Decision Procedures for MSO on Words Based on Derivatives of Regular Expressions

Dmitriy Traytel and Tobias Nipkow

June 11, 2019

Abstract

Monadic second-order logic on finite words (MSO) is a decidable yet expressive logic into which many decision problems can be encoded. Since MSO formulas correspond to regular languages, equivalence of MSO formulas can be reduced to the equivalence of some regular structures (e.g. automata). We verify an executable decision procedure for MSO formulas that is not based on automata but on regular expressions.

Decision procedures for regular expression equivalence have been formalized before (e.g. in Isabelle/HOL [1]), usually based on Brzozowski derivatives. Yet, for a straightforward embedding of MSO formulas into regular expressions an extension of regular expressions with a projection operation is required. We prove total correctness and completeness of an equivalence checker for regular expressions extended in that way. We also define a language-preserving translation of formulas into regular expressions with respect to two different semantics of MSO.

The formalization is described in the ICFP 2013 functional pearl [2].

Contents

1 Regular Sets

type-synonym *'a lang = 'a list set*

definition *conc :: 'a lang ⇒ 'a lang ⇒ 'a lang* (**infixr** *@@ 75*) **where**
A @@ B = {xs@ys | xs ys. xs:A & ys:B}

lemma [*code*]:
A @@ B = (%(xs, ys). xs @ ys) ‘ (A × B)
unfolding *conc-def* **by** *auto*

overloading *word-pow == compow :: nat ⇒ 'a list ⇒ 'a list*
begin
primrec *word-pow :: nat ⇒ 'a list ⇒ 'a list* **where**
word-pow 0 w = [] |
word-pow (Suc n) w = w @ word-pow n w
end

overloading *lang-pow == compow :: nat ⇒ 'a lang ⇒ 'a lang*
begin
primrec *lang-pow :: nat ⇒ 'a lang ⇒ 'a lang* **where**
lang-pow 0 A = {[]} |
lang-pow (Suc n) A = A @@ (lang-pow n A)
end

lemma *word-pow-alt*: *compow n w = concat (replicate n w)*
by (*induct n*) *auto*

definition *star :: 'a lang ⇒ 'a lang* **where**
star A = (⋃n. A ^^ n)

1.1 Concatenation of Languages

lemma *concI*[*simp*,*intro*]: *u : A ⟹ v : B ⟹ u@v : A @@ B*
by (*auto simp add*: *conc-def*)

lemma *concE*[*elim*]:
assumes *w ∈ A @@ B*
obtains *u v* **where** *u ∈ A v ∈ B w = u@v*
using *assms* **by** (*auto simp*: *conc-def*)

lemma *conc-mono*: *A ⊆ C ⟹ B ⊆ D ⟹ A @@ B ⊆ C @@ D*
by (*auto simp*: *conc-def*)

lemma *conc-empty*[*simp*]: **shows** *{} @@ A = {}* **and** *A @@ {} = {}*
by *auto*

lemma *conc-epsilon*[*simp*]: **shows** *{[]} @@ A = A* **and** *A @@ {[]} = A*
by (*simp-all add*:*conc-def*)

lemma *conc-assoc*: $(A \mathbin{@@} B) \mathbin{@@} C = A \mathbin{@@} (B \mathbin{@@} C)$
by (*auto elim*!: *concE*) (*simp only*: *append-assoc*[*symmetric*] *concI*)

lemma *conc-Un-distrib*:
shows $A \mathbin{@@} (B \cup C) = A \mathbin{@@} B \cup A \mathbin{@@} C$
and $(A \cup B) \mathbin{@@} C = A \mathbin{@@} C \cup B \mathbin{@@} C$
by *auto*

lemma *conc-UNION-distrib*:
shows $A \mathbin{@@} \bigcup(M \mathbin{`} I) = \bigcup((\%i.\ A \mathbin{@@} M\ i) \mathbin{`} I)$
and $\bigcup(M \mathbin{`} I) \mathbin{@@} A = \bigcup((\%i.\ M\ i \mathbin{@@} A) \mathbin{`} I)$
by *auto*

lemma *hom-image-conc*: $[\![\bigwedge xs\ ys.\ f\ (xs \mathbin{@} ys) = f\ xs \mathbin{@} f\ ys]\!] \Longrightarrow f \mathbin{`} (A \mathbin{@@} B) = f \mathbin{`} A \mathbin{@@} f \mathbin{`} B$
unfolding *conc-def* **by** (*auto simp*: *image-iff*) *metis*

lemma *map-image-conc*[*simp*]: *map f* ‘ $(A \mathbin{@@} B) =$ *map f* ‘ $A \mathbin{@@}$ *map f* ‘ B
by (*simp add*: *hom-image-conc*)

lemma *conc-subset-lists*: $A \subseteq$ *lists* $S \Longrightarrow B \subseteq$ *lists* $S \Longrightarrow A \mathbin{@@} B \subseteq$ *lists* S
by(*fastforce simp*: *conc-def in-lists-conv-set*)

1.2 Iteration of Languages

lemma *lang-pow-add*: $A \mathbin{\hat{}\hat{}} (n + m) = A \mathbin{\hat{}\hat{}} n \mathbin{@@} A \mathbin{\hat{}\hat{}} m$
by (*induct n*) (*auto simp*: *conc-assoc*)

lemma *lang-pow-simps*: $(A \mathbin{\hat{}\hat{}} Suc\ n) = (A \mathbin{\hat{}\hat{}} n \mathbin{@@} A)$
using *lang-pow-add*[*of n Suc 0 A*] **by** *auto*

lemma *lang-pow-empty*: $\{\} \mathbin{\hat{}\hat{}} n =$ (*if* $n = 0$ *then* $\{[]\}$ *else* $\{\}$)
by (*induct n*) *auto*

lemma *lang-pow-empty-Suc*[*simp*]: $(\{\}::'a\ lang) \mathbin{\hat{}\hat{}} Suc\ n = \{\}$
by (*simp add*: *lang-pow-empty*)

lemma *conc-pow-comm*:
shows $A \mathbin{@@} (A \mathbin{\hat{}\hat{}} n) = (A \mathbin{\hat{}\hat{}} n) \mathbin{@@} A$
by (*induct n*) (*simp-all add*: *conc-assoc*[*symmetric*])

lemma *length-lang-pow-ub*:
$ALL\ w : A.\ length\ w \leq k \Longrightarrow w : A \mathbin{\hat{}\hat{}} n \Longrightarrow length\ w \leq k*n$
by(*induct n arbitrary*: *w*) (*fastforce simp*: *conc-def*)+

lemma *length-lang-pow-lb*:
$ALL\ w : A.\ length\ w \geq k \Longrightarrow w : A \mathbin{\hat{}\hat{}} n \Longrightarrow length\ w \geq k*n$
by(*induct n arbitrary*: *w*) (*fastforce simp*: *conc-def*)+

lemma *lang-pow-subset-lists*: $A \subseteq lists\ S \Longrightarrow A\ \hat{}\hat{}\ n \subseteq lists\ S$
by (*induct n*) (*auto simp*: *conc-subset-lists*)

lemma *star-subset-lists*: $A \subseteq lists\ S \Longrightarrow star\ A \subseteq lists\ S$
unfolding *star-def* **by**(*blast dest*: *lang-pow-subset-lists*)

lemma *star-if-lang-pow*[*simp*]: $w : A\ \hat{}\hat{}\ n \Longrightarrow w : star\ A$
by (*auto simp*: *star-def*)

lemma *Nil-in-star*[*iff*]: $[] : star\ A$
proof (*rule star-if-lang-pow*)
show $[] : A\ \hat{}\hat{}\ 0$ **by** *simp*
qed

lemma *star-if-lang*[*simp*]: **assumes** $w : A$ **shows** $w : star\ A$
proof (*rule star-if-lang-pow*)
show $w : A\ \hat{}\hat{}\ 1$ **using** ‹$w : A$› **by** *simp*
qed

lemma *append-in-starI*[*simp*]:
assumes $u : star\ A$ **and** $v : star\ A$ **shows** $u@v : star\ A$
proof −
from ‹$u : star\ A$› **obtain** m **where** $u : A\ \hat{}\hat{}\ m$ **by** (*auto simp*: *star-def*)
moreover
from ‹$v : star\ A$› **obtain** n **where** $v : A\ \hat{}\hat{}\ n$ **by** (*auto simp*: *star-def*)
ultimately have $u@v : A\ \hat{}\hat{}\ (m+n)$ **by** (*simp add*: *lang-pow-add*)
thus *?thesis* **by** *simp*
qed

lemma *conc-star-star*: $star\ A\ @@\ star\ A = star\ A$
by (*auto simp*: *conc-def*)

lemma *conc-star-comm*:
shows $A\ @@\ star\ A = star\ A\ @@\ A$
unfolding *star-def conc-pow-comm conc-UNION-distrib*
by *simp*

lemma *star-induct*[*consumes 1*, *case-names Nil append*, *induct set*: *star*]:
assumes $w : star\ A$
and $P\ []$
and *step*: $!!u\ v.\ u : A \Longrightarrow v : star\ A \Longrightarrow P\ v \Longrightarrow P\ (u@v)$
shows $P\ w$
proof −
{ fix n **have** $w : A\ \hat{}\hat{}\ n \Longrightarrow P\ w$
by (*induct n arbitrary*: *w*) (*auto intro*: ‹$P\ []$› *step star-if-lang-pow*) **}**
with ‹$w : star\ A$› **show** $P\ w$ **by** (*auto simp*: *star-def*)
qed

lemma *star-empty*[*simp*]: *star* {} = {[]}
by (*auto elim*: *star-induct*)

lemma *star-epsilon*[*simp*]: *star* {[]} = {[]}
by (*auto elim*: *star-induct*)

lemma *star-idemp*[*simp*]: *star* (*star A*) = *star A*
by (*auto elim*: *star-induct*)

lemma *star-unfold-left*: *star A* = *A* @@ *star A* ∪ {[]} (**is** *?L* = *?R*)
proof
show *?L* ⊆ *?R* **by** (*rule*, *erule star-induct*) *auto*
qed *auto*

lemma *concat-in-star*: *set ws* ⊆ *A* ⟹ *concat ws* : *star A*
by (*induct ws*) *simp-all*

lemma *in-star-iff-concat*:
w : *star A* = (*EX ws*. *set ws* ⊆ *A* & *w* = *concat ws* & [] ∉ *set ws*)
(**is** - = (*EX ws*. *?R w ws*))
proof
assume *w* : *star A* **thus** *EX ws*. *?R w ws*
proof *induct*
case *Nil* **have** *?R* [] [] **by** *simp*
thus *?case* **..**
next
case (*append u v*)
moreover
then obtain *ws* **where** *set ws* ⊆ *A* ∧ *v* = *concat ws* ∧ [] ∉ *set ws* **by** *blast*
ultimately have *?R* (*u*@*v*) (*if u* = [] *then ws else u*#*ws*) **by** *auto*
thus *?case* **..**
qed
next
assume *EX us*. *?R w us* **thus** *w* : *star A*
by (*auto simp*: *concat-in-star*)
qed

lemma *star-conv-concat*: *star A* = {*concat ws*|*ws*. *set ws* ⊆ *A* & [] ∉ *set ws*}
by (*fastforce simp*: *in-star-iff-concat*)

lemma *star-insert-eps*[*simp*]: *star* (*insert* [] *A*) = *star*(*A*)
proof−
{ fix *us*
have *set us* ⊆ *insert* [] *A* ⟹ *EX vs*. *concat us* = *concat vs* ∧ *set vs* ⊆ *A*
(**is** *?P* ⟹ *EX vs*. *?Q vs*)
proof
let *?vs* = *filter* (%*u*. *u* ≠ []) *us*
show *?P* ⟹ *?Q ?vs* **by** (*induct us*) *auto*
qed

} thus *?thesis* **by** (*auto simp*: *star-conv-concat*)
qed

lemma *star-decom*:
assumes *a*: $x \in star\ A\ x \neq []$
shows $\exists\, a\ b.\ x = a\ @\ b \wedge a \neq [] \wedge a \in A \wedge b \in star\ A$
using *a* **by** (*induct rule*: *star-induct*) (*blast*)+

lemma *Ball-starI*: $\forall\, a \in set\ as.\ [a] \in A \Longrightarrow as \in star\ A$
by (*induct as rule*: *rev-induct*) *auto*

lemma *map-image-star*[*simp*]: *map f ‘ star A* = *star* (*map f ‘ A*)
by (*auto elim*: *star-induct*) (*auto elim*: *star-induct simp del*: *map-append simp*: *map-append*[*symmetric*] *intro*!: *imageI*)

1.3 Left-Quotients of Languages

definition *lQuot* :: *'a* ⇒ *'a lang* ⇒ *'a lang*
where *lQuot x A* = { *xs*. *x*#*xs* ∈ *A* }

definition *lQuots* :: *'a list* ⇒ *'a lang* ⇒ *'a lang*
where *lQuots xs A* = { *ys*. *xs* @ *ys* ∈ *A* }

abbreviation
lQuotss :: *'a list* ⇒ *'a lang set* ⇒ *'a lang*
where
lQuotss s As ≡ ⋃ (*lQuots s ‘ As*)

lemma *lQuot-empty*[*simp*]: *lQuot a* {} = {}
and *lQuot-epsilon*[*simp*]: *lQuot a* {[]} = {}
and *lQuot-char*[*simp*]: *lQuot a* {[*b*]} = (*if a* = *b then* {[]} *else* {})
and *lQuot-chars*[*simp*]: *lQuot a* {[*b*] | *b*. *P b*} = (*if P a then* {[]} *else* {})
and *lQuot-union*[*simp*]: *lQuot a* (*A* ∪ *B*) = *lQuot a A* ∪ *lQuot a B*
and *lQuot-inter*[*simp*]: *lQuot a* (*A* ∩ *B*) = *lQuot a A* ∩ *lQuot a B*
and *lQuot-compl*[*simp*]: *lQuot a* (−*A*) = − *lQuot a A*
by (*auto simp*: *lQuot-def*)

lemma *lQuot-conc-subset*: *lQuot a A* @@ *B* ⊆ *lQuot a* (*A* @@ *B*) (**is** *?L* ⊆ *?R*)
proof
fix *w* **assume** *w* ∈ *?L*
then obtain *u v* **where** *w* = *u* @ *v a* # *u* ∈ *A v* ∈ *B*
by (*auto simp*: *lQuot-def*)
then have *a* # *w* ∈ *A* @@ *B*
by (*auto intro*: *concI*[*of a* # *u*, *simplified*])
thus *w* ∈ *?R* **by** (*auto simp*: *lQuot-def*)
qed

lemma *lQuot-conc* [*simp*]: *lQuot c* (*A* @@ *B*) = (*lQuot c A*) @@ *B* ∪ (*if* [] ∈ *A*

then lQuot c B else {})
unfolding *lQuot-def conc-def*
by (*auto simp add*: *Cons-eq-append-conv*)

lemma *lQuot-star* [*simp*]: *lQuot c* (*star A*) = (*lQuot c A*) @@ *star A*
proof −
have *incl*: [] ∈ *A* ⟹ *lQuot c* (*star A*) ⊆ (*lQuot c A*) @@ *star A*
unfolding *lQuot-def conc-def*
apply(*auto simp add*: *Cons-eq-append-conv*)
apply(*drule star-decom*)
apply(*auto simp add*: *Cons-eq-append-conv*)
done

have *lQuot c* (*star A*) = *lQuot c* (*A* @@ *star A* ∪ {[]})
by (*simp only*: *star-unfold-left*[*symmetric*])
also have ... = *lQuot c* (*A* @@ *star A*)
by (*simp only*: *lQuot-union*) (*simp*)
also have ... = (*lQuot c A*) @@ (*star A*) ∪ (*if* [] ∈ *A then lQuot c* (*star A*) *else* {})
by *simp*
also have ... = (*lQuot c A*) @@ *star A*
using *incl* **by** *auto*
finally show *lQuot c* (*star A*) = (*lQuot c A*) @@ *star A* **.**
qed

lemma *lQuot-diff*[*simp*]: *lQuot c* (*A* − *B*) = *lQuot c A* − *lQuot c B*
by(*auto simp add*: *lQuot-def*)

lemma *lQuot-lists*[*simp*]: *c* : *S* ⟹ *lQuot c* (*lists S*) = *lists S*
by(*auto simp add*: *lQuot-def*)

lemma *lQuots-simps* [*simp*]:
shows *lQuots* [] *A* = *A*
and *lQuots* (*c* # *s*) *A* = *lQuots s* (*lQuot c A*)
and *lQuots* (*s1* @ *s2*) *A* = *lQuots s2* (*lQuots s1 A*)
unfolding *lQuots-def lQuot-def* **by** *auto*

lemma *lQuots-append*[*iff*]: *v* ∈ *lQuots w A* ⟷ *w* @ *v* ∈ *A*
by (*induct w arbitrary*: *v A*) (*auto simp add*: *lQuot-def*)

1.4 Right-Quotients of Languages

definition *rQuot* :: *'a* ⇒ *'a lang* ⇒ *'a lang*
where *rQuot x A* = { *xs*. *xs* @ [*x*] ∈ *A* }

definition *rQuots* :: *'a list* ⇒ *'a lang* ⇒ *'a lang*
where *rQuots xs A* = { *ys*. *ys* @ *rev xs* ∈ *A* }

abbreviation

```
  rQuotss :: 'a list ⇒ 'a lang set ⇒ 'a lang
where
  rQuotss s As ≡ ⋃ (rQuots s ‘ As)

lemma rQuot-rev-lQuot: rQuot x A = rev ‘ lQuot x (rev ‘ A)
  unfolding rQuot-def lQuot-def by (auto simp: rev-swap[symmetric])

lemma rQuots-rev-lQuots: rQuots x A = rev ‘ lQuots x (rev ‘ A)
  unfolding rQuots-def lQuots-def by (auto simp: rev-swap[symmetric])

lemma rQuot-empty[simp]:   rQuot a {} = {}
  and rQuot-epsilon[simp]: rQuot a {[]} = {}
  and rQuot-char[simp]:    rQuot a {[b]} = (if a = b then {[]} else {})
  and rQuot-union[simp]:   rQuot a (A ∪ B) = rQuot a A ∪ rQuot a B
  and rQuot-inter[simp]:   rQuot a (A ∩ B) = rQuot a A ∩ rQuot a B
  and rQuot-compl[simp]:   rQuot a (−A) = − rQuot a A
  by (auto simp: rQuot-def)

lemma lQuot-rQuot: lQuot a (rQuot b A) = rQuot b (lQuot a A)
  unfolding lQuot-def rQuot-def by auto

lemma rQuot-lQuot: rQuot a (lQuot b A) = lQuot b (rQuot a A)
  unfolding lQuot-def rQuot-def by auto

lemma rev-simp-invert: (xs @ [x] = rev zs) = (zs = x # rev xs)
  by (induct zs) auto

lemma rev-append-invert: (xs @ ys = rev zs) = (zs = rev ys @ rev xs)
  by (induct xs arbitrary: ys rule: rev-induct) auto

lemma image-rev-lists[simp]: rev ‘ lists S = lists S
proof (intro set-eqI)
  fix xs
  show xs ∈ rev ‘ lists S ⟷ xs ∈ lists S
  proof (induct xs rule: rev-induct)
    case (snoc x xs)
    thus ?case by (auto intro!: image-eqI[of - rev] simp: rev-simp-invert)
  qed simp
qed

lemma image-rev-conc[simp]: rev ‘ (A @@ B) = rev ‘ B @@ rev ‘ A
  by auto (auto simp: rev-append[symmetric] simp del: rev-append)

lemma image-rev-star[simp]: rev ‘ star A = star (rev ‘ A)
  by (auto elim: star-induct) (auto elim: star-induct simp: rev-append[symmetric]
simp del: rev-append)

lemma rQuot-conc [simp]: rQuot c (A @@ B) = A @@ (rQuot c B) ∪ (if [] ∈ B
then rQuot c A else {})
```

unfolding *rQuot-rev-lQuot* **by** (*auto simp*: *image-image image-Un*)

lemma *rQuot-star* [*simp*]: *rQuot c* (*star A*) = *star A* @@ (*rQuot c A*)
unfolding *rQuot-rev-lQuot* **by** (*auto simp*: *image-image*)

lemma *rQuot-diff*[*simp*]: *rQuot c* (*A* − *B*) = *rQuot c A* − *rQuot c B*
by(*auto simp add*: *rQuot-def*)

lemma *rQuot-lists*[*simp*]: *c* : *S* ⟹ *rQuot c* (*lists S*) = *lists S*
by(*auto simp add*: *rQuot-def*)

lemma *rQuots-simps* [*simp*]:
shows *rQuots* [] *A* = *A*
and *rQuots* (*c* # *s*) *A* = *rQuots s* (*rQuot c A*)
and *rQuots* (*s1* @ *s2*) *A* = *rQuots s2* (*rQuots s1 A*)
unfolding *rQuots-def rQuot-def* **by** *auto*

lemma *rQuots-append*[*iff*]: *v* ∈ *rQuots w A* ⟷ *v* @ *rev w* ∈ *A*
by (*induct w arbitrary*: *v A*) (*auto simp add*: *rQuot-def*)

1.5 Two-Sided-Quotients of Languages

definition *biQuot* :: *'a* ⇒ *'a* ⇒ *'a lang* ⇒ *'a lang*
where *biQuot x y A* = { *xs*. *x* # *xs* @ [*y*] ∈ *A* }

definition *biQuots* :: *'a list* ⇒ *'a list* ⇒ *'a lang* ⇒ *'a lang*
where *biQuots xs ys A* = { *zs*. *xs* @ *zs* @ *rev ys* ∈ *A* }

abbreviation
biQuotss :: *'a list* ⇒ *'a list* ⇒ *'a lang set* ⇒ *'a lang*
where
biQuotss xs ys As ≡ ⋃ (*biQuots xs ys* ‘ *As*)

lemma *biQuot-rQuot-lQuot*: *biQuot x y A* = *rQuot y* (*lQuot x A*)
unfolding *biQuot-def rQuot-def lQuot-def* **by** *auto*

lemma *biQuot-lQuot-rQuot*: *biQuot x y A* = *lQuot x* (*rQuot y A*)
unfolding *biQuot-def rQuot-def lQuot-def* **by** *auto*

lemma *biQuots-rQuots-lQuots*: *biQuots x y A* = *rQuots y* (*lQuots x A*)
unfolding *biQuots-def rQuots-def lQuots-def* **by** *auto*

lemma *biQuots-lQuots-rQuots*: *biQuots x y A* = *lQuots x* (*rQuots y A*)
unfolding *biQuots-def rQuots-def lQuots-def* **by** *auto*

lemma *biQuot-empty*[*simp*]: *biQuot a b* {} = {}
and *biQuot-epsilon*[*simp*]: *biQuot a b* {[]} = {}
and *biQuot-char*[*simp*]: *biQuot a b* {[*c*]} = {}
and *biQuot-union*[*simp*]: *biQuot a b* (*A* ∪ *B*) = *biQuot a b A* ∪ *biQuot a b B*

and *biQuot-inter*[*simp*]: $biQuot\ a\ b\ (A \cap B) = biQuot\ a\ b\ A \cap biQuot\ a\ b\ B$
and *biQuot-compl*[*simp*]: $biQuot\ a\ b\ (-A) = -\ biQuot\ a\ b\ A$
by (*auto simp*: *biQuot-def*)

lemma *biQuot-conc* [*simp*]: $biQuot\ a\ b\ (A\ @@\ B) =$
$lQuot\ a\ A\ @@\ rQuot\ b\ B \cup$
(*if* $[] \in A \wedge [] \in B$ *then* $biQuot\ a\ b\ A \cup biQuot\ a\ b\ B$
else if $[] \in A$ *then* $biQuot\ a\ b\ B$
else if $[] \in B$ *then* $biQuot\ a\ b\ A$
else $\{\}$)
unfolding *biQuot-rQuot-lQuot* **by** *auto*

lemma *biQuot-star* [*simp*]: $biQuot\ a\ b\ (star\ A) = biQuot\ a\ b\ A \cup lQuot\ a\ A\ @@\ star\ A\ @@\ rQuot\ b\ A$
unfolding *biQuot-rQuot-lQuot* **by** *auto*

lemma *biQuot-diff*[*simp*]: $biQuot\ a\ b\ (A - B) = biQuot\ a\ b\ A - biQuot\ a\ b\ B$
by(*auto simp add*: *biQuot-def*)

lemma *biQuot-lists*[*simp*]: $a : S \Longrightarrow b : S \Longrightarrow biQuot\ a\ b\ (lists\ S) = lists\ S$
by(*auto simp add*: *biQuot-def*)

lemma *biQuots-simps* [*simp*]:
shows $biQuots\ []\ []\ A = A$
and $biQuots\ (a\#as)\ (b\#bs)\ A = biQuots\ as\ bs\ (biQuot\ a\ b\ A)$
and $[\![length\ s1 = length\ t1;\ length\ s2 = length\ t2]\!] \Longrightarrow$
$biQuots\ (s1\ @\ s2)\ (t1\ @\ t2)\ A = biQuots\ s2\ t2\ (biQuots\ s1\ t1\ A)$
unfolding *biQuots-def biQuot-def* **by** *auto*

lemma *biQuots-append*[*iff*]: $v \in biQuots\ u\ w\ A \longleftrightarrow u\ @\ v\ @\ rev\ w \in A$
unfolding *biQuots-def* **by** *auto*

