1 \in LV\ r\ s \land v2 \in LV\ r\ s\} \subseteq \{(v1, v2).\ v1 \in LV\ r\ s \land v2 \in LV\ r\ s\}$
by *auto*
ultimately have *finite* $\{(v1, v2).\ v1 :\sqsubset val\ v2 \land v1 \in LV\ r\ s \land v2 \in LV\ r\ s\}$
using *finite-subset* **by** *blast*
moreover
have *acyclicP* $(\lambda v1\ v2.\ v1 :\sqsubset val\ v2 \land v1 \in LV\ r\ s \land v2 \in LV\ r\ s)$
unfolding *acyclic-def*
by (*smt* (*verit*, *ccfv-threshold*) *PosOrd-irrefl PosOrd-trans tranclp-trans-induct tranclp-unfold*)
ultimately show *wf* $\{(v1, v2).\ v1 :\sqsubset val\ v2 \land v1 \in LV\ r\ s \land v2 \in LV\ r\ s\}$
using *finite-acyclic-wf* **by** *blast*
qed

unused-thms

end

References

[1] 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.

[2] 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.