1.6 Arden's Lemma

lemma *arden-helper*:
assumes *eq*: $X = A\ @@\ X \cup B$
shows $X = (A\ \hat{}\hat{}\ Suc\ n)\ @@\ X \cup (\bigcup m{\leq}n.\ (A\ \hat{}\hat{}\ m)\ @@\ B)$
proof (*induct n*)
case *0*
show $X = (A\ \hat{}\hat{}\ Suc\ 0)\ @@\ X \cup (\bigcup m{\leq}0.\ (A\ \hat{}\hat{}\ m)\ @@\ B)$
using *eq* **by** *simp*
next
case (*Suc n*)
have *ih*: $X = (A\ \hat{}\hat{}\ Suc\ n)\ @@\ X \cup (\bigcup m{\leq}n.\ (A\ \hat{}\hat{}\ m)\ @@\ B)$ **by** *fact*
also have $\ldots = (A\ \hat{}\hat{}\ Suc\ n)\ @@\ (A\ @@\ X \cup B) \cup (\bigcup m{\leq}n.\ (A\ \hat{}\hat{}\ m)\ @@\ B)$
using *eq* **by** *simp*
also have $\ldots = (A\ \hat{}\hat{}\ Suc\ (Suc\ n))\ @@\ X \cup ((A\ \hat{}\hat{}\ Suc\ n)\ @@\ B) \cup (\bigcup m{\leq}n.\ (A\ \hat{}\hat{}\ m)\ @@\ B)$
by (*simp add*: *conc-Un-distrib conc-assoc*[*symmetric*] *conc-pow-comm*)

also have $\ldots = (A \hat{\ }\hat{\ } Suc\ (Suc\ n))\ @@\ X \cup (\bigcup m{\leq}Suc\ n.\ (A \hat{\ }\hat{\ } m)\ @@\ B)$
by (*auto simp add*: *atMost-Suc*)
finally show $X = (A \hat{\ }\hat{\ } Suc\ (Suc\ n))\ @@\ X \cup (\bigcup m{\leq}Suc\ n.\ (A \hat{\ }\hat{\ } m)\ @@\ B)$
.
qed

lemma *Arden*:
assumes $[] \notin A$
shows $X = A\ @@\ X \cup B \longleftrightarrow X = star\ A\ @@\ B$
proof
assume *eq*: $X = A\ @@\ X \cup B$
{ fix w **assume** $w : X$
let *?n* = *size w*
from ‹$[] \notin A$› **have** *ALL u* : *A*. *length u* $\geq$ *1*
by (*metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq*)
hence *ALL u* : *A*^^(*?n+1*). *length u* $\geq$ *?n+1*
by (*metis length-lang-pow-lb nat-mult-1*)
hence *ALL u* : *A*^^(*?n+1*)@@*X*. *length u* $\geq$ *?n+1*
by(*auto simp only*: *conc-def length-append*)
hence $w \notin A \hat{\ }\hat{\ } (?n+1)@@X$ **by** *auto*
hence w : *star A* @@ *B* **using** ‹$w : X$› **using** *arden-helper*[*OF eq*, **where** *n*=*?n*]
by (*auto simp add*: *star-def conc-UNION-distrib*)
} moreover
{ fix w **assume** w : *star A* @@ *B*
hence *EX n*. $w : A \hat{\ }\hat{\ } n\ @@\ B$ **by**(*auto simp*: *conc-def star-def*)
hence $w : X$ **using** *arden-helper*[*OF eq*] **by** *blast*
} ultimately show $X = star\ A\ @@\ B$ **by** *blast*
next
assume *eq*: $X = star\ A\ @@\ B$
have $star\ A = A\ @@\ star\ A \cup \{[]\}$
by (*rule star-unfold-left*)
then have $star\ A\ @@\ B = (A\ @@\ star\ A \cup \{[]\})\ @@\ B$
by *metis*
also have $\ldots = (A\ @@\ star\ A)\ @@\ B \cup B$
unfolding *conc-Un-distrib* **by** *simp*
also have $\ldots = A\ @@\ (star\ A\ @@\ B) \cup B$
by (*simp only*: *conc-assoc*)
finally show $X = A\ @@\ X \cup B$
using *eq* **by** *blast*
qed

lemma *reversed-arden-helper*:
assumes *eq*: $X = X\ @@\ A \cup B$
shows $X = X\ @@\ (A \hat{\ }\hat{\ } Suc\ n) \cup (\bigcup m{\leq}n.\ B\ @@\ (A \hat{\ }\hat{\ } m))$
proof (*induct n*)
case *0*
show $X = X\ @@\ (A \hat{\ }\hat{\ } Suc\ 0) \cup (\bigcup m{\leq}0.\ B\ @@\ (A \hat{\ }\hat{\ } m))$

```
    using eq by simp
next
  case (Suc n)
  have ih: X = X @@ (A ^^ Suc n) ∪ (⋃m≤n. B @@ (A ^^ m)) by fact
  also have ... = (X @@ A ∪ B) @@ (A ^^ Suc n) ∪ (⋃m≤n. B @@ (A ^^ m))
using eq by simp
  also have ... = X @@ (A ^^ Suc (Suc n)) ∪ (B @@ (A ^^ Suc n)) ∪ (⋃m≤n.
B @@ (A ^^ m))
    by (simp add: conc-Un-distrib conc-assoc)
  also have ... = X @@ (A ^^ Suc (Suc n)) ∪ (⋃m≤Suc n. B @@ (A ^^ m))
    by (auto simp add: atMost-Suc)
  finally show X = X @@ (A ^^ Suc (Suc n)) ∪ (⋃m≤Suc n. B @@ (A ^^ m))
.
qed

theorem reversed-Arden:
  assumes nemp: [] ∉ A
  shows X = X @@ A ∪ B ⟷ X = B @@ star A
proof
 assume eq: X = X @@ A ∪ B
  { fix w assume w : X
    let ?n = size w
    from ‹[] ∉ A› have ALL u : A. length u ≥ 1
      by (metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq)
    hence ALL u : A^^(?n+1). length u ≥ ?n+1
      by (metis length-lang-pow-lb nat-mult-1)
    hence ALL u : X @@ A^^(?n+1). length u ≥ ?n+1
      by(auto simp only: conc-def length-append)
    hence w ∉ X @@ A^^(?n+1) by auto
     hence w : B @@ star A using ‹w : X› using reversed-arden-helper[OF eq,
where n=?n]
      by (auto simp add: star-def conc-UNION-distrib)
  } moreover
  { fix w assume w : B @@ star A
    hence EX n. w : B @@ A^^n by (auto simp: conc-def star-def)
    hence w : X using reversed-arden-helper[OF eq] by blast
  } ultimately show X = B @@ star A by blast
next
  assume eq: X = B @@ star A
  have star A = {[]} ∪ star A @@ A
    unfolding conc-star-comm[symmetric]
    by(metis Un-commute star-unfold-left)
  then have B @@ star A = B @@ ({[]} ∪ star A @@ A)
    by metis
  also have ... = B ∪ B @@ (star A @@ A)
    unfolding conc-Un-distrib by simp
  also have ... = B ∪ (B @@ star A) @@ A
    by (simp only: conc-assoc)
  finally show X = X @@ A ∪ B
```

using *eq* **by** *blast*
qed

1.7 Lists of Fixed Length

abbreviation *listsN* **where** *listsN n S* ≡ {*xs*. *xs* ∈ *lists S* ∧ *length xs* = *n*}

lemma *tl-listsN*: *A* ⊆ *listsN* (*n* + *1*) *S* ⟹ *tl* ‘ *A* ⊆ *listsN n S*
proof (*intro image-subsetI*)
fix *xs* **assume** *A* ⊆ *listsN* (*n* + *1*) *S xs* ∈ *A*
thus *tl xs* ∈ *listsN n S* **by** (*induct xs*) *auto*
qed

lemma *map-tl-listsN*: *A* ⊆ *lists* (*listsN* (*n* + *1*) *S*) ⟹ *map tl* ‘ *A* ⊆ *lists* (*listsN n S*)
proof (*intro image-subsetI*)
fix *xss* **assume** *A* ⊆ *lists* (*listsN* (*n* + *1*) *S*) *xss* ∈ *A*
hence *set xss* ⊆ *listsN* (*n* + *1*) *S* **by** *auto*
hence ∀ *xs* ∈ *set xss*. *tl xs* ∈ *listsN n S* **using** *tl-listsN*[*of set xss S n*] **by** *auto*
thus *map tl xss* ∈ *lists* (*listsN n S*) **by** (*induct xss*) *auto*
qed

2 Π-Extended Regular Expressions

2.1 Syntax of regular expressions

datatype *'a rexp* =
Zero |
Full |
One |
Atom 'a |
Plus (*'a rexp*) (*'a rexp*) |
Times (*'a rexp*) (*'a rexp*) |
Star (*'a rexp*) |
Not (*'a rexp*) |
Inter (*'a rexp*) (*'a rexp*) |
Pr (*'a rexp*)
derive *linorder rexp*

Lifting constructors to lists

fun *rexp-of-list* **where**
rexp-of-list OPERATION N [] = *N*
| *rexp-of-list OPERATION N* [*x*] = *x*
| *rexp-of-list OPERATION N* (*x* # *xs*) = *OPERATION x* (*rexp-of-list OPERATION N xs*)

abbreviation *PLUS* ≡ *rexp-of-list Plus Zero*
abbreviation *TIMES* ≡ *rexp-of-list Times One*

abbreviation *INTERSECT* ≡ *rexp-of-list Inter Full*

lemma *list-singleton-induct* [*case-names nil single cons*]:
  assumes *nil*: *P* []
  assumes *single*: ⋀*x*. *P* [*x*]
  assumes *cons*: ⋀*x y xs*. *P* (*y* # *xs*) ⟹ *P* (*x* # (*y* # *xs*))
  shows *P xs*
  using *assms*
proof (*induct xs*)
  case (*Cons x xs*) **thus** *?case* **by** (*cases xs*) *auto*
qed *simp*

2.2 ACI normalization

fun *toplevel-summands* :: *'a rexp* ⇒ *'a rexp set* **where**
  toplevel-summands (*Plus r s*) = *toplevel-summands r* ∪ *toplevel-summands s*
| *toplevel-summands r* = {*r*}

abbreviation (*input*) *flatten LISTOP X* ≡ *LISTOP* (*sorted-list-of-set X*)

lemma *toplevel-summands-nonempty*[*simp*]:
  toplevel-summands r ≠ {}
  by (*induct r*) *auto*

lemma *toplevel-summands-finite*[*simp*]:
  finite (*toplevel-summands r*)
  by (*induct r*) *auto*

primrec *ACI-norm* :: (*'a*::*linorder*) *rexp* ⇒ *'a rexp* («-») **where**
  «*Zero*» = *Zero*
| «*Full*» = *Full*
| «*One*» = *One*
| «*Atom a*» = *Atom a*
| «*Plus r s*» = *flatten PLUS* (*toplevel-summands* (*Plus* «*r*» «*s*»))
| «*Times r s*» = *Times* «*r*» «*s*»
| «*Star r*» = *Star* «*r*»
| «*Not r*» = *Not* «*r*»
| «*Inter r s*» = *Inter* «*r*» «*s*»
| «*Pr r*» = *Pr* «*r*»

lemma *Plus-toplevel-summands*:
  Plus r s ∈ *toplevel-summands t* ⟹ *False*
  by (*induct t*) *auto*

lemma *toplevel-summands-not-Plus*[*simp*]:
  (∀ *r s*. *x* ≠ *Plus r s*) ⟹ *toplevel-summands x* = {*x*}
  by (*induct x*) *auto*

lemma *toplevel-summands-PLUS-strong*:

⟦*xs* ≠ []; *list-all* (λ*x*. ¬(∃ *r s*. *x* = *Plus r s*)) *xs*⟧ ⟹ *toplevel-summands* (*PLUS xs*) = *set xs*
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *toplevel-summands-flatten*:
⟦*X* ≠ {}; *finite X*; ∀ *x* ∈ *X*. ¬(∃ *r s*. *x* = *Plus r s*)⟧ ⟹ *toplevel-summands* (*flatten PLUS X*) = *X*
using *toplevel-summands-PLUS-strong*[*of sorted-list-of-set X*]
unfolding *list-all-iff* **by** *fastforce*

lemma *ACI-norm-Plus*:
«*r*» = *Plus s t* ⟹ ∃ *s t*. *r* = *Plus s t*
by (*induct r*) *auto*

lemma *toplevel-summands-flatten-ACI-norm-image*:
toplevel-summands (*flatten PLUS* (*ACI-norm* ‘ *toplevel-summands r*)) = *ACI-norm* ‘ *toplevel-summands r*
by (*intro toplevel-summands-flatten*) (*auto dest*!: *ACI-norm-Plus intro*: *Plus-toplevel-summands*)

lemma *toplevel-summands-flatten-ACI-norm-image-Union*:
toplevel-summands (*flatten PLUS* (*ACI-norm* ‘ *toplevel-summands r* ∪ *ACI-norm* ‘ *toplevel-summands s*)) =
ACI-norm ‘ *toplevel-summands r* ∪ *ACI-norm* ‘ *toplevel-summands s*
by (*intro toplevel-summands-flatten*) (*auto dest*!: *ACI-norm-Plus*[*OF sym*] *intro*: *Plus-toplevel-summands*)

lemma *toplevel-summands-ACI-norm*:
toplevel-summands «*r*» = *ACI-norm* ‘ *toplevel-summands r*
by (*induct r*) (*auto simp*: *toplevel-summands-flatten-ACI-norm-image-Union*)

lemma *ACI-norm-flatten*:
«*r*» = *flatten PLUS* (*ACI-norm* ‘ *toplevel-summands r*)
by (*induct r*) (*auto simp*: *image-Un toplevel-summands-flatten-ACI-norm-image*)

theorem *ACI-norm-idem*[*simp*]:
««*r*»» = «*r*»
proof (*induct r*)
case (*Plus r s*)
have ««*Plus r s*»» = «*flatten PLUS* (*toplevel-summands* «*r*» ∪ *toplevel-summands* «*s*»)»
(**is** - = «*flatten PLUS ?U*») **by** *simp*
also have ... = *flatten PLUS* (*ACI-norm* ‘ *toplevel-summands* (*flatten PLUS ?U*))
by (*simp only*: *ACI-norm-flatten*)
also have *toplevel-summands* (*flatten PLUS ?U*) = *?U*
by (*intro toplevel-summands-flatten*) (*auto intro*: *Plus-toplevel-summands*)
also have *flatten PLUS* (*ACI-norm* ‘ *?U*) = *flatten PLUS* (*toplevel-summands* «*r*» ∪ *toplevel-summands* «*s*»)
by (*simp only*: *image-Un toplevel-summands-ACI-norm*[*symmetric*] *Plus*)

```
  finally show ?case by simp
qed auto

fun ACI-nPlus :: 'a::linorder rexp ⇒ 'a rexp ⇒ 'a rexp
where
  ACI-nPlus (Plus r1 r2) s = ACI-nPlus r1 (ACI-nPlus r2 s)
| ACI-nPlus r (Plus s1 s2) =
  (if r = s1 then Plus s1 s2
  else if r < s1 then Plus r (Plus s1 s2)
  else Plus s1 (ACI-nPlus r s2))
| ACI-nPlus r s =
  (if r = s then r
  else if r < s then Plus r s
  else Plus s r)

fun ACI-norm-alt where
  ACI-norm-alt Zero = Zero
| ACI-norm-alt Full = Full
| ACI-norm-alt One = One
| ACI-norm-alt (Atom a) = Atom a
| ACI-norm-alt (Plus r s) = ACI-nPlus (ACI-norm-alt r) (ACI-norm-alt s)
| ACI-norm-alt (Times r s) = Times (ACI-norm-alt r) (ACI-norm-alt s)
| ACI-norm-alt (Star r) = Star (ACI-norm-alt r)
| ACI-norm-alt (Not r) = Not (ACI-norm-alt r)
| ACI-norm-alt (Inter r s) = Inter (ACI-norm-alt r) (ACI-norm-alt s)
| ACI-norm-alt (Pr r) = Pr (ACI-norm-alt r)

lemma toplevel-summands-ACI-nPlus:
  toplevel-summands (ACI-nPlus r s) = toplevel-summands (Plus r s)
  by (induct r s rule: ACI-nPlus.induct) auto

lemma toplevel-summands-ACI-norm-alt:
  toplevel-summands (ACI-norm-alt r) = ACI-norm-alt ‘ toplevel-summands r
  by (induct r) (auto simp: toplevel-summands-ACI-nPlus)

lemma ACI-norm-alt-Plus:
  ACI-norm-alt r = Plus s t ⟹ ∃ s t. r = Plus s t
  by (induct r) auto

lemma toplevel-summands-flatten-ACI-norm-alt-image:
   toplevel-summands (flatten PLUS (ACI-norm-alt ‘ toplevel-summands r)) =
ACI-norm-alt ‘ toplevel-summands r
 by (intro toplevel-summands-flatten) (auto dest!: ACI-norm-alt-Plus intro: Plus-toplevel-summands)

lemma ACI-norm-ACI-norm-alt: ≪ACI-norm-alt r≫ = ≪r≫
proof (induction r)
  case (Plus r s) show ?case
    using ACI-norm-flatten [of ACI-norm-alt (Plus r s)] ACI-norm-flatten [of Plus
r s]
```

by (*auto simp*: *image-Un toplevel-summands-ACI-nPlus*)
(*metis Plus.IH toplevel-summands-ACI-norm*)
qed *auto*

lemma *ACI-nPlus-singleton-PLUS*:
⟦*xs* ≠ []; *sorted xs*; *distinct xs*; ∀ *x* ∈ {*x*} ∪ *set xs*. ¬(∃ *r s*. *x* = *Plus r s*)⟧ ⟹
ACI-nPlus x (*PLUS xs*) = (*if x* ∈ *set xs then PLUS xs else PLUS* (*insort x xs*))
proof (*induct xs rule*: *list-singleton-induct*)
case (*single y*)
thus *?case*
by (*cases x y rule*: *linorder-cases*) (*induct x y rule*: *ACI-nPlus.induct*, *auto*)+
next
case (*cons y1 y2 ys*) **thus** *?case* **by** (*cases x*) (*auto*)
qed *simp*

lemma *ACI-nPlus-PLUS*:
⟦*xs1* ≠ []; *xs2* ≠ []; ∀ *x* ∈ *set* (*xs1* @ *xs2*). ¬(∃ *r s*. *x* = *Plus r s*); *sorted xs2*; *distinct xs2*⟧⟹
ACI-nPlus (*PLUS xs1*) (*PLUS xs2*) = *flatten PLUS* (*set* (*xs1* @ *xs2*))
proof (*induct xs1 arbitrary*: *xs2 rule*: *list-singleton-induct*)
case (*single x1*) **thus** *?case*
apply (*auto intro*!: *trans*[*OF ACI-nPlus-singleton-PLUS*] *simp*: *insert-absorb simp del*: *sorted-list-of-set-insert*)
apply (*metis finite-set finite-sorted-distinct-unique sorted-list-of-set*)
apply (*metis remdups-id-iff-distinct sorted-list-of-set-sort-remdups sorted-sort-id*)
done
next
case (*cons x11 x12 xs1*) **thus** *?case*
apply (*simp del*: *sorted-list-of-set-insert*)
apply (*rule trans*[*OF ACI-nPlus-singleton-PLUS*])
apply (*auto simp del*: *sorted-list-of-set-insert simp add*: *insert-commute*[*of x11*])
apply (*auto simp only*: *Un-insert-left*[*of x11*, *symmetric*] *insert-absorb*) []
apply (*auto simp only*: *Un-insert-right*[*of - x11*, *symmetric*] *insert-absorb*) []
apply (*auto simp add*: *insert-commute*[*of x12*])
done
qed *simp*

lemma *ACI-nPlus-flatten-PLUS*:
⟦*X1* ≠ {}; *X2* ≠ {}; *finite X1*; *finite X2*; ∀ *x* ∈ *X1* ∪ *X2*. ¬(∃ *r s*. *x* = *Plus r s*)⟧⟹
ACI-nPlus (*flatten PLUS X1*) (*flatten PLUS X2*) = *flatten PLUS* (*X1* ∪ *X2*)
by (*rule trans*[*OF ACI-nPlus-PLUS*]) *auto*

lemma *ACI-nPlus-ACI-norm*[*simp*]: *ACI-nPlus* ≪*r*≫ ≪*s*≫ = ≪*Plus r s*≫
using *ACI-norm-flatten* [*of r*] *ACI-norm-flatten* [*of s*] *ACI-norm-flatten* [*of Plus r s*]
apply (*auto intro*!: *trans* [*OF ACI-nPlus-flatten-PLUS*])
apply (*auto simp*: *image-Un Un-assoc intro*!: *trans* [*OF ACI-nPlus-flatten-PLUS*])

apply (*metis ACI-norm-Plus Plus-toplevel-summands*)+
done

lemma *ACI-norm-alt*:
ACI-norm-alt r = ≪*r*≫
by (*induct r*) *auto*

declare *ACI-norm-alt*[*symmetric*, *code*]

2.3 Finality

primrec *final* :: *'a rexp* ⇒ *bool*
where
final Zero = *False*
| *final Full* = *True*
| *final One* = *True*
| *final* (*Atom* -) = *False*
| *final* (*Plus r s*) = (*final r* ∨ *final s*)
| *final* (*Times r s*) = (*final r* ∧ *final s*)
| *final* (*Star* -) = *True*
| *final* (*Not r*) = (~ *final r*)
| *final* (*Inter r1 r2*) = (*final r1* ∧ *final r2*)
| *final* (*Pr r*) = *final r*

lemma *toplevel-summands-final*:
final s = (∃ *r*∈*toplevel-summands s*. *final r*)
by (*induct s*) *auto*

lemma *final-PLUS*:
final (*PLUS xs*) = (∃ *r* ∈ *set xs*. *final r*)
by (*induct xs rule*: *list-singleton-induct*) *auto*

theorem *ACI-norm-final*[*simp*]:
final ≪*r*≫ = *final r*
proof (*induct r*)
case (*Plus r1 r2*) **thus** *?case* **using** *toplevel-summands-final* **by** (*auto simp*: *final-PLUS*)
qed *auto*

2.4 Wellformedness w.r.t. an alphabet

locale *alphabet* =
fixes Σ :: *nat* ⇒ *'a set* (Σ -)
and *wf-atom* :: *nat* ⇒ *'b* :: *linorder* ⇒ *bool*
begin

primrec *wf* :: *nat* ⇒ *'b rexp* ⇒ *bool*
where
wf n Zero = *True* |
wf n Full = *True* |

wf n One = True |
wf n (Atom a) = (wf-atom n a) |
wf n (Plus r s) = (wf n r ∧ wf n s) |
wf n (Times r s) = (wf n r ∧ wf n s) |
wf n (Star r) = wf n r |
wf n (Not r) = wf n r |
wf n (Inter r s) = (wf n r ∧ wf n s) |
wf n (Pr r) = wf (n + 1) r

primrec *wf-word* **where**
wf-word n [] = True
| *wf-word n (w # ws) = ((w ∈ Σ n) ∧ wf-word n ws)*

lemma *wf-word-snoc*[*simp*]: *wf-word n (ws @ [w]) = ((w ∈ Σ n) ∧ wf-word n ws)*
by (*induct ws*) *auto*

lemma *wf-word-append*[*simp*]: *wf-word n (ws @ vs) = (wf-word n ws ∧ wf-word n vs)*
by (*induct ws arbitrary*: *vs*) *auto*

lemma *wf-word*: *wf-word n w = (w ∈ lists (Σ n))*
by (*induct w*) *auto*

lemma *toplevel-summands-wf*:
wf n s = (∀ r∈toplevel-summands s. wf n r)
by (*induct s*) *auto*

lemma *wf-PLUS*[*simp*]:
wf n (PLUS xs) = (∀ r ∈ set xs. wf n r)
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *wf-TIMES*[*simp*]:
wf n (TIMES xs) = (∀ r ∈ set xs. wf n r)
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *wf-flatten-PLUS*[*simp*]:
finite X ⟹ wf n (flatten PLUS X) = (∀ r ∈ X. wf n r)
using *wf-PLUS*[*of n sorted-list-of-set X*] **by** *fastforce*

theorem *ACI-norm-wf*[*simp*]:
wf n ≪r≫ = wf n r
proof (*induct r arbitrary*: *n*)
case (*Plus r1 r2*) **thus** *?case*
using *toplevel-summands-wf*[*of n ≪r1≫*] *toplevel-summands-wf*[*of n ≪r2≫*] **by** *auto*
qed *auto*

lemma *wf-INTERSECT*[*simp*]:
wf n (INTERSECT xs) = (∀ r ∈ set xs. wf n r)

by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *wf-flatten-INTERSECT*[*simp*]:
finite X $\Longrightarrow$ *wf n* (*flatten INTERSECT X*) = ($\forall$ *r* $\in$ *X*. *wf n r*)
using *wf-INTERSECT*[*of n sorted-list-of-set X*] **by** *fastforce*

end

2.5 Language

locale *project* =
alphabet Σ *wf-atom* **for** Σ :: *nat* $\Rightarrow$ *'a set* **and** *wf-atom* :: *nat* $\Rightarrow$ *'b* :: *linorder* $\Rightarrow$ *bool* +
fixes *project* :: *'a* $\Rightarrow$ *'a*
and *lookup* :: *'b* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
assumes *project*: $\bigwedge$*a*. *a* $\in$ Σ (*Suc n*) $\Longrightarrow$ *project a* $\in$ Σ *n*
begin

primrec *lang* :: *nat* $\Rightarrow$ *'b rexp* => *'a lang* **where**
lang n Zero = {} |
lang n Full = *lists* (Σ *n*) |
lang n One = {[]} |
lang n (*Atom b*) = {[*x*] | *x*. *lookup b x* $\wedge$ *x* $\in$ Σ *n*} |
lang n (*Plus r s*) = (*lang n r*) $\cup$ (*lang n s*) |
lang n (*Times r s*) = *conc* (*lang n r*) (*lang n s*) |
lang n (*Star r*) = *star* (*lang n r*) |
lang n (*Not r*) = *lists* (Σ *n*) $-$ *lang n r* |
lang n (*Inter r s*) = (*lang n r* $\cap$ *lang n s*) |
lang n (*Pr r*) = *map project* ‘ *lang* (*n* + *1*) *r*

lemma *wf-word-map-project*[*simp*]: *wf-word* (*Suc n*) *ws* $\Longrightarrow$ *wf-word n* (*map project ws*)
by (*induct ws arbitrary*: *n*) (*auto intro*: *project*)

lemma *wf-lang-wf-word*: *wf n r* $\Longrightarrow$ $\forall$ *w* $\in$ *lang n r*. *wf-word n w*
by (*induct r arbitrary*: *n*) (*auto elim*: *rev-subsetD*[*OF* - *conc-mono*] *star-induct intro*: *iffD2*[*OF wf-word*])

lemma *lang-subset-lists*: *wf n r* $\Longrightarrow$ *lang n r* $\subseteq$ *lists* (Σ *n*)
proof (*induct r arbitrary*: *n*)
case *Pr* **thus** *?case* **by** (*fastforce intro*!: *project*)
qed (*auto simp*: *conc-subset-lists star-subset-lists*)

lemma *toplevel-summands-lang*:
r $\in$ *toplevel-summands s* $\Longrightarrow$ *lang n r* $\subseteq$ *lang n s*
by (*induct s*) *auto*

lemma *toplevel-summands-lang-UN*:
lang n s = ($\bigcup$ *r*$\in$*toplevel-summands s*. *lang n r*)

by (*induct s*) *auto*

lemma *toplevel-summands-in-lang*:
w ∈ *lang n s* = (∃ *r*∈*toplevel-summands s*. *w* ∈ *lang n r*)
by (*induct s*) *auto*

lemma *lang-PLUS*[*simp*]:
lang n (*PLUS xs*) = (⋃ *r* ∈ *set xs*. *lang n r*)
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *lang-TIMES*[*simp*]:
lang n (*TIMES xs*) = *foldr* (@@) (*map* (*lang n*) *xs*) {[]}
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *lang-flatten-PLUS*:
finite X ⟹ *lang n* (*flatten PLUS X*) = (⋃ *r* ∈ *X*. *lang n r*)
using *lang-PLUS*[*of n sorted-list-of-set X*] **by** *fastforce*

theorem *ACI-norm-lang*[*simp*]:
lang n ≪*r*≫ = *lang n r*
proof (*induct r arbitrary*: *n*)
case (*Plus r1 r2*)
moreover
from *Plus*[*symmetric*] **have** *lang n* (*Plus r1 r2*) ⊆ *lang n* ≪*Plus r1 r2*≫
using *toplevel-summands-in-lang*[*of - n* ≪*r1*≫] *toplevel-summands-in-lang*[*of - n* ≪*r2*≫]
by *auto*
ultimately show *?case* **by** (*fastforce dest*!: *toplevel-summands-lang*)
qed *auto*

lemma *lang-final*: *final r* = ([] ∈ *lang n r*)
using *concI*[*of* [] - []] **by** (*induct r arbitrary*: *n*) *auto*

lemma *in-lang-INTERSECT*:
wf-word n w ⟹ *w* ∈ *lang n* (*INTERSECT xs*) = (∀ *r* ∈ *set xs*. *w* ∈ *lang n r*)
by (*induct xs rule*: *list-singleton-induct*) (*auto simp*: *wf-word*)

lemma *lang-INTERSECT*:
lang n (*INTERSECT xs*) = (*if xs* = [] *then lists* (Σ *n*) *else* ⋂ *r* ∈ *set xs*. *lang n r*)
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *lang-flatten-INTERSECT*[*simp*]:
assumes *finite X X* ≠ {} ∀ *r*∈*X*. *wf n r*
shows *w* ∈ *lang n* (*flatten INTERSECT X*) = (∀ *r* ∈ *X*. *w* ∈ *lang n r*) (**is** *?L* = *?R*)
proof
assume *?L*
thus *?R* **using** *in-lang-INTERSECT*[*OF bspec*[*OF wf-lang-wf-word*[*OF iffD2*[*OF*

wf-flatten-INTERSECT]]],
OF assms(*1*,*3*) ‹*?L*›, *of sorted-list-of-set X*] *assms*(*1*)
by *auto*
next
assume *?R*
with *assms* **show** *?L* **by** (*intro iffD2*[*OF in-lang-INTERSECT*]) (*auto dest*: *wf-lang-wf-word*)
qed

end

3 Derivatives of Π-Extended Regular Expressions

locale *embed* = *project* Σ *wf-atom project lookup*
for Σ :: *nat* ⇒ *'a set*
and *wf-atom* :: *nat* ⇒ *'b* :: *linorder* ⇒ *bool*
and *project* :: *'a* ⇒ *'a*
and *lookup* :: *'b* ⇒ *'a* ⇒ *bool* +
fixes *embed* :: *'a* ⇒ *'a list*
assumes *embed*: ⋀*a*. *a* ∈ Σ *n* ⟹ *b* ∈ *set* (*embed a*) = (*b* ∈ Σ (*Suc n*) ∧ *project b* = *a*)
begin

3.1 Syntactic Derivatives

primrec *lderiv* :: *'a* ⇒ *'b rexp* ⇒ *'b rexp* **where**
lderiv - Zero = *Zero*
| *lderiv - Full* = *Full*
| *lderiv - One* = *Zero*
| *lderiv a* (*Atom b*) = (*if lookup b a then One else Zero*)
| *lderiv a* (*Plus r s*) = *Plus* (*lderiv a r*) (*lderiv a s*)
| *lderiv a* (*Times r s*) =
(*let r's* = *Times* (*lderiv a r*) *s*
in if final r then Plus r's (*lderiv a s*) *else r's*)
| *lderiv a* (*Star r*) = *Times* (*lderiv a r*) (*Star r*)
| *lderiv a* (*Not r*) = *Not* (*lderiv a r*)
| *lderiv a* (*Inter r s*) = *Inter* (*lderiv a r*) (*lderiv a s*)
| *lderiv a* (*Pr r*) = *Pr* (*PLUS* (*map* (λ*a'*. *lderiv a' r*) (*embed a*)))

primrec *lderivs* **where**
lderivs [] *r* = *r*
| *lderivs* (*w*#*ws*) *r* = *lderivs ws* (*lderiv w r*)

3.2 Finiteness of ACI-Equivalent Derivatives

lemma *toplevel-summands-lderiv*:

toplevel-summands (*lderiv as r*) = (⋃ *s*∈*toplevel-summands r*. *toplevel-summands* (*lderiv as s*))
by (*induct r*) (*auto simp*: *Let-def*)

lemma *lderivs-Zero*[*simp*]: *lderivs xs Zero* = *Zero*
by (*induct xs*) *auto*

lemma *lderivs-Full*[*simp*]: *lderivs xs Full* = *Full*
by (*induct xs*) *auto*

lemma *lderivs-One*: *lderivs xs One* ∈ {*Zero*, *One*}
by (*induct xs*) *auto*

lemma *lderivs-Atom*: *lderivs xs* (*Atom as*) ∈ {*Zero*, *One*, *Atom as*}
proof (*induct xs*)
case (*Cons x xs*) **thus** *?case* **by** (*auto intro*: *insertE*[*OF lderivs-One*])
qed *simp*

lemma *lderivs-Plus*: *lderivs xs* (*Plus r s*) = *Plus* (*lderivs xs r*) (*lderivs xs s*)
by (*induct xs arbitrary*: *r s*) *auto*

lemma *lderivs-PLUS*: *lderivs xs* (*PLUS ys*) = *PLUS* (*map* (*lderivs xs*) *ys*)
by (*induct ys rule*: *list-singleton-induct*) (*auto simp*: *lderivs-Plus*)

lemma *toplevel-summands-lderivs-Times*: *toplevel-summands* (*lderivs xs* (*Times r s*)) ⊆
{*Times* (*lderivs xs r*) *s*} ∪
{*r'*. ∃ *ys zs*. *r'* ∈ *toplevel-summands* (*lderivs ys s*) ∧ *ys* ≠ [] ∧ *zs* @ *ys* = *xs*}
proof (*induct xs arbitrary*: *r s*)
case (*Cons x xs*)
thus *?case* **by** (*auto simp*: *Let-def lderivs-Plus*) (*fastforce intro*: *exI*[*of* - *x*#*xs*])+
qed *simp*

lemma *toplevel-summands-lderivs-Star-nonempty*:
xs ≠ [] ⟹ *toplevel-summands* (*lderivs xs* (*Star r*)) ⊆
{*Times* (*lderivs ys r*) (*Star r*) | *ys*. ∃ *zs*. *ys* ≠ [] ∧ *zs* @ *ys* = *xs*}
proof (*induct xs rule*: *length-induct*)
case (*1 xs*)
then obtain *y ys* **where** *xs* = *y* # *ys* **by** (*cases xs*) *auto*
thus *?case* **using** *spec*[*OF 1*(*1*)]
by (*auto dest*!: *subsetD*[*OF toplevel-summands-lderivs-Times*] *intro*: *exI*[*of* - *y*#*ys*])
(*auto elim*!: *impE dest*!: *meta-spec subsetD*)
qed

lemma *toplevel-summands-lderivs-Star*:
toplevel-summands (*lderivs xs* (*Star r*)) ⊆
{*Star r*} ∪ {*Times* (*lderivs ys r*) (*Star r*) | *ys*. ∃ *zs*. *ys* ≠ [] ∧ *zs* @ *ys* = *xs*}
by (*cases xs* = []) (*auto dest*!: *toplevel-summands-lderivs-Star-nonempty*)

lemma *ex-lderivs-Pr*: $\exists s.$ *lderivs ass* (*Pr r*) = *Pr s*
by (*induct ass arbitrary*: *r*) *auto*

lemma *toplevel-summands-PLUS*:
xs ≠ [] ⟹ *toplevel-summands* (*PLUS* (*map f xs*)) = (⋃ *r* ∈ *set xs*. *toplevel-summands* (*f r*))
by (*induct xs rule*: *list-singleton-induct*) *simp-all*

lemma *lderiv-toplevel-summands-Zero*:
⟦*lderivs xs* (*Pr r*) = *Pr s*; *toplevel-summands r* = {*Zero*}⟧ ⟹ *toplevel-summands s* = {*Zero*}
proof (*induct xs arbitrary*: *r s*)
case (*Cons y ys*)
from *Cons.prems*(*1*) **have** *toplevel-summands* (*PLUS* (*map* (λ*a*. *lderiv a r*) (*embed y*))) = {*Zero*}
proof (*cases embed y* = [])
case *False*
show *?thesis* **using** *Cons.prems*(*2*) **unfolding** *toplevel-summands-PLUS*[*OF False*]
by (*subst toplevel-summands-lderiv*) (*simp add*: *False*)
qed *simp*
with *Cons* **show** *?case* **by** *simp*
qed *simp*

lemma *toplevel-summands-lderivs-Pr*:
⟦*xs* ≠ []; *lderivs xs* (*Pr r*) = *Pr s*⟧ ⟹
toplevel-summands s ⊆ {*Zero*} ∨ *toplevel-summands s* ⊆ (⋃ *xs*. *toplevel-summands* (*lderivs xs r*))
proof (*induct xs arbitrary*: *r s*)
case (*Cons y ys*) **note** ∗ = *this*
show *?case*
proof (*cases embed y* = [])
case *True* **with** *Cons* **show** *?thesis* **by** (*cases ys* = []) (*auto dest*: *lderiv-toplevel-summands-Zero*)
next
case *False*
show *?thesis*
proof (*cases ys*)
case *Nil* **with** ∗ **show** *?thesis*
by (*auto simp*: *toplevel-summands-PLUS*[*OF False*]) (*metis lderivs.simps*)
next
case (*Cons z zs*)
have *toplevel-summands s* ⊆ {*Zero*} ∨ *toplevel-summands s* ⊆
(⋃ *xs*. *toplevel-summands* (*lderivs xs* (*PLUS* (*map* (λ*a*. *lderiv a r*) (*embed y*))))) (**is** - ∨ *?B*)
by (*rule* ∗(*1*)) (*auto simp*: *Cons* ∗(*3*)[*symmetric*])
thus *?thesis*
proof
assume *?B*

```
      also have ... ⊆ (⋃xs. toplevel-summands (lderivs xs r))
        by (auto simp: lderivs-PLUS toplevel-summands-PLUS[OF False]) (metis
lderivs.simps(2))
      finally show ?thesis ..
    qed blast
  qed
 qed
qed simp

lemma lderivs-Pr:
 {lderivs xs (Pr r) | xs. True} ⊆
  {Pr s | s. toplevel-summands s ⊆ {Zero} ∨
            toplevel-summands s ⊆ (⋃xs. toplevel-summands (lderivs xs r))}
 (is ?L ⊆?R)
proof (rule subsetI)
 fix s assume s ∈ ?L
 then obtain xs where s = lderivs xs (Pr r) by blast
 moreover obtain t where lderivs xs (Pr r) = Pr t using ex-lderivs-Pr by blast
 ultimately show s ∈ ?R
  by (cases xs = []) (auto dest!: toplevel-summands-lderivs-Pr elim!: allE[of - []])
qed

lemma ACI-norm-toplevel-summands-Zero: toplevel-summands r ⊆ {Zero} ⟹
≪r≫ = Zero
 by (subst ACI-norm-flatten) (auto dest: subset-singletonD)

lemma ACI-norm-lderivs-Pr:
 ACI-norm ‘ {lderivs xs (Pr r) | xs. True} ⊆
   {Pr Zero} ∪ {Pr ≪s≫ | s. toplevel-summands s ⊆ (⋃xs. toplevel-summands
≪lderivs xs r≫)}
proof (intro subset-trans[OF image-mono[OF lderivs-Pr]] subsetI,
      elim imageE CollectE exE conjE disjE)
 fix x x′ s :: 'b rexp
 assume ∗: x = ≪x′≫ x′ = Pr s and toplevel-summands s ⊆ {Zero}
 hence ≪Pr s≫ = Pr Zero using ACI-norm-toplevel-summands-Zero by simp
 thus x ∈ {Pr Zero} ∪
  {Pr ≪s≫ |s. toplevel-summands s ⊆ (⋃xs. toplevel-summands ≪lderivs xs r≫)}
   unfolding ∗ by blast
next
 fix x x′ s :: 'b rexp
 assume ∗: x = ≪x′≫ x′ = Pr s and toplevel-summands s ⊆ (⋃xs. toplevel-summands
(lderivs xs r))
 hence toplevel-summands ≪s≫ ⊆ (⋃xs. toplevel-summands ≪lderivs xs r≫)
   by (fastforce simp: toplevel-summands-ACI-norm)
 moreover have x = Pr ≪≪s≫≫ unfolding ∗ ACI-norm-idem ACI-norm.simps(10)
..
 ultimately show x ∈ {Pr Zero} ∪
  {Pr ≪s≫ |s. toplevel-summands s ⊆ (⋃xs. toplevel-summands ≪lderivs xs r≫)}
   by blast
```

qed

lemma *finite-ACI-norm-toplevel-summands*: *finite B* $\Longrightarrow$ *finite* $\{f\ \ll s\gg |s.\ toplevel\text{-}summands\ s \subseteq B\}$
apply (*elim finite-surj* [*OF iffD2* [*OF finite-Pow-iff*], *of - - f o flatten PLUS o image ACI-norm*])
apply (*subst ACI-norm-flatten*)
apply *auto*
done

lemma *lderivs-Not*: *lderivs xs* (*Not r*) = *Not* (*lderivs xs r*)
by (*induct xs arbitrary*: *r*) *auto*

lemma *lderivs-Inter*: *lderivs xs* (*Inter r s*) = *Inter* (*lderivs xs r*) (*lderivs xs s*)
by (*induct xs arbitrary*: *r s*) *auto*

theorem *finite-lderivs*: *finite* $\{\ll lderivs\ xs\ r\gg \mid xs\ .\ True\}$
proof (*induct r*)
case *Zero* **show** *?case* **by** *simp*
next
case *One* **show** *?case*
by (*rule finite-surj*[*of* {*Zero*, *One*}]) (*blast intro*: *insertE*[*OF lderivs-One*])+
next
case (*Atom as*) **show** *?case*
by (*rule finite-surj*[*of* {*Zero*, *One*, *Atom as*}]) (*blast intro*: *insertE*[*OF lderivs-Atom*])+
next
case (*Plus r s*)
show *?case* **by** (*auto simp*: *lderivs-Plus intro*!: *finite-surj*[*OF finite-cartesian-product*[*OF Plus*]])
next
case (*Times r s*)
hence *finite* ($\bigcup$ (*toplevel-summands* ‘ $\{\ll lderivs\ xs\ s\gg \mid xs\ .\ True\}$)) **by** *auto*
moreover have $\{\ll r'\gg |r'.\ \exists ys.\ r' \in toplevel\text{-}summands\ (lderivs\ ys\ s)\} =$
$\{r'.\ \exists ys.\ r' \in toplevel\text{-}summands\ \ll lderivs\ ys\ s\gg\}$
unfolding *toplevel-summands-ACI-norm* **by** *auto*
ultimately have *fin*: *finite* $\{\ll r'\gg |r'.\ \exists ys.\ r' \in toplevel\text{-}summands\ (lderivs\ ys\ s)\}$
by (*fastforce intro*: *finite-subset*[*of* - $\bigcup$ (*toplevel-summands* ‘ $\{\ll lderivs\ xs\ s\gg \mid xs\ .\ True\}$)])
let *?X* = $\lambda xs.\ \{Times\ (lderivs\ ys\ r)\ s \mid ys.\ True\} \cup \{r'.\ r' \in (\bigcup ys.\ toplevel\text{-}summands\ (lderivs\ ys\ s))\}$
show *?case*
proof (*simp only*: *ACI-norm-flatten*,
rule finite-surj[*of* $\{X.\ \exists xs.\ X \subseteq ACI\text{-}norm$ ‘ *?X xs*} - *flatten PLUS*])
show *finite* $\{X.\ \exists xs.\ X \subseteq ACI\text{-}norm$ ‘ *?X xs*$\}$
using *fin* **by** (*fastforce simp*: *image-Un elim*: *finite-subset*[*rotated*] *intro*: *finite-surj*[*OF Times*(*1*), *of* - $\lambda r.\ Times\ r\ \ll s\gg$])

qed (*fastforce dest*!: *subsetD*[*OF toplevel-summands-lderivs-Times*] *intro*!: *imageI*)
next
case (*Star r*)
let *?f* = λ*f r'*. *Times r'* (*Star* (*f r*))
let *?X* = {*Star r*} ∪ *?f id* ‘ {*r'*. *r'* ∈ {*lderivs ys r*|*ys*. *True*}}
show *?case*
proof (*simp only*: *ACI-norm-flatten*,
rule finite-surj[*of* {*X*. *X* ⊆ *ACI-norm* ‘ *?X*} - *flatten PLUS*])
have ∗: ⋀*X*. *ACI-norm* ‘ *?f* (λ*x*. *x*) ‘ *X* = *?f ACI-norm* ‘ *ACI-norm* ‘ *X* **by** (*auto simp*: *image-def*)
show *finite* {*X*. *X* ⊆ *ACI-norm* ‘ *?X*}
by (*rule finite-Collect-subsets*)
(*auto simp*: ∗ *intro*!: *finite-imageI*[*of* - *?f ACI-norm*] *intro*: *finite-subset*[*OF* - *Star*])
qed (*fastforce dest*!: *subsetD*[*OF toplevel-summands-lderivs-Star*] *intro*!: *imageI*)
next
case (*Not r*) **thus** *?case* **by** (*auto simp*: *lderivs-Not*) (*blast intro*: *finite-surj*)
next
case (*Inter r s*)
show *?case* **by** (*auto simp*: *lderivs-Inter intro*!: *finite-surj*[*OF finite-cartesian-product*[*OF Inter*]])
next
case (*Pr r*)
hence ∗: *finite* (⋃ (*toplevel-summands* ‘ {≪*lderivs xs r*≫ | *xs* . *True*})) **by** *auto*
have *finite* (⋃*xs*. *toplevel-summands* ≪*lderivs xs r*≫) **by** (*rule finite-subset*[*OF* - ∗]) *auto*
hence *fin*: *finite* {*Pr* ≪*s*≫ |*s*. *toplevel-summands s* ⊆ (⋃*xs*. *toplevel-summands* ≪*lderivs xs r*≫)}
by (*intro finite-ACI-norm-toplevel-summands*)
have {*s*. ∃*xs*. *s* = ≪*lderivs xs* (*Pr r*)≫} = {≪*s*≫| *s*. ∃*xs*. *s* = *lderivs xs* (*Pr r*)}
by *auto*
thus *?case* **using** *finite-subset*[*OF ACI-norm-lderivs-Pr*, *of r*] *fin* **unfolding** *image-Collect* **by** *auto*
qed

3.3 Wellformedness and language of derivatives

lemma *wf-lderiv*[*simp*]: *wf n r* ⟹ *wf n* (*lderiv w r*)
by (*induct r arbitrary*: *n w*) (*auto simp add*: *Let-def*)

lemma *wf-lderivs*[*simp*]: *wf n r* ⟹ *wf n* (*lderivs ws r*)
by (*induct ws arbitrary*: *r*) (*auto intro*: *wf-lderiv*)

lemma *lQuot-map-project*:
assumes *as* ∈ Σ *n A* ⊆ *lists* (Σ (*Suc n*))
shows *lQuot as* (*map project* ‘ *A*) = *map project* ‘ (⋃ *a* ∈ *set* (*embed as*). *lQuot a A*) (**is** *?L* = *?R*)
proof (*intro equalityI image-subsetI subsetI*)

fix *xss* **assume** *xss* ∈ *?L*
with *assms* **obtain** *zss*
where *zss*: *zss* ∈ *A as* # *xss* = *map project zss*
unfolding *lQuot-def* **by** *fastforce*
hence *xss* = *map project* (*tl zss*) **by** *auto*
with *zss assms*(*2*) **show** *xss* ∈ *?R* **using** *embed*[*OF project*, *of* - *n*] **unfolding** *lQuot-def* **by** *fastforce*
next
fix *xss* **assume** *xss* ∈ (⋃ *a* ∈ *set* (*embed as*). *lQuot a A*)
with *assms*(*1*) **show** *map project xss* ∈ *lQuot as* (*map project* ‘ *A*) **unfolding** *lQuot-def*
by (*fastforce intro*!: *rev-image-eqI simp*: *embed*)
qed

lemma *lang-lderiv*: ⟦*wf n r*; *w* ∈ Σ *n*⟧ ⟹ *lang n* (*lderiv w r*) = *lQuot w* (*lang n r*)
proof (*induct r arbitrary*: *n w*)
case (*Pr r*)
hence ∗: *wf* (*Suc n*) *r* ⋀*w*′. *w*′ ∈ *set* (*embed w*) ⟹ *w*′ ∈ Σ (*Suc n*) **by** (*auto simp*: *embed*)
from *Pr*(*1*)[*OF* ∗] *lQuot-map-project*[*OF Pr*(*3*) *lang-subset-lists*[*OF* ∗(*1*)]] **show** *?case*
by (*auto simp*: *wf-lderiv*[*OF* ∗(*1*)])
qed (*auto simp*: *Let-def lang-final*[*symmetric*])

lemma *lang-lderivs*: ⟦*wf n r*; *wf-word n ws*⟧ ⟹ *lang n* (*lderivs ws r*) = *lQuots ws* (*lang n r*)
by (*induct ws arbitrary*: *r*) (*auto simp*: *lang-lderiv*)

corollary *lderivs-final*:
assumes *wf n r wf-word n ws*
shows *final* (*lderivs ws r*) ⟷ *ws* ∈ *lang n r*
using *lang-lderivs*[*OF assms*] *lang-final*[*of lderivs ws r n*] **by** *auto*

abbreviation *lderivs-set n r s* ≡ {(≪*lderivs w r*≫, ≪*lderivs w s*≫) | *w*. *wf-word n w*}

3.4 Deriving preserves ACI-equivalence

lemma *ACI-norm-PLUS*:
list-all2 (λ*r s*. ≪*r*≫ = ≪*s*≫) *xs ys* ⟹ ≪*PLUS xs*≫ = ≪*PLUS ys*≫
proof (*induct rule*: *list-all2-induct*)
case (*Cons x xs y ys*)
hence *length xs* = *length ys* **by** (*elim list-all2-lengthD*)
thus *?case* **using** *Cons* **by** (*induct xs ys rule*: *list-induct2*) *auto*
qed *simp*

lemma *toplevel-summands-ACI-norm-lderiv*:
(⋃ *a*∈*toplevel-summands r*. *toplevel-summands* ≪*lderiv as* ≪*a*≫≫) = *toplevel-summands*

```
«lderiv as «r»» 
proof (induct r)
  case (Plus r1 r2) thus ?case
   unfolding toplevel-summands.simps toplevel-summands-ACI-norm
     toplevel-summands-lderiv[of as «Plus r1 r2»] image-Un Union-Un-distrib
   by (simp add: image-UN)
qed (auto simp: Let-def)

theorem ACI-norm-lderiv:
  «lderiv as «r»» = «lderiv as r»
proof (induct r arbitrary: as)
  case (Plus r1 r2) thus ?case
   unfolding lderiv.simps ACI-norm-flatten[of lderiv as «Plus r1 r2»]
     toplevel-summands-lderiv[of as «Plus r1 r2»] image-Un image-UN
  by (auto simp: toplevel-summands-ACI-norm toplevel-summands-flatten-ACI-norm-image-Union)
   (auto simp: toplevel-summands-ACI-norm[symmetric] toplevel-summands-ACI-norm-lderiv)
next
  case (Pr r)
  hence list-all2 (λr s. «r» = «s»)
    (map (λa. lderiv a «r») (embed as)) (map (λa. lderiv a r) (embed as))
    unfolding list-all2-map1 list-all2-map2 by (intro list-all2-refl)
 thus ?case unfolding lderiv.simps ACI-norm.simps by (blast intro: ACI-norm-PLUS)
qed (simp-all add: Let-def)


corollary lderiv-preserves: «r» = «s» ⟹ «lderiv as r» = «lderiv as s»
  by (rule box-equals[OF - ACI-norm-lderiv ACI-norm-lderiv]) (erule arg-cong)

lemma lderivs-snoc[simp]: lderivs (ws @ [w]) r = (lderiv w (lderivs ws r))
  by (induct ws arbitrary: r) auto

theorem ACI-norm-lderivs:
  «lderivs ws «r»» = «lderivs ws r»
proof (induct ws arbitrary: r rule: rev-induct)
  case (snoc w ws) thus ?case
    using ACI-norm-lderiv[of w lderivs ws r] ACI-norm-lderiv[of w lderivs ws «r»]
by auto
qed simp

lemma lderivs-alt: «lderivs w r» = fold (λa r. «lderiv a r») w «r»
  by (induct w arbitrary: r) (auto simp: ACI-norm-lderiv)

lemma finite-fold-lderiv: finite {fold (λa r. «lderiv a r») w «s» |w. True}
  using finite-lderivs unfolding lderivs-alt .

end
```

4 Some Useful Regular Operators

```
primrec REV :: 'a rexp ⇒ 'a rexp where
  REV Zero = Zero
| REV Full = Full
| REV One = One
| REV (Atom a) = Atom a
| REV (Plus r s) = Plus (REV r) (REV s)
| REV (Times r s) = Times (REV s) (REV r)
| REV (Star r) = Star (REV r)
| REV (Not r) = Not (REV r)
| REV (Inter r s) = Inter (REV r) (REV s)
| REV (Pr r) = Pr (REV r)

lemma REV-REV[simp]: REV (REV r) = r
  by (induct r) auto

lemma final-REV[simp]: final (REV r) = final r
  by (induct r) auto

lemma REV-PLUS: REV (PLUS xs) = PLUS (map REV xs)
  by (induct xs rule: list-singleton-induct) auto

lemma (in alphabet) wf-REV[simp]: wf n r ⟹ wf n (REV r)
  by (induct r arbitrary: n) auto

lemma (in project) lang-REV[simp]: lang n (REV r) = rev ‘ lang n r
  by (induct r arbitrary: n) (auto simp: image-image rev-map image-set-diff)

context embed
begin

primrec rderiv :: 'a ⇒ 'b rexp ⇒ 'b rexp where
  rderiv - Zero = Zero
| rderiv - Full = Full
| rderiv - One = Zero
| rderiv a (Atom b) = (if lookup b a then One else Zero)
| rderiv a (Plus r s) = Plus (rderiv a r) (rderiv a s)
| rderiv a (Times r s) =
    (let rs' = Times r (rderiv a s)
     in if final s then Plus rs' (rderiv a r) else rs')
| rderiv a (Star r) = Times (Star r) (rderiv a r)
| rderiv a (Not r) = Not (rderiv a r)
| rderiv a (Inter r s) = Inter (rderiv a r) (rderiv a s)
| rderiv a (Pr r) = Pr (PLUS (map (λa'. rderiv a' r) (embed a)))

primrec rderivs where
  rderivs [] r = r
```

| *rderivs* (*w*#*ws*) *r* = *rderivs ws* (*rderiv w r*)

lemma *rderivs-snoc*: *rderivs* (*ws* @ [*w*]) *r* = *rderiv w* (*rderivs ws r*)
by (*induct ws arbitrary*: *r*) *auto*

lemma *rderivs-append*: *rderivs* (*ws* @ *ws'*) *r* = *rderivs ws'* (*rderivs ws r*)
by (*induct ws arbitrary*: *r*) *auto*

lemma *rderiv-lderiv*: *rderiv as r* = *REV* (*lderiv as* (*REV r*))
by (*induct r arbitrary*: *as*) (*auto simp*: *Let-def o-def REV-PLUS*)

lemma *rderivs-lderivs*: *rderivs w r* = *REV* (*lderivs w* (*REV r*))
by (*induct w arbitrary*: *r*) (*auto simp*: *rderiv-lderiv*)

lemma *wf-rderiv*[*simp*]: *wf n r* ⟹ *wf n* (*rderiv w r*)
unfolding *rderiv-lderiv* **by** (*rule wf-REV*[*OF wf-lderiv*[*OF wf-REV*]])

lemma *wf-rderivs*[*simp*]: *wf n r* ⟹ *wf n* (*rderivs ws r*)
unfolding *rderivs-lderivs* **by** (*rule wf-REV*[*OF wf-lderivs*[*OF wf-REV*]])

lemma *lang-rderiv*: ⟦*wf n r*; *as* ∈ Σ *n*⟧ ⟹ *lang n* (*rderiv as r*) = *rQuot as* (*lang n r*)
unfolding *rderiv-lderiv rQuot-rev-lQuot* **by** (*simp add*: *lang-lderiv*)

lemma *lang-rderivs*: ⟦*wf n r*; *wf-word n w*⟧ ⟹ *lang n* (*rderivs w r*) = *rQuots w* (*lang n r*)
unfolding *rderivs-lderivs rQuots-rev-lQuots* **by** (*simp add*: *lang-lderivs*)

corollary *rderivs-final*:
assumes *wf n r wf-word n w*
shows *final* (*rderivs w r*) ⟷ *rev w* ∈ *lang n r*
using *lang-rderivs*[*OF assms*] *lang-final*[*of rderivs w r n*] **by** *auto*

lemma *toplevel-summands-REV*[*simp*]: *toplevel-summands* (*REV r*) = *REV* ‘ *toplevel-summands r*
by (*induct r*) *auto*

lemma *ACI-norm-REV*: ≪*REV* ≪*r*≫≫ = ≪*REV r*≫
proof (*induct r*)
case (*Plus r s*)
show *?case*
using [[*unfold-abs-def* = *false*]]
unfolding *REV.simps ACI-norm.simps Plus*[*symmetric*] *image-Un*[*symmetric*]
toplevel-summands.simps(*1*) *toplevel-summands-ACI-norm toplevel-summands-REV*
unfolding *toplevel-summands.simps*(*1*)[*symmetric*] *ACI-norm-flatten toplevel-summands-REV*
unfolding *ACI-norm-flatten*[*symmetric*] *toplevel-summands-ACI-norm*
..
qed *auto*

lemma *ACI-norm-rderiv*: $\ll rderiv\ as\ \ll r\gg\gg = \ll rderiv\ as\ r\gg$
unfolding *rderiv-lderiv* **by** (*metis ACI-norm-REV ACI-norm-lderiv*)

lemma *ACI-norm-rderivs*: $\ll rderivs\ w\ \ll r\gg\gg = \ll rderivs\ w\ r\gg$
unfolding *rderivs-lderivs* **by** (*metis ACI-norm-REV ACI-norm-lderivs*)

theorem *finite-rderivs*: *finite* $\{\ll rderivs\ xs\ r\gg \mid xs\ .\ True\}$
unfolding *rderivs-lderivs*
by (*subst ACI-norm-REV*[*symmetric*]) (*auto intro*: *finite-surj*[*OF finite-lderivs*, *of* - $\lambda r.\ \ll REV\ r\gg$])

lemma *lderiv-PLUS*[*simp*]: *lderiv a* (*PLUS xs*) = *PLUS* (*map* (*lderiv a*) *xs*)
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *rderiv-PLUS*[*simp*]: *rderiv a* (*PLUS xs*) = *PLUS* (*map* (*rderiv a*) *xs*)
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *lang-rderiv-lderiv*: *lang n* (*rderiv a* (*lderiv b r*)) = *lang n* (*lderiv b* (*rderiv a r*))
by (*induct r arbitrary*: *n a b*) (*auto simp*: *Let-def conc-assoc*)

lemma *lang-lderiv-rderiv*: *lang n* (*lderiv a* (*rderiv b r*)) = *lang n* (*rderiv b* (*lderiv a r*))
by (*induct r arbitrary*: *n a b*) (*auto simp*: *Let-def conc-assoc*)

lemma *lang-rderiv-lderivs*[*simp*]: $[\![wf\ n\ r;\ wf\text{-}word\ n\ w;\ a \in \Sigma\ n]\!] \Longrightarrow$
lang n (*rderiv a* (*lderivs w r*)) = *lang n* (*lderivs w* (*rderiv a r*))
by (*induct w arbitrary*: *n r*)
(*auto*, *auto simp*: *lang-lderivs lang-lderiv lang-rderiv lQuot-rQuot*)

lemma *lang-lderiv-rderivs*[*simp*]: $[\![wf\ n\ r;\ wf\text{-}word\ n\ w;\ a \in \Sigma\ n]\!] \Longrightarrow$
lang n (*lderiv a* (*rderivs w r*)) = *lang n* (*rderivs w* (*lderiv a r*))
by (*induct w arbitrary*: *n r*)
(*auto*, *auto simp*: *lang-rderivs lang-lderiv lang-rderiv lQuot-rQuot*)

definition *biderivs w1 w2* = *rderivs w2 o lderivs w1*

lemma *lang-biderivs*: $[\![wf\ n\ r;\ wf\text{-}word\ n\ w1;\ wf\text{-}word\ n\ w2]\!] \Longrightarrow$
lang n (*biderivs w1 w2 r*) = *biQuots w1 w2* (*lang n r*)
unfolding *biderivs-def* **by** (*auto simp*: *lang-rderivs lang-lderivs in-lists-conv-set*)

lemma *wf-biderivs*[*simp*]: *wf n r* $\Longrightarrow$ *wf n* (*biderivs w1 w2 r*)
unfolding *biderivs-def* **by** *simp*

corollary *biderivs-final*:
assumes *wf n r wf-word n w1 wf-word n w2*
shows *final* (*biderivs w1 w2 r*) $\longleftrightarrow$ *w1* @ *rev w2* $\in$ *lang n r*
using *lang-biderivs*[*OF assms*] *lang-final*[*of biderivs w1 w2 r n*] **by** *auto*

lemma *ACI-norm-biderivs*: *≪biderivs w1 w2 ≪r≫≫ = ≪biderivs w1 w2 r≫*
unfolding *biderivs-def* **by** (*metis ACI-norm-lderivs ACI-norm-rderivs o-apply*)

lemma *finite* {*≪biderivs w1 w2 r≫* | *w1 w2* . *True*}
proof −
have {*≪biderivs w1 w2 r≫* | *w1 w2* . *True*} =
(⋃ *s* ∈ {*≪lderivs as r≫* | *as* . *True*}. {*≪rderivs bs s≫* | *bs* . *True*})
unfolding *biderivs-def* **by** (*fastforce simp*: *ACI-norm-rderivs*)
also have *finite* . . . **by** (*rule iffD2*[*OF finite-UN*[*OF finite-lderivs*] *ballI*[*OF finite-rderivs*]])
finally show *?thesis* **.**
qed

end

4.1 Quotioning by the same letter

definition *fin-cut-same x xs* = *take* (*LEAST n*. *drop n xs* = *replicate* (*length xs* − *n*) *x*) *xs*

lemma *fin-cut-same-Nil*[*simp*]: *fin-cut-same x* [] = []
unfolding *fin-cut-same-def* **by** *simp*

lemma *Least-fin-cut-same*: (*LEAST n*. *drop n xs* = *replicate* (*length xs* − *n*) *y*) =
length xs − *length* (*takeWhile* (*λx*. *x* = *y*) (*rev xs*))
(**is** *Least ?P* = *?min*)
proof (*rule Least-equality*)
show *?P ?min* **by** (*induct xs rule*: *rev-induct*) (*auto simp*: *Suc-diff-le replicate-append-same*)
next
fix *m* **assume** *?P m*
have *length xs* − *m* ≤ *length* (*takeWhile* (*λx*. *x* = *y*) (*rev xs*))
proof (*intro length-takeWhile-less-P-nth*)
fix *i* **assume** *i* < *length xs* − *m*
hence *rev xs* ! *i* ∈ *set* (*drop m xs*)
by (*induct xs arbitrary*: *i rule*: *rev-induct*) (*auto simp*: *nth-Cons′*)
with ‹*?P m*› **show** *rev xs* ! *i* = *y* **by** *simp*
qed *simp*
thus *?min* ≤ *m* **by** *linarith*
qed

lemma *takeWhile-takes-all*: *length xs* = *m* ⟹ *m* ≤ *length* (*takeWhile P xs*) ⟷ *Ball* (*set xs*) *P*
by *hypsubst-thin* (*induct xs*, *auto*)

lemma *fin-cut-same-Cons*[*simp*]: *fin-cut-same x* (*y* # *xs*) =
(*if fin-cut-same x xs* = [] *then if x* = *y then* [] *else* [*y*] *else y* # *fin-cut-same x xs*)
unfolding *fin-cut-same-def Least-fin-cut-same*

apply *auto*
apply (*simp add*: *takeWhile-takes-all*)
apply (*simp add*: *takeWhile-takes-all*)
apply *auto*
apply (*metis* (*full-types*) *Suc-diff-le length-rev length-takeWhile-le take-Suc-Cons*)
apply (*simp add*: *takeWhile-takes-all*)
apply (*subst takeWhile-append2*)
apply *auto*
apply (*simp add*: *takeWhile-takes-all*)
apply *auto*
apply (*metis* (*full-types*) *Suc-diff-le length-rev length-takeWhile-le take-Suc-Cons*)
done

lemma *fin-cut-same-singleton*[*simp*]: *fin-cut-same x* (*xs* @ [*x*]) = *fin-cut-same x xs*
by (*induct xs*) *auto*

lemma *fin-cut-same-replicate*[*simp*]: *fin-cut-same x* (*xs* @ *replicate n x*) = *fin-cut-same x xs*
by (*induct n arbitrary*: *xs*)
(*auto simp*: *replicate-append-same*[*symmetric*] *append-assoc*[*symmetric*] *simp del*: *append-assoc*)

lemma *fin-cut-sameE*: *fin-cut-same x xs* = *ys* ⟹ ∃ *m*. *xs* = *ys* @ *replicate m x*
by (*induct xs arbitrary*: *ys*) (*auto*, *metis replicate-Suc*)

definition *SAMEQUOT a A* = {*fin-cut-same a x* @ *replicate m a*| *x m*. *x* ∈ *A*}

lemma *SAMEQUOT-mono*: *A* ⊆ *B* ⟹ *SAMEQUOT a A* ⊆ *SAMEQUOT a B*
unfolding *SAMEQUOT-def* **by** *auto*

locale *embed2* = *embed* Σ *wf-atom project lookup embed*
for Σ :: *nat* ⇒ *'a set*
and *wf-atom* :: *nat* ⇒ *'b* :: *linorder* ⇒ *bool*
and *project* :: *'a* ⇒ *'a*
and *lookup* :: *'b* ⇒ *'a* ⇒ *bool*
and *embed* :: *'a* ⇒ *'a list* +
fixes *singleton* :: *'a* ⇒ *'b*
assumes *wf-singleton*[*simp*]: *a* ∈ Σ *n* ⟹ *wf-atom n* (*singleton a*)
assumes *lookup-singleton*[*simp*]: *lookup* (*singleton a*) *a'* = (*a* = *a'*)
begin

lemma *finite-rderivs-same*: *finite* {≪*rderivs* (*replicate m a*) *r*≫ | *m* . *True*}
by (*auto intro*: *finite-subset*[*OF - finite-rderivs*])

lemma *wf-word-replicate*[*simp*]: *a* ∈ Σ *n* ⟹ *wf-word n* (*replicate m a*)
by (*induct m*) *auto*

lemma *star-singleton*[*simp*]: *star* {[*x*]} = {*replicate m x* | *m* . *True*}

proof (*intro equalityI subsetI*)
fix *xs* **assume** *xs* ∈ *star* {[*x*]}
thus *xs* ∈ {*replicate m x* | *m* . *True*} **by** (*induct xs*) (*auto*, *metis replicate-Suc*)
qed (*auto intro*: *Ball-starI*)

definition *samequot a r* = *Times* (*flatten PLUS* {≪*rderivs* (*replicate m a*) *r*≫ | *m* . *True*}) (*Star* (*Atom* (*singleton a*)))

lemma *wf-samequot*: ⟦*wf n r*; *a* ∈ Σ *n*⟧ ⟹ *wf n* (*samequot a r*)
unfolding *samequot-def wf.simps wf-flatten-PLUS*[*OF finite-rderivs-same*] **by** *auto*

lemma *lang-samequot*: ⟦*wf n r*; *a* ∈ Σ *n*⟧ ⟹
lang n (*samequot a r*) = *SAMEQUOT a* (*lang n r*)
unfolding *SAMEQUOT-def samequot-def lang.simps lang-flatten-PLUS*[*OF finite-rderivs-same*]
apply (*rule sym*)
apply (*auto simp*: *lang-rderivs*)
apply (*intro concI*)
apply *auto*
apply (*insert fin-cut-sameE*[*OF refl*, *of* - *a*])
apply (*drule meta-spec*)
apply (*erule exE*)
apply (*intro exI conjI*)
apply (*rule refl*)
apply (*auto simp*: *lang-rderivs*)
apply (*erule subst*)
apply *assumption*
apply (*erule concE*)
apply (*auto simp*: *lang-rderivs*)
apply (*drule meta-spec*)
apply (*erule exE*)
apply (*intro exI conjI*)
defer
apply *assumption*
unfolding *fin-cut-same-replicate*
apply (*erule trans*)
unfolding *fin-cut-same-replicate*
apply (*rule refl*)
done

fun *rderiv-and-add* **where**
rderiv-and-add as (- :: *bool*, *rs*) =
(*let*
r = ≪*rderiv as* (*hd rs*)≫
in if r ∈ *set rs then* (*False*, *rs*) *else* (*True*, *r* # *rs*))

definition *invar-rderiv-and-add as r brs* ≡
(*if fst brs then True else* ≪*rderiv as* (*hd* (*snd brs*))≫ ∈ *set* (*snd brs*)) ∧

snd brs $\neq$ [] $\wedge$ *distinct* (*snd brs*) $\wedge$
($\forall$ *i* $<$ *length* (*snd brs*). *snd brs* ! *i* = $\ll$*rderivs* (*replicate* (*length* (*snd brs*) $-$ *1* $-$ *i*) *as*) *r*$\gg$)

lemma *invar-rderiv-and-add-init*: *invar-rderiv-and-add as r* (*True*, [$\ll$*r*$\gg$])
unfolding *invar-rderiv-and-add-def* **by** *auto*

lemma *invar-rderiv-and-add-step*: *invar-rderiv-and-add as r brs* $\Longrightarrow$ *fst brs* $\Longrightarrow$
invar-rderiv-and-add as r (*rderiv-and-add as brs*)
unfolding *invar-rderiv-and-add-def* **by** (*cases brs*) (*auto simp*:
Let-def nth-Cons' ACI-norm-rderiv rderivs-snoc[*symmetric*] *neq-Nil-conv replicate-append-same*)

lemma *rderivs-replicate-mult*: [$\![$$\ll$*rderivs* (*replicate i as*) *r*$\gg$ = $\ll$*r*$\gg$; *i* $>$ *0*$\!]$ $\Longrightarrow$
$\ll$*rderivs* (*replicate* (*m* $*$ *i*) *as*) *r*$\gg$ = $\ll$*r*$\gg$
proof (*induct m arbitrary*: *r*)
case (*Suc m*)
hence $\ll$*rderivs* (*replicate* (*m* $*$ *i*) *as*) $\ll$*rderivs* (*replicate i as*) *r*$\gg\gg$ = $\ll$*r*$\gg$
by (*auto simp*: *ACI-norm-rderivs*)
thus *?case* **by** (*auto simp*: *ACI-norm-rderivs replicate-add rderivs-append*)
qed *simp*

lemma *rderivs-replicate-mult-rest*:
assumes $\ll$*rderivs* (*replicate i as*) *r*$\gg$ = $\ll$*r*$\gg$ *k* $<$ *i*
shows $\ll$*rderivs* (*replicate* (*m* $*$ *i* + *k*) *as*) *r*$\gg$ = $\ll$*rderivs* (*replicate k as*) *r*$\gg$ (**is** *?L* = *?R*)
proof $-$
have *?L* = $\ll$*rderivs* (*replicate k as*) $\ll$*rderivs* (*replicate* (*m* $*$ *i*) *as*) *r*$\gg\gg$
by (*simp add*: *ACI-norm-rderivs replicate-add rderivs-append*)
also have $\ll$*rderivs* (*replicate* (*m* $*$ *i*) *as*) *r*$\gg$ = $\ll$*r*$\gg$ **using** *assms*
by (*simp add*: *rderivs-replicate-mult*)
finally show *?thesis* **by** (*simp add*: *ACI-norm-rderivs*)
qed

lemma *rderivs-replicate-mod*:
assumes $\ll$*rderivs* (*replicate i as*) *r*$\gg$ = $\ll$*r*$\gg$ *i* $>$ *0*
shows $\ll$*rderivs* (*replicate m as*) *r*$\gg$ = $\ll$*rderivs* (*replicate* (*m mod i*) *as*) *r*$\gg$ (**is** *?L* = *?R*)
by (*subst div-mult-mod-eq*[*symmetric*, *of m i*])
(*intro rderivs-replicate-mult-rest*[*OF assms*(*1*)] *mod-less-divisor*[*OF assms*(*2*)])

lemma *rderivs-replicate-diff*: [$\![$$\ll$*rderivs* (*replicate i as*) *r*$\gg$ = $\ll$*rderivs* (*replicate j as*) *r*$\gg$; *i* $>$ *j*$\!]$ $\Longrightarrow$
$\ll$*rderivs* (*replicate* (*i* $-$ *j*) *as*) (*rderivs* (*replicate j as*) *r*)$\gg$ = $\ll$*rderivs* (*replicate j as*) *r*$\gg$
unfolding *rderivs-append*[*symmetric*] *replicate-add*[*symmetric*] **by** *auto*

lemma *samequot-wf*:
assumes *wf n r while-option fst* (*rderiv-and-add as*) (*True*, [$\ll$*r*$\gg$]) = *Some* (*b*, *rs*)

shows *wf n* (*PLUS rs*)
proof −
have ¬ *b* **using** *while-option-stop*[*OF assms*(*2*)] **by** *simp*
from *while-option-rule*[**where** *P*=*invar-rderiv-and-add as r*,
OF invar-rderiv-and-add-step assms(*2*) *invar-rderiv-and-add-init*]
have ∗: *invar-rderiv-and-add as r* (*b*, *rs*) **by** *simp*
thus *wf n* (*PLUS rs*) **unfolding** *invar-rderiv-and-add-def wf-PLUS*
by (*auto simp*: *in-set-conv-nth wf-rderivs*[*OF assms*(*1*)])
qed

lemma *samequot-soundness*:
assumes *while-option fst* (*rderiv-and-add as*) (*True*, [«*r*»]) = *Some* (*b*, *rs*)
shows *lang n* (*PLUS rs*) = ⋃ (*lang n* ‘ {«*rderivs* (*replicate m as*) *r*» | *m*. *True*})
proof −
have ¬ *b* **using** *while-option-stop*[*OF assms*] **by** *simp*
moreover
from *while-option-rule*[**where** *P*=*invar-rderiv-and-add as r*,
OF invar-rderiv-and-add-step assms invar-rderiv-and-add-init]
have ∗: *invar-rderiv-and-add as r* (*b*, *rs*) **by** *simp*
ultimately obtain *i* **where** *i*: *i* < *length rs* **and** «*rderivs* (*replicate* (*length rs* − *Suc i*) *as*) *r*» =
«*rderivs* (*replicate* (*Suc* (*length rs* − *Suc 0*)) *as*) *r*» (**is** «*rderivs ?x r*» = -)
unfolding *invar-rderiv-and-add-def* **by** (*auto simp*: *in-set-conv-nth hd-conv-nth ACI-norm-rderiv*
rderivs-snoc[*symmetric*] *replicate-append-same*)
with ∗ **have** «*rderivs ?x r*» = «*rderivs* (*replicate* (*length rs*) *as*) *r*»
by (*auto simp*: *invar-rderiv-and-add-def*)
with *i* **have** *cyc*: «*rderivs* (*replicate* (*Suc i*) *as*) (*rderivs ?x r*)» = «*rderivs ?x r*»
by (*fastforce dest*: *rderivs-replicate-diff*[*OF sym*])
{ fix *m*
have ∃ *i*<*length rs*. *rs* ! *i* = «*rderivs* (*replicate m as*) *r*»
proof (*cases m* > *length rs* − *Suc i*)
case *True*
with *i* **obtain** *m*′ **where** *m*: *m* = *m*′ + *length rs* − *Suc i*
by *atomize-elim* (*auto intro*: *exI*[*of* - *m* − (*length rs* − *Suc i*)])
with *i* **have** «*rderivs* (*replicate m as*) *r*» = «*rderivs* (*replicate m*′ *as*) (*rderivs ?x r*)»
unfolding *replicate-add*[*symmetric*] *rderivs-append*[*symmetric*] **by** (*simp add*: *add.commute*)
also from *cyc* **have** ... = «*rderivs* (*replicate* (*m*′ *mod* (*Suc i*)) *as*) (*rderivs ?x r*)»
by (*elim rderivs-replicate-mod*) *simp*
also from *i* **have** ... = «*rderivs* (*replicate* (*m*′ *mod* (*Suc i*) + *length rs* − *Suc i*) *as*) *r*»
unfolding *rderivs-append*[*symmetric*] *replicate-add*[*symmetric*] **by** (*simp add*: *add.commute*)
also from *m i* **have** ... = «*rderivs* (*replicate* ((*m* − (*length rs* − *Suc i*))

```
mod (Suc i) + length rs − Suc i) as) r≫
      by simp
    also have ... = ≪rderivs (replicate (length rs − Suc (i − (m − (length rs −
Suc i)) mod (Suc i))) as) r≫
      by (subst Suc-diff-le[symmetric])
      (metis less-Suc-eq-le mod-less-divisor zero-less-Suc, simp add: add.commute)
   finally have ∃ j < length rs. ≪rderivs (replicate m as) r≫ = ≪rderivs (replicate
(length rs − Suc j) as) r≫
      using i by (metis less-imp-diff-less)
     with ∗ show ?thesis unfolding invar-rderiv-and-add-def by auto
  next
    case False
    with i have ∃ j < length rs. m = length rs − Suc j
      by (induct m)
        (metis diff-self-eq-0 gr-implies-not0 lessI nat.exhaust,
      metis (no-types) One-nat-def Suc-diff-Suc diff-Suc-1 gr0-conv-Suc less-imp-diff-less
          not-less-eq not-less-iff-gr-or-eq)
    with ∗ show ?thesis unfolding invar-rderiv-and-add-def by auto
  qed
 }
 hence ⋃ (lang n ‘ {≪rderivs (replicate m as) r≫ |m. True}) ⊆ lang n (PLUS
rs)
   by (fastforce simp: in-set-conv-nth intro!: bexI[rotated])
 moreover from ∗ have lang n (PLUS rs) ⊆ ⋃ (lang n ‘ {≪rderivs (replicate
m as) r≫ |m. True})
   unfolding invar-rderiv-and-add-def by (fastforce simp: in-set-conv-nth)
 ultimately show lang n (PLUS rs) = ⋃ (lang n ‘ {≪rderivs (replicate m as)
r≫ |m. True}) by blast
qed

lemma length-subset-card: ⟦finite X; distinct (x # xs); set (x # xs) ⊆ X⟧ ⟹
length xs < card X
 by (metis card-mono distinct-card impossible-Cons not-le-imp-less order-trans)

lemma samequot-termination:
 assumes while-option fst (rderiv-and-add as) (True, [≪r≫]) = None (is ?cl =
None)
 shows False
proof −
 let ?D = {≪rderivs (replicate m as) r≫ | m . True}
 let ?f = λ(b, rs). card ?D + 1 − length rs + (if b then 1 else 0)
 have ∃ st. ?cl = Some st
   apply (rule measure-while-option-Some[of invar-rderiv-and-add as r - - ?f])
    apply (auto simp: invar-rderiv-and-add-init invar-rderiv-and-add-step)
   apply (auto simp: invar-rderiv-and-add-def Let-def neq-Nil-conv in-set-conv-nth
      intro!: diff-less-mono2 length-subset-card[OF finite-rderivs-same, simplified])
     apply auto []
    apply fastforce
   apply (metis Suc-less-eq nth-Cons-Suc)
```

done
with *assms* **show** *False* **by** *auto*
qed

definition *samequot-exec a r* =
Times (*PLUS* (*snd* (*the* (*while-option fst* (*rderiv-and-add a*) (*True*, [≪*r*≫])))))
(*Star* (*Atom* (*singleton a*)))

lemma *wf-samequot-exec*: ⟦*wf n r*; *as* ∈ Σ *n*⟧ ⟹ *wf n* (*samequot-exec as r*)
unfolding *samequot-exec-def*
by (*cases while-option fst* (*rderiv-and-add as*) (*True*, [≪*r*≫]))
(*auto dest*: *samequot-termination samequot-wf*)

lemma *samequot-exec-samequot*: *lang n* (*samequot-exec as r*) = *lang n* (*samequot as r*)
unfolding *samequot-exec-def samequot-def lang.simps lang-flatten-PLUS*[*OF finite-rderivs-same*]
by (*cases while-option fst* (*rderiv-and-add as*) (*True*, [≪*r*≫]))
(*auto dest*: *samequot-termination dest*!: *samequot-soundness*[*of - - - - n*] *simp del*: *ACI-norm-lang*)

lemma *lang-samequot-exec*:
⟦*wf n r*; *as* ∈ Σ *n*⟧ ⟹ *lang n* (*samequot-exec as r*) = *SAMEQUOT as* (*lang n r*)
unfolding *samequot-exec-samequot* **by** (*rule lang-samequot*)

end

4.2 Suffix and Prefix Languages

definition *Suffix* :: *'a lang* ⇒ *'a lang* **where**
Suffix L = {*w*. ∃ *u*. *u* @ *w* ∈ *L*}

definition *Prefix* :: *'a lang* ⇒ *'a lang* **where**
Prefix L = {*w*. ∃ *u*. *w* @ *u* ∈ *L*}

lemma *Prefix-Suffix*: *Prefix L* = *rev* ‘ *Suffix* (*rev* ‘ *L*)
unfolding *Prefix-def Suffix-def*
by (*auto simp*: *rev-append-invert*
intro: *image-eqI*[*of - rev*, *OF rev-rev-ident*[*symmetric*]]
image-eqI[*of - rev*, *OF rev-append*[*symmetric*]])

definition *Root* :: *'a lang* ⇒ *'a lang* **where**
Root L = {*x* . ∃ *n* > *0*. *x* ^^ *n* ∈ *L*}

definition *Cycle* :: *'a lang* ⇒ *'a lang* **where**
Cycle L = {*u* @ *w* | *u w*. *w* @ *u* ∈ *L*}

context *embed*
begin

context
fixes $n :: nat$
begin

definition $SUFFIX :: 'b\ rexp \Rightarrow 'b\ rexp$ **where**
$SUFFIX\ r = flatten\ PLUS\ \{\ll lderivs\ w\ r\gg |\ w.\ wf\text{-}word\ n\ w\}$

lemma *finite-lderivs-wf*: $finite\ \{\ll lderivs\ w\ r\gg |\ w.\ wf\text{-}word\ n\ w\}$
by (*auto intro*: *finite-subset*[*OF* - *finite-lderivs*])

definition $PREFIX :: 'b\ rexp \Rightarrow 'b\ rexp$ **where**
$PREFIX\ r = REV\ (SUFFIX\ (REV\ r))$

lemma *wf-SUFFIX*[*simp*]: $wf\ n\ r \Longrightarrow wf\ n\ (SUFFIX\ r)$
unfolding *SUFFIX-def* **by** (*intro iffD2*[*OF wf-flatten-PLUS*[*OF finite-lderivs-wf*]])
auto

lemma *lang-SUFFIX*[*simp*]: $wf\ n\ r \Longrightarrow lang\ n\ (SUFFIX\ r) = Suffix\ (lang\ n\ r)$
unfolding *SUFFIX-def Suffix-def*
using *lang-flatten-PLUS*[*OF finite-lderivs-wf*] *lang-lderivs wf-lang-wf-word*
by *fastforce*

lemma *wf-PREFIX*[*simp*]: $wf\ n\ r \Longrightarrow wf\ n\ (PREFIX\ r)$
unfolding *PREFIX-def* **by** *auto*

lemma *lang-PREFIX*[*simp*]: $wf\ n\ r \Longrightarrow lang\ n\ (PREFIX\ r) = Prefix\ (lang\ n\ r)$
unfolding *PREFIX-def* **by** (*auto simp*: *Prefix-Suffix*)

end

lemma *take-drop-CycleI*[*intro*!]: $x \in L \Longrightarrow drop\ i\ x\ @\ take\ i\ x \in Cycle\ L$
unfolding *Cycle-def* **by** *fastforce*

lemma *take-drop-CycleI'*[*intro*!]: $drop\ i\ x\ @\ take\ i\ x \in L \Longrightarrow x \in Cycle\ L$
by (*drule take-drop-CycleI*[*of* - - *length x* − *i*]) *auto*

end

5 Π-Extended Dual Regular Expressions

5.1 Syntax of regular expressions

datatype $'a\ rexp\text{-}dual =$
$CoZero\ (co\text{:}\ bool)\ |$
$CoOne\ (co\text{:}\ bool)\ |$
$CoAtom\ (co\text{:}\ bool)\ 'a\ |$
$CoPlus\ (co\text{:}\ bool)\ 'a\ rexp\text{-}dual\ 'a\ rexp\text{-}dual\ |$

CoTimes (*co*: *bool*) *'a rexp-dual 'a rexp-dual* |
CoStar (*co*: *bool*) *'a rexp-dual* |
CoPr (*co*: *bool*) *'a rexp-dual*
derive *linorder rexp-dual*

abbreviation *CoPLUS-dual b* ≡ *rexp-of-list* (*CoPlus b*) (*CoZero b*)
abbreviation *bool-unop-dual b* ≡ (*if b then id else HOL.Not*)
abbreviation *bool-binop-dual b* ≡ (*if b then* (∨) *else* (∧))
abbreviation *set-binop-dual b* ≡ (*if b then* (∪) *else* (∩))

primrec *final-dual* :: *'a rexp-dual* ⇒ *bool*
where
final-dual (*CoZero b*) = (¬ *b*)
| *final-dual* (*CoOne b*) = *b*
| *final-dual* (*CoAtom b* -) = (¬ *b*)
| *final-dual* (*CoPlus b r s*) = *bool-binop-dual b* (*final-dual r*) (*final-dual s*)
| *final-dual* (*CoTimes b r s*) = *bool-binop-dual* (¬ *b*) (*final-dual r*) (*final-dual s*)
| *final-dual* (*CoStar b* -) = *b*
| *final-dual* (*CoPr* - *r*) = *final-dual r*

context *alphabet*
begin

primrec *wf-dual* :: *nat* ⇒ *'b rexp-dual* ⇒ *bool*
where
wf-dual n (*CoZero* -) = *True* |
wf-dual n (*CoOne* -) = *True* |
wf-dual n (*CoAtom* - *a*) = (*wf-atom n a*) |
wf-dual n (*CoPlus* - *r s*) = (*wf-dual n r* ∧ *wf-dual n s*) |
wf-dual n (*CoTimes* - *r s*) = (*wf-dual n r* ∧ *wf-dual n s*) |
wf-dual n (*CoStar* - *r*) = *wf-dual n r* |
wf-dual n (*CoPr* - *r*) = *wf-dual* (*n* + *1*) *r*

lemma *wf-dual-PLUS-dual*[*simp*]:
wf-dual n (*CoPLUS-dual b xs*) = (∀ *r* ∈ *set xs*. *wf-dual n r*)
by (*induct xs rule*: *list-singleton-induct*) *auto*

abbreviation *set-unop-dual n b A* ≡ *if b then A else lists* (Σ *n*) − *A*

end

context *project*
begin

primrec *lang-dual* :: *nat* ⇒ *'b rexp-dual* => *'a lang* **where**
lang-dual n (*CoZero b*) = *set-unop-dual n b* {} |
lang-dual n (*CoOne b*) = *set-unop-dual n b* {[]} |
lang-dual n (*CoAtom b a*) = *set-unop-dual n b* {[*x*] | *x*. *lookup a x* ∧ *x* ∈ Σ *n*} |
lang-dual n (*CoPlus b r s*) = *set-binop-dual b* (*lang-dual n r*) (*lang-dual n s*) |

lang-dual n (CoTimes b r s) = set-unop-dual n b
(set-unop-dual n b (lang-dual n r) @@ set-unop-dual n b (lang-dual n s)) |
lang-dual n (CoStar b r) = set-unop-dual n b (star (set-unop-dual n b (lang-dual n r))) |
lang-dual n (CoPr b r) = set-unop-dual n b (map project ‘ (set-unop-dual (n + 1) b (lang-dual (n + 1) r)))

lemma *wf-dual-lang-dual-wf-word*: *wf-dual n r ⟹ ∀ w ∈ lang-dual n r. wf-word n w*
by (*induct r arbitrary*: *n*) (*auto elim*: *rev-subsetD*[*OF - conc-mono*] *star-induct intro*: *iffD2*[*OF wf-word*] *wf-word-map-project*)

lemma *lang-dual-subset-lists*: *wf-dual n r ⟹ lang-dual n r ⊆ lists (Σ n)*
proof (*induct r arbitrary*: *n*)
case (*CoPr b r*) **thus** *?case* **by** (*cases b*) (*fastforce intro*!: *project*)+
qed (*auto simp*: *conc-subset-lists star-subset-lists*)

lemma *lang-dual-final-dual*: *final-dual r = ([] ∈ lang-dual n r)*
by (*induct r arbitrary*: *n*) (*auto intro*: *concI*[*of* [] - [], *simplified*])

lemma *lang-dual-PLUS-dual*[*simp*]:
lang-dual n (CoPLUS-dual True xs) = (⋃ r ∈ set xs. lang-dual n r)
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *lang-dual-CoPLUS-dual*[*simp*]:
lang-dual n (CoPLUS-dual False xs) = (if xs = [] then lists (Σ n) else ⋂ r ∈ set xs. lang-dual n r)
by (*induct xs rule*: *list-singleton-induct*) *auto*

end

context *embed*
begin

primrec *lderiv-dual* :: *'a ⇒ 'b rexp-dual ⇒ 'b rexp-dual* **where**
lderiv-dual - (CoZero b) = (CoZero b)
| lderiv-dual - (CoOne b) = (CoZero b)
| lderiv-dual a (CoAtom b c) = (if lookup c a then CoOne b else CoZero b)
| lderiv-dual a (CoPlus b r s) = CoPlus b (lderiv-dual a r) (lderiv-dual a s)
| lderiv-dual a (CoTimes b r s) =
(let r's = CoTimes b (lderiv-dual a r) s
in if bool-unop-dual b (final-dual r) then CoPlus b r's (lderiv-dual a s) else r's)
| lderiv-dual a (CoStar b r) = CoTimes b (lderiv-dual a r) (CoStar b r)
| lderiv-dual a (CoPr b r) = CoPr b (CoPLUS-dual b (map (λa'. lderiv-dual a' r) (embed a)))

primrec *lderivs-dual* **where**
lderivs-dual [] r = r
| lderivs-dual (w#ws) r = lderivs-dual ws (lderiv-dual w r)

lemma *wf-dual-lderiv-dual*[*simp*]: *wf-dual n r* $\Longrightarrow$ *wf-dual n* (*lderiv-dual w r*)
by (*induct r arbitrary*: *n w*) (*auto simp add*: *Let-def*)

lemma *wf-dual-lderivs-dual*[*simp*]: *wf-dual n r* $\Longrightarrow$ *wf-dual n* (*lderivs-dual ws r*)
by (*induct ws arbitrary*: *r*) (*auto intro*: *wf-dual-lderiv-dual*)

lemma *lang-dual-lderiv-dual*: $[\![$*wf-dual n r*; *w* $\in$ Σ *n*$]\!] \Longrightarrow$
lang-dual n (*lderiv-dual w r*) = *lQuot w* (*lang-dual n r*)
proof (*induct r arbitrary*: *n w*)
case (*CoPr b r*)
hence $*$: *wf-dual* (*Suc n*) *r* $\bigwedge$*w'*. *w'* $\in$ *set* (*embed w*) $\Longrightarrow$ *w'* $\in$ Σ (*Suc n*) **by** (*auto simp*: *embed*)
then show *?case* **using** *lQuot-map-project*[*OF CoPr*(*3*) *lang-dual-subset-lists*[*OF* $*$(*1*)]]
lQuot-map-project[*OF CoPr*(*3*) *Diff-subset*, *of lang-dual* (*n* + *1*) *r*]
by (*simp-all add*: *CoPr*(*1*,*3*))
qed (*auto 0 3 simp*: *Let-def lang-dual-final-dual*[*symmetric*])

lemma *lang-dual-lderivs-dual*: $[\![$*wf-dual n r*; *wf-word n ws*$]\!] \Longrightarrow$
lang-dual n (*lderivs-dual ws r*) = *lQuots ws* (*lang-dual n r*)
by (*induct ws arbitrary*: *r*) (*auto simp*: *lang-dual-lderiv-dual*)

corollary *lderivs-dual-final-dual*:
assumes *wf-dual n r wf-word n ws*
shows *final-dual* (*lderivs-dual ws r*) $\longleftrightarrow$ *ws* $\in$ *lang-dual n r*
using *lang-dual-lderivs-dual*[*OF assms*] *lang-dual-final-dual*[*of lderivs-dual ws r n*] **by** *auto*

end

fun *pnCoPlus* :: *bool* $\Rightarrow$ *'a*::*linorder rexp-dual* $\Rightarrow$ *'a rexp-dual* $\Rightarrow$ *'a rexp-dual*
where
pnCoPlus b1 (*CoZero b2*) *r* = (*if b1* = *b2 then r else CoZero b2*)
| *pnCoPlus b1 r* (*CoZero b2*) = (*if b1* = *b2 then r else CoZero b2*)
| *pnCoPlus b1* (*CoPlus b2 r s*) *t* =
(*if b1* = *b2 then pnCoPlus b2 r* (*pnCoPlus b2 s t*) *else CoPlus b1* (*CoPlus b2 r s*) *t*)
| *pnCoPlus b1 r* (*CoPlus b2 s t*) =
(*if b1* = *b2 then*
(*if r* = *s then* (*CoPlus b2 s t*)
else if r $\leq$ *s then CoPlus b2 r* (*CoPlus b2 s t*)
else CoPlus b2 s (*pnCoPlus b2 r t*))
else CoPlus b1 r (*CoPlus b2 s t*))
| *pnCoPlus b r s* =
(*if r* = *s then r*
else if r $\leq$ *s then CoPlus b r s*
else CoPlus b s r)

lemma (**in** *alphabet*) *wf-dual-pnCoPlus*[*simp*]: ⟦*wf-dual n r*; *wf-dual n s*⟧ ⟹ *wf-dual n* (*pnCoPlus b r s*)
by (*induct b r s rule*: *pnCoPlus.induct*) *auto*

lemma (**in** *project*) *lang-dual-pnCoPlus*[*simp*]: ⟦*wf-dual n r*; *wf-dual n s*⟧ ⟹
lang-dual n (*pnCoPlus b r s*) = *lang-dual n* (*CoPlus b r s*)
proof (*induct b r s rule*: *pnCoPlus.induct*)
case *1* **thus** *?case* **by** (*auto dest*: *lang-dual-subset-lists*)
next
case *2-1* **thus** *?case* **by** *auto*
next
case *2-2* **thus** *?case* **by** *auto*
next
case *2-3* **thus** *?case* **by** (*auto dest*: *lang-dual-subset-lists*)
next
case *2-4* **thus** *?case* **by** (*auto dest*!: *lang-dual-subset-lists dest*:
subsetD[*OF conc-subset-lists*, *unfolded in-lists-conv-set*, *rotated* −*1*])
next
case *2-5* **thus** *?case* **by** (*auto dest*!: *lang-dual-subset-lists dest*:
subsetD[*OF star-subset-lists*, *unfolded in-lists-conv-set*, *rotated* −*1*])
next
case *2-6* **thus** *?case* **by** (*auto 4 4 dest*!: *lang-dual-subset-lists intro*: *project*)
next
case *3-1* **thus** *?case* **by** *auto*
next
case *3-2* **thus** *?case* **by** *auto*
next
case *3-3* **thus** *?case* **by** *auto*
next
case *3-4* **thus** *?case* **by** *auto*
next
case *3-5* **thus** *?case* **by** *auto*
next
case *3-6* **thus** *?case* **by** *auto*
next
case *4-1* **thus** *?case* **by** *auto*
next
case *4-2* **thus** *?case* **by** *auto*
next
case *4-3* **thus** *?case* **by** *auto*
next
case *4-4* **thus** *?case* **by** *auto*
next
case *4-5* **thus** *?case* **by** *auto*
next
case *5-1* **thus** *?case* **by** *auto*
next
case *5-2* **thus** *?case* **by** *auto*
next

case *5-3* **thus** *?case* **by** *auto*
next
case *5-4* **thus** *?case* **by** *auto*
next
case *5-5* **thus** *?case* **by** *auto*
next
case *5-6* **thus** *?case* **by** *auto*
next
case *5-7* **thus** *?case* **by** *auto*
next
case *5-8* **thus** *?case* **by** *auto*
next
case *5-9* **thus** *?case* **by** *auto*
next
case *5-10* **thus** *?case*
by *auto* (*metis* (*no-types*, *hide-lams*) *Cons-in-lists-iff Diff-iff imageI list.simps*(*8*) *list.simps*(*9*) *lists.Nil*)+
next
case *5-11* **thus** *?case* **by** *auto*
next
case *5-12* **thus** *?case* **by** *auto*
next
case *5-13* **thus** *?case* **by** *auto*
next
case *5-14* **thus** *?case* **by** *auto*
next
case *5-15* **thus** *?case* **by** *auto*
next
case *5-16* **thus** *?case* **by** *auto*
next
case *5-17* **thus** *?case* **by** *auto*
next
case *5-18* **thus** *?case* **by** *auto*
next
case *5-19* **thus** *?case* **by** (*auto dest*!: *lang-dual-subset-lists dest*:
subsetD[*OF star-subset-lists*, *unfolded in-lists-conv-set*, *rotated* −*1*])
next
case *5-20* **thus** *?case* **by** *auto*
next
case *5-21* **thus** *?case* **by** *auto*
next
case *5-22* **thus** *?case*
by *auto* (*metis* (*no-types*, *hide-lams*) *Cons-in-lists-iff Diff-iff imageI list.simps*(*8*) *list.simps*(*9*) *lists.Nil*)+
next
case *5-23* **thus** *?case* **by** *auto*
next
case *5-24* **thus** *?case* **by** *auto*
next

case *5-25* **thus** *?case* **by** *auto*
qed

fun *pnCoTimes* :: *bool* ⇒ *'a::linorder rexp-dual* ⇒ *'a rexp-dual* ⇒ *'a rexp-dual* **where**
pnCoTimes b1 (*CoZero b2*) *r* = (*if b1* = *b2 then CoZero b1 else CoTimes b1* (*CoZero b2*) *r*)
| *pnCoTimes b1* (*CoOne b2*) *r* = (*if b1* = *b2 then r else CoTimes b1* (*CoOne b2*) *r*)
| *pnCoTimes b1* (*CoPlus b2 r s*) *t* = (*if b1* = *b2 then pnCoPlus b2* (*pnCoTimes b2 r t*) (*pnCoTimes b2 s t*)
else CoTimes b1 (*CoPlus b2 r s*) *t*)
| *pnCoTimes b r s* = *CoTimes b r s*

lemma (**in** *alphabet*) *wf-dual-pnCoTimes*[*simp*]: ⟦*wf-dual n r*; *wf-dual n s*⟧ ⟹ *wf-dual n* (*pnCoTimes b r s*)
by (*induct b r s rule*: *pnCoTimes.induct*) *auto*

lemma (**in** *project*) *lang-dual-pnCoTimes*[*simp*]: ⟦*wf-dual n r*; *wf-dual n s*⟧ ⟹ *lang-dual n* (*pnCoTimes b r s*) = *lang-dual n* (*CoTimes b r s*)
apply (*induct b r s rule*: *pnCoTimes.induct*)
apply (*auto*, *auto dest*!: *lang-dual-subset-lists dest*: *project*
subsetD[*OF star-subset-lists*, *unfolded in-lists-conv-set*, *rotated* −*1*]
subsetD[*OF conc-subset-lists*, *unfolded in-lists-conv-set*, *rotated* −*1*])
by (*metis* (*full-types*) *Diff-iff conc-epsilon*(*1*) *double-diff empty-subsetI in-listsI insert-subset lists.Nil subset-refl*)

fun *pnCoPr* :: *bool* ⇒ *'a::linorder rexp-dual* ⇒ *'a rexp-dual* **where**
pnCoPr b1 (*CoZero b2*) = (*if b1* = *b2 then CoZero b2 else CoPr b1* (*CoZero b2*))
| *pnCoPr b1* (*CoOne b2*) = (*if b1* = *b2 then CoOne b2 else CoPr b1* (*CoOne b2*))
| *pnCoPr b1* (*CoPlus b2 r s*) = (*if b1* = *b2 then pnCoPlus b2* (*pnCoPr b2 r*) (*pnCoPr b2 s*)
else CoPr b1 (*CoPlus b2 r s*))
| *pnCoPr b r* = *CoPr b r*

lemma (**in** *alphabet*) *wf-dual-pnCoPr*[*simp*]: *wf-dual* (*Suc n*) *r* ⟹ *wf-dual n* (*pnCoPr b r*)
by (*induct b r rule*: *pnCoPr.induct*) *auto*

lemma (**in** *project*) *lang-dual-pnCoPr*[*simp*]: *wf-dual* (*Suc n*) *r* ⟹ *lang-dual n* (*pnCoPr b r*) = *lang-dual n* (*CoPr b r*)
by (*induct b r rule*: *pnCoPr.induct*) *auto*

primrec *pnorm-dual* :: *'a::linorder rexp-dual* ⇒ *'a rexp-dual* **where**
pnorm-dual (*CoZero b*) = (*CoZero b*)
| *pnorm-dual* (*CoOne b*) = (*CoOne b*)
| *pnorm-dual* (*CoAtom b a*) = (*CoAtom b a*)
| *pnorm-dual* (*CoPlus b r s*) = *pnCoPlus b* (*pnorm-dual r*) (*pnorm-dual s*)

| *pnorm-dual* (*CoTimes b r s*) = *pnCoTimes b* (*pnorm-dual r*) *s*
| *pnorm-dual* (*CoStar b r*) = *CoStar b r*
| *pnorm-dual* (*CoPr b r*) = *pnCoPr b* (*pnorm-dual r*)

lemma (**in** *alphabet*) *wf-dual-pnorm-dual*[*simp*]: *wf-dual n r* ⟹ *wf-dual n* (*pnorm-dual r*)
by (*induct r arbitrary*: *n*) *auto*

lemma (**in** *project*) *lang-dual-pnorm-dual*[*simp*]: *wf-dual n r* ⟹ *lang-dual n* (*pnorm-dual r*) = *lang-dual n r*
by (*induct r arbitrary*: *n*) *auto*

primrec *CoNot* **where**
CoNot (*CoZero b*) = *CoZero* (¬ *b*)
| *CoNot* (*CoOne b*) = *CoOne* (¬ *b*)
| *CoNot* (*CoAtom b a*) = *CoAtom* (¬ *b*) *a*
| *CoNot* (*CoPlus b r s*) = *CoPlus* (¬ *b*) (*CoNot r*) (*CoNot s*)
| *CoNot* (*CoTimes b r s*) = *CoTimes* (¬ *b*) (*CoNot r*) (*CoNot s*)
| *CoNot* (*CoStar b r*) = *CoStar* (¬ *b*) (*CoNot r*)
| *CoNot* (*CoPr b r*) = *CoPr* (¬ *b*) (*CoNot r*)

primrec *rexp-dual-of* **where**
rexp-dual-of Zero = *CoZero True*
| *rexp-dual-of Full* = *CoZero False*
| *rexp-dual-of One* = *CoOne True*
| *rexp-dual-of* (*Atom a*) = *CoAtom True a*
| *rexp-dual-of* (*Plus r s*) = *CoPlus True* (*rexp-dual-of r*) (*rexp-dual-of s*)
| *rexp-dual-of* (*Times r s*) = *CoTimes True* (*rexp-dual-of r*) (*rexp-dual-of s*)
| *rexp-dual-of* (*Star r*) = *CoStar True* (*rexp-dual-of r*)
| *rexp-dual-of* (*Not r*) = *CoNot* (*rexp-dual-of r*)
| *rexp-dual-of* (*Inter r s*) = *CoPlus False* (*rexp-dual-of r*) (*rexp-dual-of s*)
| *rexp-dual-of* (*Pr r*) = *CoPr True* (*rexp-dual-of r*)

lemma (**in** *alphabet*) *wf-dual-CoNot*[*simp*]: *wf-dual n r* ⟹ *wf-dual n* (*CoNot r*)
by (*induct r arbitrary*: *n*) *auto*

lemma (**in** *project*) *lang-dual-CoNot*[*simp*]: *wf-dual n r* ⟹ *lang-dual n* (*CoNot r*) = *lists* (Σ *n*) − *lang-dual n r*
apply (*induct r arbitrary*: *n*)
apply (*auto dest*!: *lang-dual-subset-lists simp*: *double-diff intro*!: *project*)
apply *force*
apply (*metis* (*full-types*) *Diff-subset contra-subsetD in-listsD star-subset-lists*)
done

lemma (**in** *alphabet*) *wf-dual-rexp-dual-of*[*simp*]: *wf n r* ⟹ *wf-dual n* (*rexp-dual-of r*)
by (*induct r arbitrary*: *n*) *auto*

lemma (**in** *project*) *lang-dual-rexp-dual-of*[*simp*]: *wf n r* ⟹ *lang-dual n* (*rexp-dual-of*

r) = *lang n r*
by (*induct r arbitrary*: *n*) *auto*

end

6 Deciding Equivalence of Π-Extended Regular Expressions

lemma *image2p-in-rel*: *BNF-Greatest-Fixpoint.image2p f g* (*in-rel R*) = *in-rel* (*map-prod f g* ‘ *R*)
unfolding *image2p-def fun-eq-iff* **by** *auto*

lemma *image2p-apply*: *BNF-Greatest-Fixpoint.image2p f g R x y* = (∃ *x′ y′*. *R x′ y′* ∧ *f x′* = *x* ∧ *g y′* = *y*)
unfolding *image2p-def fun-eq-iff* **by** *auto*

lemma *rtrancl-fold-product*:
shows {((*r*, *s*), (*f a r*, *f a s*))| *r s a*. *a* ∈ *A*} ^* =
{((*r*, *s*), (*fold f w r*, *fold f w s*))| *r s w*. *w* ∈ *lists A*} (**is** *?L* = *?R*)
proof−
{ fix *x* :: (′*a* × ′*a*) × ′*a* × ′*a*
obtain *r s r′ s′* **where** *x*: *x* = ((*r*, *s*), (*r′*, *s′*)) **by** (*cases x*) *auto*
have ((*r*, *s*), (*r′*, *s′*)) ∈ *?L* ⟹ ((*r*, *s*), (*r′*, *s′*)) ∈ *?R*
proof(*induction rule*: *converse-rtrancl-induct2*)
case *refl* **show** *?case* **by**(*force intro*!: *fold-simps*(*1*)[*symmetric*])
next
case *step* **thus** *?case* **by**(*force intro*!: *fold-simps*(*2*)[*symmetric*])
qed
with *x* **have** *x* ∈ *?L* ⟹ *x* ∈ *?R* **by** *simp*
} moreover
{ fix *x* :: (′*a* × ′*a*) × ′*a* × ′*a*
obtain *r s r′ s′* **where** *x*: *x* = ((*r*, *s*), (*r′*, *s′*)) **by** (*cases x*) *auto*
{ fix *w* **have** ∀ *x*∈*set w*. *x* ∈ *A* ⟹ ((*r*, *s*), *fold f w r*, *fold f w s*) ∈ *?L*
proof(*induction w rule*: *rev-induct*)
case *Nil* **show** *?case* **by** *simp*
next
case *snoc* **thus** *?case* **by** (*auto elim*!: *rtrancl-into-rtrancl*)
qed
}
hence ((*r*, *s*), (*r′*, *s′*)) ∈ *?R* ⟹ ((*r*, *s*), (*r′*, *s′*)) ∈ *?L* **by** *auto*
with *x* **have** *x* ∈ *?R* ⟹ *x* ∈ *?L* **by** *simp*
} ultimately show *?thesis* **by** *blast*
qed

lemma *in-fold-lQuot*: *v* ∈ *fold lQuot w L* ⟷ *w* @ *v* ∈ *L*
by (*induct w arbitrary*: *L*) (*simp-all add*: *lQuot-def*)

lemma (**in** *project*) *lang-eq-ext*: ⟦*wf n r*; *wf n s*⟧ ⟹ (*lang n r* = *lang n s*) = (∀ *w* ∈ *lists*(Σ *n*). *w* ∈ *lang n r* ⟷ *w* ∈ *lang n s*)
by (*auto dest*!: *lang-subset-lists*)

lemma (**in** *project*) *lang-eq-ext-Nil-fold-Deriv*:
fixes *r s n*
assumes *WF*: *wf n r wf n s*
defines 𝔅 ≡ {(*fold lQuot w* (*lang n r*), *fold lQuot w* (*lang n s*))| *w*. *w*∈*lists* (Σ *n*)}
shows *lang n r* = *lang n s* ⟷ (∀(*K*, *L*) ∈ 𝔅. [] ∈ *K* ⟷ [] ∈ *L*)
unfolding *lang-eq-ext*[*OF WF*] 𝔅-*def*
by (*subst* (*1 2*) *in-fold-lQuot*[*of* [], *simplified*, *symmetric*]) *auto*

locale *rexp-DA* = *project set o* σ *wf-atom project lookup*
for σ :: *nat* ⇒ 'a *list*
and *wf-atom* :: *nat* ⇒ 'b :: *linorder* ⇒ *bool*
and *project* :: 'a ⇒ 'a
and *lookup* :: 'b ⇒ 'a ⇒ *bool* +
fixes *init* :: 'b *rexp* ⇒ 's
fixes *delta* :: 'a ⇒ 's ⇒ 's
fixes *final* :: 's ⇒ *bool*
fixes *wf-state* :: 's ⇒ *bool*
fixes *post* :: 's ⇒ 's
fixes *L* :: 's ⇒ 'a *lang*
fixes *n* :: *nat*
assumes *L-init*[*simp*]: *wf n r* ⟹ *L* (*init r*) = *lang n r*
assumes *L-delta*[*simp*]: ⟦*a* ∈ *set* (σ *n*); *wf-state s*⟧ ⟹ *L* (*delta a s*) = *lQuot a* (*L s*)
assumes *final-iff-Nil*[*simp*]: *final s* ⟷ [] ∈ *L s*
assumes *L-wf-state*[*dest*]: *wf-state s* ⟹ *L s* ⊆ *lists* (*set* (σ *n*))
assumes *init-wf-state*[*simp*]: *wf n r* ⟹ *wf-state* (*init r*)
assumes *delta-wf-state*[*simp*]: ⟦*a* ∈ *set* (σ *n*); *wf-state s*⟧ ⟹ *wf-state* (*delta a s*)
assumes *L-post*[*simp*]: *wf-state s* ⟹ *L* (*post s*) = *L s*
assumes *wf-state-post*[*simp*]: *wf-state s* ⟹ *wf-state* (*post s*)
begin

lemma *L-deltas*[*simp*]: ⟦*wf-word n w*; *wf-state s*⟧ ⟹ *L* (*fold delta w s*) = *fold lQuot w* (*L s*)
by (*induction w arbitrary*: *s*) *auto*

definition *progression* (**infix** → *60*) **where**
R → *S* = (∀ *s1 s2*. *R s1 s2* ⟶ *wf-state s1* ∧ *wf-state s2* ∧ *final s1* = *final s2* ∧
(∀ *x* ∈ *set* (σ *n*). *BNF-Greatest-Fixpoint.image2p post post S* (*post* (*delta x s1*)) (*post* (*delta x s2*))))

lemma *SUPR-progression*[*intro*!]: ∀ *n*. ∃ *m*. *X n* → *Y m* ⟹ (*SUP n*. *X n*) → (*SUP n*. *Y n*)
unfolding *progression-def image2p-def* **by** *fastforce*

definition *bisimulation* **where**
bisimulation R = R → R

definition *bisimulation-upto* **where**
bisimulation-upto R f = R → f R

declare *image2pI*[*intro*!] *image2pE*[*elim*!]
lemmas *bisim-def = bisimulation-def progression-def*
lemmas *bisim-upto-def = bisimulation-upto-def progression-def*

definition *compatible* **where**
compatible f = (mono f ∧ (∀ R S. R → S ⟶ f R → f S))

lemmas *compat-def = compatible-def progression-def*

lemma *bisimulation-upto-bisimulation*:
assumes *compatible f bisimulation-upto R f*
obtains *S* **where** *bisimulation S R ≤ S*
proof
{ fix *n* **from** *assms* **have** *(f^^n) R → (f^^Suc n) R*
by *(induct n) (auto simp*: *bisimulation-upto-def compatible-def)* **}**
then show *bisimulation (SUP n. (f^^n) R)*
unfolding *bisimulation-def* **by** *(auto simp del*: *funpow.simps)*
show *R ≤ (SUP n. (f^^n) R)* **by** *(auto intro*!: *exI*[*of - 0*])
qed

lemma *bisimulation-eqL*: *bisimulation (λs1 s2. wf-state s1 ∧ wf-state s2 ∧ L s1 = L s2)*
unfolding *bisim-def* **by** *(auto simp*: *lQuot-def)*

lemma *coinduction*:
assumes *bisim*[*unfolded bisim-def*]: *bisimulation R* **and**
WF: *wf-state s1 wf-state s2* **and** *R*: *R s1 s2*
shows *L s1 = L s2*
proof *(rule set-eqI)*
fix *w*
from *R WF* **show** *w ∈ L s1 ⟷ w ∈ L s2*
proof *(induction w arbitrary*: *s1 s2)*
case *Nil* **then show** *?case* **using** *bisim* **by** *simp*
next
case *(Cons a w s1 s2)*
show *?case*
proof *cases*
assume *a*: *a ∈ set (σ n)*
with ‹*R s1 s2*› **obtain** *s1' s2'* **where** *R s1' s2' wf-state s1' wf-state s2'* **and**
∗[*symmetric*]: *post s1' = post (delta a s1) post s2' = post (delta a s2)*
using *bisim* **unfolding** *image2p-apply* **by** *blast*
then have *w ∈ L (post (delta a s1)) ⟷ w ∈ L (post (delta a s2))*

```
      unfolding ∗ using Cons.IH[of s1′ s2′] by simp
    with a Cons.prems(2,3) show ?case by (simp add: lQuot-def)
  next
    assume a ∉ set (σ n)
    thus ?case using Cons.prems bisim by force
  qed
 qed
qed

lemma coinduction-upto:
  assumes bisimulation-upto R f and WF: wf-state s1 wf-state s2 and R s1 s2
compatible f
 shows L s1 = L s2
proof (rule bisimulation-upto-bisimulation[OF assms(5,1)])
 fix S assume R ≤ S
 assume bisimulation S
 then show L s1 = L s2
 proof (rule coinduction[OF - WF])
  from ‹R ≤ S› ‹R s1 s2› show S s1 s2 by blast
 qed
qed

fun test-invariant where
 test-invariant (ws, - :: ('s × 's) list , - :: 's rel) = (case ws of [] ⇒ False | (w::'a
list,p,q)#- ⇒ final p = final q)
fun test where test (ws, - :: 's rel) = (case ws of [] ⇒ False | (p,q)#- ⇒ final p
= final q)

fun step-invariant where step-invariant (ws, ps, N) =
  (let
    (w, r, s) = hd ws;
    ps′ = (r, s) # ps;
    succs = map (λa.
      let r′ = delta a r; s′ = delta a s
      in ((a # w, r′, s′), (post r′, post s′))) (σ n);
    new = remdups′ snd (filter (λ(-, rs). rs ∉ N) succs);
    ws′ = tl ws @ map fst new;
    N′ = set (map snd new) ∪ N
  in (ws′, ps′, N′))

fun step where step (ws, N) =
  (let
    (r, s) = hd ws;
    succs = map (λa.
      let r′ = delta a r; s′ = delta a s
      in ((r′, s′), (post r′, post s′))) (σ n);
    new = remdups′ snd (filter (λ(-, rs). rs ∉ N) succs)
  in (tl ws @ map fst new, set (map snd new) ∪ N))
```

definition *closure-invariant* **where** *closure-invariant* = *while-option test-invariant step-invariant*
definition *closure* **where** *closure* = *while-option test step*

definition *invariant* **where**
invariant r s = (λ(*ws*, *ps*, *N*).
(*r*, *s*) ∈ *snd* ‘ *set ws* ∪ *set ps* ∧
distinct (*map snd ws* @ *ps*) ∧
bij-betw (*map-prod post post*) (*set* (*map snd ws* @ *ps*)) *N* ∧
(∀ (*w*, *r'*, *s'*) ∈ *set ws*. *fold delta* (*rev w*) *r* = *r'* ∧ *fold delta* (*rev w*) *s* = *s'* ∧
wf-word n (*rev w*) ∧ *wf-state r'* ∧ *wf-state s'*) ∧
(∀ (*r'*, *s'*) ∈ *set ps*. (∃ *w*. *fold delta w r* = *r'* ∧ *fold delta w s* = *s'*) ∧
wf-state r' ∧ *wf-state s'* ∧ (*final r'* ⟷ *final s'*) ∧
(∀ *a*∈*set* (σ *n*). (*post* (*delta a r'*), *post* (*delta a s'*)) ∈ *N*)))

lemma *invariant-start*:
⟦*wf-state r*; *wf-state s*⟧ ⟹ *invariant r s* ([([], *r*, *s*)], [], {(*post r*, *post s*)})
by (*auto simp add*: *invariant-def bij-betw-def*)

lemma *step-invariant-mono*:
assumes *step-invariant* (*ws*, *ps*, *N*) = (*ws'*, *ps'*, *N'*)
shows *snd* ‘ *set ws* ∪ *set ps* ⊆ *snd* ‘ *set ws'* ∪ *set ps'*
using *assms* **proof** (*intro subsetI*, *elim UnE*)
fix *x* **assume** *x* ∈ *snd* ‘*set ws*
with *assms* **show** *x* ∈ *snd* ‘ *set ws'* ∪ *set ps'*
proof (*cases x* = *snd* (*hd ws*))
case *False* **with** ‹*x* ∈ *image snd* (*set ws*)› **have** *x* ∈ *snd* ‘ *set* (*tl ws*) **by** (*cases ws*) *auto*
with *assms* **show** *?thesis* **by** (*auto split*: *prod.splits simp*: *Let-def*)
qed (*auto split*: *prod.splits simp*: *Let-def*)
qed (*auto split*: *prod.splits simp*: *Let-def*)

lemma *step-invatiant-unfold*: *step-invariant* (*w* # *ws*, *ps*, *N*) = (*ws'*, *ps'*, *N'*) ⟹
(∃ *xs r s*.
w = (*xs*, *r*, *s*) ∧ *ps'* = (*r*, *s*) # *ps* ∧
ws' = *ws* @ *remdups'* (*map-prod post post o snd*) (*filter* (λ(-, *p*). *map-prod post post p* ∉ *N*)
(*map* (λ*a*. (*a*#*xs*, *delta a r*, *delta a s*)) (σ *n*))) ∧
N' = *set* (*map* (λ*a*. (*post* (*delta a r*), *post* (*delta a s*))) (σ *n*)) ∪ *N*)
by (*auto split*: *prod.splits dest*!: *mp-remdups'*
simp: *Let-def filter-map set-n-lists image-Collect image-image comp-def*)

lemma *invariant*: *invariant r s st* ⟹ *test-invariant st* ⟹ *invariant r s* (*step-invariant st*)
proof (*unfold invariant-def*, (*split prod.splits*)+, *elim case-prodE conjE*, *clarify*, *intro allI impI conjI*)
fix *ws ps N ws' ps' N'*
assume *test-invariant*: *test-invariant* (*ws*, *ps*, *N*)
and *step-invariant*: *step-invariant* (*ws*, *ps*, *N*) = (*ws'*, *ps'*, *N'*)

and *rs*: $(r, s) \in snd$ ‘ $set\ ws \cup set\ ps$
and *distinct*: $distinct\ (map\ snd\ ws\ @\ ps)$
and *bij*: $bij\text{-}betw\ (map\text{-}prod\ post\ post)\ (set\ (map\ snd\ ws\ @\ ps))\ N$
and *ws*: $\forall (w, r', s') \in set\ ws.\ fold\ delta\ (rev\ w)\ r = r' \wedge fold\ delta\ (rev\ w)\ s = s' \wedge$
$wf\text{-}word\ n\ (rev\ w) \wedge wf\text{-}state\ r' \wedge wf\text{-}state\ s'$
(**is** $\forall (w, r', s') \in set\ ws.\ ?ws\ w\ r'\ s'$)
and *ps*: $\forall (r', s') \in set\ ps.\ (\exists w.\ fold\ delta\ w\ r = r' \wedge fold\ delta\ w\ s = s') \wedge$
$wf\text{-}state\ r' \wedge wf\text{-}state\ s' \wedge (final\ r' \longleftrightarrow final\ s') \wedge$
$(\forall a{\in}set\ (\sigma\ n).\ (post\ (delta\ a\ r'), post\ (delta\ a\ s')) \in N)$
(**is** $\forall (r, s) \in set\ ps.\ ?ps\ r\ s\ N$)
from *test-invariant* **obtain** *x xs* **where** *ws-Cons*: $ws = x\ \#\ xs$ **by** (*cases ws*) *auto*
obtain $w\ r'\ s'$ **where** *x*: $x = (w, r', s')$ **and** *ps'*: $ps' = (r', s')\ \#\ ps$
and *ws'*: $ws' = xs\ @\ remdups'\ (map\text{-}prod\ post\ post\ o\ snd)$
$(filter\ (\lambda(\text{-}, p).\ map\text{-}prod\ post\ post\ p \notin N)$
$(map\ (\lambda a.\ (a\ \#\ w, delta\ a\ r', delta\ a\ s'))\ (\sigma\ n)))$
and *N'*: $N' = (set\ (map\ (\lambda a.\ (post\ (delta\ a\ r'), post\ (delta\ a\ s')))\ (\sigma\ n)) - N) \cup N$
using *step-invatiant-unfold*[*OF step-invariant*[*unfolded ws-Cons*]] **by** *blast*
hence *ws'ps'*: $set\ (map\ snd\ ws'\ @\ ps') =$
$set\ (remdups'\ (map\text{-}prod\ post\ post)\ (filter\ (\lambda p.\ map\text{-}prod\ post\ post\ p \notin N)$
$(map\ (\lambda a.\ (delta\ a\ r', delta\ a\ s'))\ (\sigma\ n)))) \cup (set\ (map\ snd\ ws\ @\ ps))$
unfolding *ws' ps' ws-Cons x* **by** (*auto dest*!: *mp-remdups' simp*: *filter-map image-image image-Un o-def*)
from *rs step-invariant* **show** $(r, s) \in snd$ ‘ $set\ ws' \cup set\ ps'$ **by** (*blast dest*: *step-invariant-mono*)

from *distinct ps' ws' ws-Cons x bij* **show** $distinct\ (map\ snd\ ws'\ @\ ps')$
by (*auto simp*: *bij-betw-def*
intro!: *imageI*[*of - - map-prod post post*] *distinct-remdups'-strong*
map-prod-imageI[*of - - - post post*]
dest!: *mp-remdups'*
elim: *image-eqI*[*of - snd*, *OF sym*[*OF snd-conv*]])

from *ps' ws' N' ws x bij* **show** $bij\text{-}betw\ (map\text{-}prod\ post\ post)\ (set\ (map\ snd\ ws'\ @\ ps'))\ N'$
unfolding *ws'ps' N'* **by** (*intro bij-betw-combine*[*OF - bij*]) (*auto simp*: *bij-betw-def map-prod-def*)

from *ws x ws-Cons* **have** *wr's'*: $?ws\ w\ r'\ s'$ **by** *auto*
with *ws ws-Cons* **show** $\forall (w, r', s') \in set\ ws'.\ ?ws\ w\ r'\ s'$ **unfolding** *ws'*
by (*auto dest*!: *mp-remdups' elim*!: *subsetD*)

from *ps wr's' test-invariant*[*unfolded ws-Cons x*] **show** $\forall (r', s') \in set\ ps'.\ ?ps\ r'\ s'\ N'$ **unfolding** *ps' N'*
by (*fastforce simp*: *image-Collect*)
qed

lemma *step-commute*: *ws* ≠ [] ⟹
(*case step-invariant* (*ws*, *ps*, *N*) *of* (*ws'*, *ps'*, *N'*) ⇒ (*map snd ws'*, *N'*)) = *step* (*map snd ws*, *N*)
apply (*auto split*: *prod.splits*)
apply (*auto simp only*: *step-invariant.simps step.simps Let-def map-apfst-remdups' filter-map list.map-comp apfst-def map-prod-def snd-conv id-def*)
apply (*auto simp*: *filter-map comp-def map-tl hd-map*)
apply (*intro image-eqI*, *auto*)+
done

lemma *closure-invariant-closure*:
map-option (λ(*ws*, *ps*, *N*). (*map snd ws*, *N*)) (*closure-invariant* (*ws*, *ps*, *N*)) = *closure* (*map snd ws*, *N*)
unfolding *closure-invariant-def closure-def*
by (*rule trans*[*OF while-option-commute*[*of - test - - step*]])
(*auto split*: *list.splits simp del*: *step-invariant.simps step.simps list.map simp*: *step-commute*)

lemma
assumes *result*: *closure-invariant* ([([], *init r*, *init s*)], [], {(*post* (*init r*), *post* (*init s*))}) =
Some(*ws*, *ps*, *N*) (**is** *closure-invariant* ([([], *?r*, *?s*)], -) = -)
and *WF*: *wf n r wf n s*
shows *closure-invariant-sound*: *ws* = [] ⟹ *lang n r* = *lang n s* **and**
counterexample: *ws* ≠ [] ⟹ *rev* (*fst* (*hd ws*)) ∈ *lang n r* ⟷ *rev* (*fst* (*hd ws*)) ∉ *lang n s*
proof −
from *WF* **have** *wf-state*: *wf-state ?r wf-state ?s* **by** *simp-all*
from *invariant invariant-start*[*OF wf-state*] **have** *invariant-ps*: *invariant ?r ?s* (*ws*, *ps*, *N*)
by (*rule while-option-rule*[*OF - result*[*unfolded closure-invariant-def*]])
{ assume *ws* = []
with *invariant-ps* **have** *bisimulation* (*in-rel* (*set ps*)) (*?r*, *?s*) ∈ *set ps*
by (*auto simp*: *bij-betw-def invariant-def bisimulation-def progression-def image2p-in-rel*)
with *wf-state* **have** *L ?r* = *L ?s* **by** (*auto dest*: *coinduction*)
with *WF* **show** *lang n r* = *lang n s* **by** *simp*
}
{ assume *ws* ≠ []
then obtain *w r' s' ws'* **where** *ws*: *ws* = (*w*, *r'*, *s'*) # *ws'* **by** (*cases ws*) *auto*
with *invariant-ps* **have** *r'* = *fold delta* (*rev w*) (*init r*) *s'* = *fold delta* (*rev w*) (*init s*)
wf-word n (*rev w*) **unfolding** *invariant-def* **by** *auto*
moreover have ¬ *test-invariant* ((*w*, *r'*, *s'*) # *ws'*, *ps*, *N*)
by (*rule while-option-stop*[*OF result*[*unfolded ws closure-invariant-def*]])
ultimately have *rev* (*fst* (*hd ws*)) ∈ *L ?r* ⟷ *rev* (*fst* (*hd ws*)) ∉ *L ?s*
unfolding *ws* **using** *wf-state* **by** (*simp add*: *in-fold-lQuot*)
with *WF* **show** *rev* (*fst* (*hd ws*)) ∈ *lang n r* ⟷ *rev* (*fst* (*hd ws*)) ∉ *lang n s* **by** *simp*

```
  }
qed

lemma closure-sound:
  assumes result: closure ([(init r, init s)], {(post (init r), post (init s))}) = Some
([], N)
  and WF: wf n r wf n s
  shows lang n r = lang n s
  using trans[OF closure-invariant-closure[of [([], init r, init s)], simplified] result]
  by (auto dest: closure-invariant-sound[OF - WF])

definition check-eqv where
  check-eqv r s =
    (let r' = init r; s' = init s in (case closure ([(r', s')], {(post r', post s')}) of
       Some ([], -) ⇒ True | - ⇒ False))

lemma check-eqv-sound:
  assumes check-eqv r s and WF: wf n r wf n s
  shows lang n r = lang n s
  using closure-sound assms
  by (auto simp: check-eqv-def Let-def split: option.splits list.splits)

definition counterexample where
  counterexample r s =
    (let r' = init r; s' = init s in (case closure-invariant ([([], r', s')], [], {(post r',
post s')}) of
       Some((w,-,-) # -, -) ⇒ Some (rev w) | - => None))

lemma counterexample-sound:
  assumes result: counterexample r s = Some w and WF: wf n r wf n s
  shows w ∈ lang n r ⟷ w ∉ lang n s
  using assms unfolding counterexample-def Let-def
  by (auto dest!: counterexample[of r s] split: option.splits list.splits)
```

Auxiliary exacutable functions:

```
definition reachable :: 'b rexp ⇒ 's set where
  reachable s = snd (the (rtrancl-while (λ-. True) (λs. map (λa. post (delta a s))
(σ n)) (init s)))

definition automaton :: 'b rexp ⇒ (('s * 'a) * 's) set where
  automaton s =
    snd (the
    (let i = init s;
         start = (([i], {post i}), {});
         test-invariant = λ((ws, Z), A). ws ≠ [];
         step-invariant = λ((ws, Z), A).
           (let s = hd ws;
                new-edges = map (λa. ((s, a), delta a s)) (σ n);
                new = remdups (filter (λss. post ss ∉ Z) (map snd new-edges))
```

```
      in ((new @ tl ws, post ‘ set new ∪ Z), set new-edges ∪ A))
  in while-option test-invariant step-invariant start))

definition match :: 'b rexp ⇒ 'a list ⇒ bool where
  match s w = final (fold delta w (init s))

lemma match-correct: ⟦wf-word n w; wf n s⟧ ⟹ match s w ⟷ w ∈ lang n s
  unfolding match-def
  by (induct w arbitrary: s) (auto simp: in-fold-lQuot lQuot-def)

end

locale rexp-DFA = rexp-DA σ wf-atom project lookup init delta final wf-state post
L n
  for σ :: nat ⇒ 'a list
  and wf-atom :: nat ⇒ 'b :: linorder ⇒ bool
  and project :: 'a ⇒ 'a
  and lookup :: 'b ⇒ 'a ⇒ bool
  and init :: 'b rexp ⇒ 's
  and delta :: 'a ⇒ 's ⇒ 's
  and final :: 's ⇒ bool
  and wf-state :: 's ⇒ bool
  and post :: 's ⇒ 's
  and L :: 's ⇒ 'a lang
  and n :: nat +
assumes fin: finite {fold delta w (init s) | w. True}
begin

abbreviation Reachable s ≡ {fold delta w (init s) | w. True}

lemma closure-invariant-termination:
  assumes WF: wf n r wf n s
   and result: closure-invariant ([([], init r, init s)], [], {(post (init r), post (init
s))}) = None
    (is closure-invariant ([([], ?r, ?s)], -) = None is ?cl = None)
  shows False
proof −
  let ?D =  post ‘ Reachable r × post ‘ Reachable s
  let ?X = λps. ?D − map-prod post post ‘ set ps
  let ?f = λ(ws, ps, N). card (?X ps)
  have ∃ st. ?cl = Some st unfolding closure-invariant-def
  proof (rule measure-while-option-Some[of invariant ?r ?s - - ?f], intro conjI)
    fix st assume base: invariant ?r ?s st and test-invariant st
    hence step: invariant ?r ?s (step-invariant st) by (rule invariant)
    obtain ws ps N where st: st = (ws, ps, N) by (cases st) blast
    hence finite (?X ps) by (blast intro: finite-cartesian-product fin)
   moreover obtain ws' ps' N' where step-invariant: step-invariant (ws, ps, N)
= (ws', ps', N')
      by (cases step-invariant (ws, ps, N)) blast
```

```
    moreover
    { have map-prod post post ‘ set ps ⊆ ?D using base[unfolded st invariant-def]
by fast
      moreover
      have map-prod post post ‘ set ps′ ⊆ ?D using step[unfolded st step-invariant
invariant-def]
        by fast
      moreover
      { have distinct (map snd ws @ ps) inj-on (map-prod post post) (set (map snd
ws @ ps))
          using base[unfolded st invariant-def] by (auto simp: bij-betw-def)
        hence distinct (map (map-prod post post) (map snd ws @ ps)) unfolding
distinct-map ..
        hence map-prod post post ‘ set ps ⊂ map-prod post post ‘ set (snd (hd ws)
# ps)
          using ‹test-invariant st› st by (cases ws) (simp-all, blast)
       moreover have map-prod post post ‘ set ps′ = map-prod post post ‘ set (snd
(hd ws) # ps)
          using step-invariant by (auto split: prod.splits)
       ultimately have map-prod post post ‘ set ps ⊂ map-prod post post ‘ set ps′
by simp
      }
      ultimately have ?X ps′ ⊂ ?X ps by (auto simp add: image-set simp del:
set-map)
    }
    ultimately show ?f (step-invariant st) < ?f st unfolding st step-invariant
      using psubset-card-mono[of ?X ps ?X ps′] by simp
  qed (auto simp add: invariant-start WF invariant)
  then show False using result by auto
qed

lemma closure-termination:
  assumes WF: wf n r wf n s
  and result: closure ([(init r, init s)], {(post (init r), post (init s))}) = None
  shows False
  using trans[OF closure-invariant-closure[of [([], init r, init s)], simplified] result]
  by (auto intro: closure-invariant-termination[OF WF])

lemma closure-invariant-complete:
  assumes eq: lang n r = lang n s
  and WF:  wf n r wf n s
  shows ∃ps N. closure-invariant ([([], init r, init s)], [], {(post (init r), post (init
s))}) =
    Some([], ps, N) (is ∃ - -. closure-invariant ([([], ?r, ?s)], -) = - is ∃ - -. ?cl = -)
proof (cases ?cl)
  case (Some st)
  moreover obtain ws ps N where ws-ps-N: st = (ws, ps, N) by (cases st) blast
  ultimately show ?thesis
  proof (cases ws)
```

case (*Cons wrs ws*)
with *Some* **obtain** *w* **where** *counterexample r s* = *Some w* **unfolding** *counterexample-def*
by (*cases wrs*) (*auto simp*: *ws-ps-N*)
with *eq counterexample-sound*[*OF - WF*] **show** *?thesis* **by** *blast*
qed *blast*
qed (*auto intro*: *closure-invariant-termination*[*OF WF*])

lemma *closure-complete*:
assumes *lang n r* = *lang n s wf n r wf n s*
shows ∃ *N*. *closure* ([(*init r*, *init s*)], {(*post* (*init r*), *post* (*init s*))}) = *Some* ([], *N*)
using *closure-invariant-complete*[*OF assms*]
by (*subst closure-invariant-closure*[*of* [([], *init r*, *init s*)], *simplified*, *symmetric*]) *auto*

lemma *check-eqv-complete*:
assumes *lang n r* = *lang n s wf n r wf n s*
shows *check-eqv r s*
using *closure-complete*[*OF assms*] **by** (*auto simp*: *check-eqv-def*)

lemma *counterexample-complete*:
assumes *lang n r* ≠ *lang n s* **and** *WF*: *wf n r wf n s*
shows ∃ *w*. *counterexample r s* = *Some w*
using *closure-invariant-sound*[*OF - WF*] *closure-invariant-termination*[*OF WF*] *assms*
by (*fastforce simp*: *counterexample-def Let-def split*: *option.splits list.splits*)

end

locale *rexp-DA-no-post* = *rexp-DA* σ *wf-atom project lookup init delta final wf-state id L n*
for σ :: *nat* ⇒ *'a list*
and *wf-atom* :: *nat* ⇒ *'b* :: *linorder* ⇒ *bool*
and *project* :: *'a* ⇒ *'a*
and *lookup* :: *'b* ⇒ *'a* ⇒ *bool*
and *init* :: *'b rexp* ⇒ *'s*
and *delta* :: *'a* ⇒ *'s* ⇒ *'s*
and *final* :: *'s* ⇒ *bool*
and *wf-state* :: *'s* ⇒ *bool*
and *L* :: *'s* ⇒ *'a lang*
and *n* :: *nat*
begin

lemma *step-efficient*[*code*]: *step* (*ws*, *N*) =
(*let*
(*r*, *s*) = *hd ws*;
new = *remdups* (*filter* (λ(*r*,*s*). (*r*,*s*) ∉ *N*) (*map* (λ*a*. (*delta a r*, *delta a s*)) (σ *n*)))

in (*tl ws* @ *new*, *set new* ∪ *N*))
by (*force simp*: *Let-def map-apfst-remdups' filter-map o-def split*: *prod.splits*)

end

locale *rexp-DFA-no-post* = *rexp-DFA* σ *wf-atom project lookup init delta final wf-state id L*
for σ :: *nat* ⇒ *'a list*
and *wf-atom* :: *nat* ⇒ *'b* :: *linorder* ⇒ *bool*
and *project* :: *'a* ⇒ *'a*
and *lookup* :: *'b* ⇒ *'a* ⇒ *bool*
and *init* :: *'b rexp* ⇒ *'s*
and *delta* :: *'a* ⇒ *'s* ⇒ *'s*
and *final* :: *'s* ⇒ *bool*
and *wf-state* :: *'s* ⇒ *bool*
and *L* :: *'s* ⇒ *'a lang*
begin

sublocale *rexp-DA-no-post* **by** *unfold-locales*

end

locale *rexp-DA-sim* = *project set o* σ *wf-atom project lookup*
for σ :: *nat* ⇒ *'a list*
and *wf-atom* :: *nat* ⇒ *'b* :: *linorder* ⇒ *bool*
and *project* :: *'a* ⇒ *'a*
and *lookup* :: *'b* ⇒ *'a* ⇒ *bool* +
fixes *init* :: *'b rexp* ⇒ *'s*
fixes *sim-delta* :: *'s* ⇒ *'s list*
fixes *final* :: *'s* ⇒ *bool*
fixes *wf-state* :: *'s* ⇒ *bool*
fixes *L* :: *'s* ⇒ *'a lang*
fixes *post* :: *'s* ⇒ *'s*
fixes *n* :: *nat*
assumes *L-init*[*simp*]: *wf n r* ⟹ *L* (*init r*) = *lang n r*
assumes *final-iff-Nil*[*simp*]: *final s* ⟷ [] ∈ *L s*
assumes *L-wf-state*[*dest*]: *wf-state s* ⟹ *L s* ⊆ *lists* (*set* (σ *n*))
assumes *init-wf-state*[*simp*]: *wf n r* ⟹ *wf-state* (*init r*)
assumes *L-post*[*simp*]: *wf-state s* ⟹ *L* (*post s*) = *L s*
assumes *wf-state-post*[*simp*]: *wf-state s* ⟹ *wf-state* (*post s*)
assumes *L-sim-delta*[*simp*]: *wf-state s* ⟹ *map L* (*sim-delta s*) = *map* (λ*a*. *lQuot a* (*L s*)) (σ *n*)
assumes *sim-delta-wf-state*[*simp*]: *wf-state s* ⟹ ∀ *s'* ∈ *set* (*sim-delta s*). *wf-state s'*
begin

definition *delta a s* = *sim-delta s* ! *index* (σ *n*) *a*

lemma *length-sim-delta*[*simp*]: *wf-state s* ⟹ *length* (*sim-delta s*) = *length* (σ *n*)

by (*metis L-sim-delta length-map*)

lemma *L-delta*[*simp*]: ⟦*a* ∈ *set* (*σ n*); *wf-state s*⟧ ⟹ *L* (*delta a s*) = *lQuot a* (*L s*)
using *L-sim-delta*[*of s*] **unfolding** *delta-def in-set-conv-nth*
by (*subst* (*asm*) *list-eq-iff-nth-eq*) *auto*

lemma *delta-wf-state*[*simp*]: ⟦*a* ∈ *set* (*σ n*); *wf-state s*⟧ ⟹ *wf-state* (*delta a s*)
unfolding *delta-def* **by** (*auto intro*: *bspec*[*OF sim-delta-wf-state nth-mem*])

sublocale *rexp-DA σ wf-atom project lookup init delta final wf-state post L*
by *unfold-locales auto*

sublocale *rexp-DA-sim-no-post*: *rexp-DA-no-post σ wf-atom project lookup init delta final wf-state L*
by *unfold-locales auto*

end

7 Initial Normalization of the Input

fun *toplevel-inters* **where**
toplevel-inters (*Inter r s*) = *toplevel-inters r* ∪ *toplevel-inters s*
| *toplevel-inters r* = {*r*}

lemma *toplevel-inters-nonempty*[*simp*]:
toplevel-inters r ≠ {}
by (*induct r*) *auto*

lemma *toplevel-inters-finite*[*simp*]:
finite (*toplevel-inters r*)
by (*induct r*) *auto*

context *alphabet*
begin

lemma *toplevel-inters-wf*:
wf n s = (∀ *r*∈*toplevel-inters s*. *wf n r*)
by (*induct s*) *auto*

end

context *project*
begin

lemma *toplevel-inters-lang*:

$r \in$ *toplevel-inters* $s \Longrightarrow$ *lang* $n\ s \subseteq$ *lang* $n\ r$
by (*induct* s) *auto*

lemma *toplevel-inters-lang-INT*:
lang $n\ s = (\bigcap r \in$ *toplevel-inters* $s.$ *lang* $n\ r)$
by (*induct* s) *auto*

lemma *toplevel-inters-in-lang*:
$w \in$ *lang* $n\ s = (\forall\, r \in$ *toplevel-inters* $s.\ w \in$ *lang* $n\ r)$
by (*induct* s) *auto*

lemma *lang-flatten-INTERSECT-finite*[*simp*]:
finite $X \Longrightarrow w \in$ *lang* n (*flatten INTERSECT* X) =
(*if* $X = \{\}$ *then* $w \in$ *lists* $(\Sigma\ n)$ *else* $(\forall\, r \in X.\ w \in$ *lang* $n\ r)$)
unfolding *lang-INTERSECT* **by** *auto*

end

fun *merge-distinct* **where**
merge-distinct [] $xs = xs$
| *merge-distinct* xs [] $= xs$
| *merge-distinct* $(a \# xs)\ (b \# ys) =$
(*if* $a = b$ *then merge-distinct* $xs\ (b \# ys)$
else if $a < b$ *then* $a \#$ *merge-distinct* $xs\ (b \# ys)$
else $b \#$ *merge-distinct* $(a \# xs)\ ys$)

lemma *set-merge-distinct*[*simp*]: *set* (*merge-distinct* $xs\ ys$) = *set* $xs \cup$ *set* ys
by (*induct* $xs\ ys$ *rule*: *merge-distinct.induct*) *auto*

lemma *sorted-merge-distinct*[*simp*]: ⟦*sorted* xs; *sorted* ys⟧ $\Longrightarrow$ *sorted* (*merge-distinct* $xs\ ys$)
by (*induct* $xs\ ys$ *rule*: *merge-distinct.induct*) (*auto*)

lemma *distinct-merge-distinct*[*simp*]: ⟦*sorted* xs; *distinct* xs; *sorted* ys; *distinct* ys⟧
$\Longrightarrow$
distinct (*merge-distinct* $xs\ ys$)
by (*induct* $xs\ ys$ *rule*: *merge-distinct.induct*) (*auto*)

lemma *sorted-list-of-set-merge-distinct*[*simp*]: ⟦*sorted* xs; *distinct* xs; *sorted* ys; *distinct* ys⟧ $\Longrightarrow$
merge-distinct $xs\ ys$ = *sorted-list-of-set* (*set* $xs \cup$ *set* ys)
by (*auto intro*: *sorted-distinct-set-unique*)

fun *zip-with-option* **where**
zip-with-option f (*Some* a) (*Some* b) = *Some* $(f\ a\ b)$
| *zip-with-option* - - - = *None*

lemma *zip-with-option-eq-Some*[*simp*]:
zip-with-option $f\ x\ y =$ *Some* $z \longleftrightarrow (\exists\, a\ b.\ z = f\ a\ b \wedge x =$ *Some* $a \wedge y =$ *Some*

```
b)
  by (induct f x y rule: zip-with-option.induct) auto

fun Pluss where
  Pluss (Plus r s) = zip-with-option merge-distinct (Pluss r) (Pluss s)
| Pluss Zero = Some []
| Pluss Full = None
| Pluss r = Some [r]

lemma Pluss-None[symmetric]: Pluss r = None ⟷ Full ∈ toplevel-summands r
  by (induct r) auto

lemma Pluss-Some: Pluss r = Some xs ⟷
  (Full ∉ set xs ∧ xs = sorted-list-of-set (toplevel-summands r − {Zero}))
proof (induct r arbitrary: xs)
  case (Plus r s)
  show ?case
  proof safe
    assume Pluss (Plus r s) = Some xs
      then obtain a b where ∗: Pluss r = Some a Pluss s = Some b xs =
merge-distinct a b by auto
    with Plus(1)[of a] Plus(2)[of b]
      show xs = sorted-list-of-set (toplevel-summands (Plus r s) − {Zero}) by
(simp add: Un-Diff)
    assume Full ∈ set xs with Plus(1)[of a] Plus(2)[of b] ∗ show False by (simp
add: Pluss-None)
  next
    assume Full ∉ set (sorted-list-of-set (toplevel-summands (Plus r s) − {Zero}))
    with Plus(1)[of sorted-list-of-set (toplevel-summands r − {Zero})]
      Plus(2)[of sorted-list-of-set (toplevel-summands s − {Zero})]
      show Pluss (Plus r s) = Some (sorted-list-of-set (toplevel-summands (Plus r
s) − {Zero}))
        by (simp add: Un-Diff)
  qed
qed force+

fun Inters where
  Inters (Inter r s) = zip-with-option merge-distinct (Inters r) (Inters s)
| Inters Zero = None
| Inters Full = Some []
| Inters r = Some [r]

lemma Inters-None[symmetric]: Inters r = None ⟷ Zero ∈ toplevel-inters r
  by (induct r) auto

lemma Inters-Some: Inters r = Some xs ⟷
  (Zero ∉ set xs ∧ xs = sorted-list-of-set (toplevel-inters r − {Full}))
proof (induct r arbitrary: xs)
  case (Inter r s)
```

show *?case*
proof *safe*
assume *Inters* (*Inter r s*) = *Some xs*
then obtain *a b* **where** ∗: *Inters r* = *Some a Inters s* = *Some b xs* = *merge-distinct a b* **by** *auto*
with *Inter*(*1*)[*of a*] *Inter*(*2*)[*of b*]
show *xs* = *sorted-list-of-set* (*toplevel-inters* (*Inter r s*) − {*Full*}) **by** (*simp add*: *Un-Diff*)
assume *Zero* ∈ *set xs* **with** *Inter*(*1*)[*of a*] *Inter*(*2*)[*of b*] ∗ **show** *False* **by** (*simp add*: *Inters-None*)
next
assume *Zero* ∉ *set* (*sorted-list-of-set* (*toplevel-inters* (*Inter r s*) − {*Full*}))
with *Inter*(*1*)[*of sorted-list-of-set* (*toplevel-inters r* − {*Full*})]
Inter(*2*)[*of sorted-list-of-set* (*toplevel-inters s* − {*Full*})]
show *Inters* (*Inter r s*) = *Some* (*sorted-list-of-set* (*toplevel-inters* (*Inter r s*) − {*Full*}))
by (*simp add*: *Un-Diff*)
qed
qed *force*+

definition *inPlus* **where**
inPlus r s = (*case Pluss* (*Plus r s*) *of None* ⇒ *Full* | *Some rs* ⇒ *PLUS rs*)

lemma *inPlus-alt*: *inPlus r s* = (*let X* = *toplevel-summands* (*Plus r s*) − {*Zero*} *in*
flatten PLUS (*if Full* ∈ *X then* {*Full*} *else X*))
proof (*cases Pluss r Pluss s rule*: *option.exhaust*[*case-product option.exhaust*])
case *Some-Some* **then show** *?thesis* **by** (*simp add*: *inPlus-def Pluss-None*) (*simp add*: *Pluss-Some Un-Diff*)
qed (*simp-all add*: *inPlus-def Pluss-None*)

fun *inTimes* **where**
inTimes Zero - = *Zero*
| *inTimes* - *Zero* = *Zero*
| *inTimes One r* = *r*
| *inTimes r One* = *r*
| *inTimes* (*Times r s*) *t* = *Times r* (*inTimes s t*)
| *inTimes r s* = *Times r s*

fun *inStar* **where**
inStar Zero = *One*
| *inStar Full* = *Full*
| *inStar One* = *One*
| *inStar* (*Star r*) = *Star r*
| *inStar r* = *Star r*

definition *inInter* **where**
inInter r s = (*case Inters* (*Inter r s*) *of None* ⇒ *Zero* | *Some rs* ⇒ *INTERSECT rs*)

lemma *inInter-alt*: *inInter r s* = (*let X* = *toplevel-inters* (*Inter r s*) − {*Full*} *in*
flatten INTERSECT (*if Zero* ∈ *X then* {*Zero*} *else X*))
proof (*cases Inters r Inters s rule*: *option.exhaust*[*case-product option.exhaust*])
case *Some-Some* **then show** *?thesis* **by** (*simp add*: *inInter-def Inters-None*)
(*simp add*: *Inters-Some Un-Diff*)
qed (*simp-all add*: *inInter-def Inters-None*)

fun *inNot* **where**
inNot Zero = *Full*
| *inNot Full* = *Zero*
| *inNot* (*Not r*) = *r*
| *inNot* (*Plus r s*) = *Inter* (*inNot r*) (*inNot s*)
| *inNot* (*Inter r s*) = *Plus* (*inNot r*) (*inNot s*)
| *inNot r* = *Not r*

fun *inPr* **where**
inPr Zero = *Zero*
| *inPr One* = *One*
| *inPr* (*Plus r s*) = *Plus* (*inPr r*) (*inPr s*)
| *inPr r* = *Pr r*

primrec *inorm* **where**
inorm Zero = *Zero*
| *inorm Full* = *Full*
| *inorm One* = *One*
| *inorm* (*Atom a*) = *Atom a*
| *inorm* (*Plus r s*) = *Plus* (*inorm r*) (*inorm s*)
| *inorm* (*Times r s*) = *Times* (*inorm r*) (*inorm s*)
| *inorm* (*Star r*) = *inStar* (*inorm r*)
| *inorm* (*Not r*) = *inNot* (*inorm r*)
| *inorm* (*Inter r s*) = *inInter* (*inorm r*) (*inorm s*)
| *inorm* (*Pr r*) = *inPr* (*inorm r*)

context *alphabet* **begin**

lemma *wf-inPlus*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *wf n* (*inPlus r s*)
by (*subst* (*asm*) (*1 2*) *toplevel-summands-wf*) (*auto simp*: *inPlus-alt*)

lemma *wf-inTimes*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *wf n* (*inTimes r s*)
by (*induct r s rule*: *inTimes.induct*) *auto*

lemma *wf-inStar*[*simp*]: *wf n r* ⟹ *wf n* (*inStar r*)
by (*induct r rule*: *inStar.induct*) *auto*

lemma *wf-inInter*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *wf n* (*inInter r s*)
by (*subst* (*asm*) (*1 2*) *toplevel-inters-wf*) (*auto simp*: *inInter-alt*)

lemma *wf-inNot*[*simp*]: *wf n r* ⟹ *wf n* (*inNot r*)

by (*induct r rule*: *inNot.induct*) *auto*

lemma *wf-inPr*[*simp*]: *wf* (*Suc n*) *r* ⟹ *wf n* (*inPr r*)
by (*induct r rule*: *inPr.induct*) *auto*

lemma *wf-inorm*[*simp*]: *wf n r* ⟹ *wf n* (*inorm r*)
by (*induct r arbitrary*: *n*) *auto*

end

context *project* **begin**

lemma *lang-inPlus*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*inPlus r s*) = *lang n* (*Plus r s*)
by (*auto 0 3 simp*: *inPlus-alt toplevel-summands-in-lang*[*of - n r*] *toplevel-summands-in-lang*[*of - n s*]
dest: *lang-subset-lists intro*: *bexI*[*of - Full*])

lemma *lang-inTimes*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*inTimes r s*) = *lang n* (*Times r s*)
by (*induct r s rule*: *inTimes.induct*) (*auto simp*: *conc-assoc*)

lemma *lang-inStar*[*simp*]: *wf n r* ⟹ *lang n* (*inStar r*) = *lang n* (*Star r*)
by (*induct r rule*: *inStar.induct*)
(*auto intro*: *star-if-lang dest*: *subsetD*[*OF star-subset-lists*, *rotated*])

lemma *Zero-toplevel-inters*[*dest*]: *Zero* ∈ *toplevel-inters r* ⟹ *lang n r* = {}
by (*metis lang.simps*(*1*) *subset-empty toplevel-inters-lang*)

lemma *toplevel-inters-Full*: ⟦*toplevel-inters r* = {*Full*}; *wf n r*⟧ ⟹ *lang n r* = *lists* (Σ *n*)
by (*metis antisym lang.simps*(*2*) *subsetI toplevel-inters.simps*(*3*) *toplevel-inters-in-lang*)

lemma *toplevel-inters-subset-singleton*[*simp*]: *toplevel-inters r* ⊆ {*s*} ⟷ *toplevel-inters r* = {*s*}
by (*metis subset-refl subset-singletonD toplevel-inters-nonempty*)

lemma *lang-inInter*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*inInter r s*) = *lang n* (*Inter r s*)
using *lang-subset-lists*[*of n*, *unfolded lang.simps*(*2*)[*symmetric*]]
toplevel-inters-nonempty[*of r*] *toplevel-inters-nonempty*[*of s*]
apply (*auto 0 2 simp*: *inInter-alt toplevel-inters-in-lang*[*of - n r*] *toplevel-inters-in-lang*[*of - n s*]
toplevel-inters-wf[*of n r*] *toplevel-inters-wf*[*of n s*] *Ball-def simp del*: *toplevel-inters-nonempty*
dest!: *toplevel-inters-Full*[*of - n*] *split*: *if-splits*)
by *fastforce*+

lemma *lang-inNot*[*simp*]: *wf n r* ⟹ *lang n* (*inNot r*) = *lang n* (*Not r*)
by (*induct r rule*: *inNot.induct*) (*auto dest*: *lang-subset-lists*)

lemma *lang-inPr*[*simp*]: *wf* (*Suc n*) *r* ⟹ *lang n* (*inPr r*) = *lang n* (*Pr r*)
by (*induct r rule*: *inPr.induct*) *auto*

lemma *lang-inorm*[*simp*]: *wf n r* ⟹ *lang n* (*inorm r*) = *lang n r*
by (*induct r arbitrary*: *n*) *auto*

end

8 Partial Derivatives-like Normalization

fun *pnPlus* :: *'a*::*linorder rexp* ⇒ *'a rexp* ⇒ *'a rexp* **where**
pnPlus Zero r = *r*
| *pnPlus r Zero* = *r*| *pnPlus* (*Plus r s*) *t* = *pnPlus r* (*pnPlus s t*)
| *pnPlus r* (*Plus s t*) =
(*if r* = *s then* (*Plus s t*)
else if r ≤ *s then Plus r* (*Plus s t*)
else Plus s (*pnPlus r t*))
| *pnPlus r s* =
(*if r* = *s then r*
else if r ≤ *s then Plus r s*
else Plus s r)

lemma (**in** *alphabet*) *wf-pnPlus*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *wf n* (*pnPlus r s*)
by (*induct r s rule*: *pnPlus.induct*) *auto*

lemma (**in** *project*) *lang-pnPlus*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*pnPlus r s*)
= *lang n* (*Plus r s*)
by (*induct r s rule*: *pnPlus.induct*) (*auto dest*!: *lang-subset-lists dest*: *project*
subsetD[*OF star-subset-lists*, *unfolded in-lists-conv-set*, *rotated* −*1*]
subsetD[*OF conc-subset-lists*, *unfolded in-lists-conv-set*, *rotated* −*1*])

fun *pnTimes* :: *'a*::*linorder rexp* ⇒ *'a rexp* ⇒ *'a rexp* **where**
pnTimes Zero r = *Zero*
| *pnTimes One r* = *r*
| *pnTimes* (*Plus r s*) *t* = *pnPlus* (*pnTimes r t*) (*pnTimes s t*)
| *pnTimes r s* = *Times r s*

lemma (**in** *alphabet*) *wf-pnTimes*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *wf n* (*pnTimes r s*)
by (*induct r s rule*: *pnTimes.induct*) *auto*

lemma (**in** *project*) *lang-pnTimes*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*pnTimes r*
s) = *lang n* (*Times r s*)
by (*induct r s rule*: *pnTimes.induct*) *auto*

fun *pnInter* :: *'a*::*linorder rexp* ⇒ *'a rexp* ⇒ *'a rexp* **where**
pnInter Zero r = *Zero*

```
| pnInter r Zero = Zero
| pnInter Full r = r
| pnInter r Full = r
| pnInter (Plus r s) t = pnPlus (pnInter r t) (pnInter s t)
| pnInter r (Plus s t) = pnPlus (pnInter r s) (pnInter r t)
| pnInter (Inter r s) t = pnInter r (pnInter s t)
| pnInter r (Inter s t) =
    (if r = s then Inter s t
    else if r ≤ s then Inter r (Inter s t)
    else Inter s (pnInter r t))
| pnInter r s =
    (if r = s then s
     else if r ≤ s then Inter r s
     else Inter s r)

lemma (in alphabet) wf-pnInter[simp]: ⟦wf n r; wf n s⟧ ⟹ wf n (pnInter r s)
  by (induct r s rule: pnInter.induct) auto

lemma (in project) lang-pnInter[simp]: ⟦wf n r; wf n s⟧ ⟹ lang n (pnInter r s)
= lang n (Inter r s)
   by (induct r s rule: pnInter.induct) (auto, auto dest!: lang-subset-lists dest:
project
      subsetD[OF star-subset-lists, unfolded in-lists-conv-set, rotated −1]
      subsetD[OF conc-subset-lists, unfolded in-lists-conv-set, rotated −1])

fun pnNot :: 'a::linorder rexp ⇒ 'a rexp where
  pnNot (Plus r s) = pnInter (pnNot r) (pnNot s)
| pnNot (Inter r s) = pnPlus (pnNot r) (pnNot s)
| pnNot Full = Zero
| pnNot Zero = Full
| pnNot (Not r) = r
| pnNot r = Not r

lemma (in alphabet) wf-pnNot[simp]: wf n r ⟹ wf n (pnNot r)
  by (induct r rule: pnNot.induct) auto

lemma (in project) lang-pnNot[simp]: wf n r ⟹ lang n (pnNot r) = lang n (Not
r)
  by (induct r rule: pnNot.induct) (auto dest: lang-subset-lists)

fun pnPr :: 'a::linorder rexp ⇒ 'a rexp where
  pnPr Zero = Zero
| pnPr One = One
| pnPr (Plus r s) = pnPlus (pnPr r) (pnPr s)
| pnPr r = Pr r

lemma (in alphabet) wf-pnPr[simp]: wf (Suc n) r ⟹ wf n (pnPr r)
  by (induct r rule: pnPr.induct) auto
```

lemma (**in** *project*) *lang-pnPr*[*simp*]: *wf* (*Suc n*) *r* ⟹ *lang n* (*pnPr r*) = *lang n* (*Pr r*)
by (*induct r rule*: *pnPr.induct*) *auto*

primrec *pnorm* :: *'a*::*linorder rexp* ⇒ *'a rexp* **where**
pnorm Zero = *Zero*
| *pnorm Full* = *Full*
| *pnorm One* = *One*
| *pnorm* (*Atom a*) = *Atom a*
| *pnorm* (*Plus r s*) = *pnPlus* (*pnorm r*) (*pnorm s*)
| *pnorm* (*Times r s*) = *pnTimes* (*pnorm r*) *s*
| *pnorm* (*Star r*) = *Star r*
| *pnorm* (*Inter r s*) = *pnInter* (*pnorm r*) (*pnorm s*)
| *pnorm* (*Not r*) = *pnNot* (*pnorm r*)
| *pnorm* (*Pr r*) = *pnPr* (*pnorm r*)

lemma (**in** *alphabet*) *wf-pnorm*[*simp*]: *wf n r* ⟹ *wf n* (*pnorm r*)
by (*induct r arbitrary*: *n*) *auto*

lemma (**in** *project*) *lang-pnorm*[*simp*]: *wf n r* ⟹ *lang n* (*pnorm r*) = *lang n r*
by (*induct r arbitrary*: *n*) *auto*

9 Monadic Second-Order Logic Formulas

9.1 Interpretations and Encodings

type-synonym *'a interp* = *'a list* × (*nat* + *nat set*) *list*

abbreviation *enc-atom-bool I n* ≡ *map* (λ*x*. *case x of Inl p* ⇒ *n* = *p* | *Inr P* ⇒ *n* ∈ *P*) *I*

abbreviation *enc-atom I n a* ≡ (*a*, *enc-atom-bool I n*)

9.2 Syntax and Semantics of MSO

datatype *'a formula* =
FQ 'a nat
| *FLess nat nat*
| *FIn nat nat*
| *FNot 'a formula*
| *FOr 'a formula 'a formula*
| *FAnd 'a formula 'a formula*
| *FExists 'a formula*
| *FEXISTS 'a formula*

primrec *FOV* :: *'a formula* ⇒ *nat set* **where**
FOV (*FQ a m*) = {*m*}
| *FOV* (*FLess m1 m2*) = {*m1*, *m2*}

| *FOV* (*FIn m M*) = {*m*}
| *FOV* (*FNot* φ) = *FOV* φ
| *FOV* (*FOr* φ_1 φ_2) = *FOV* φ_1 ∪ *FOV* φ_2
| *FOV* (*FAnd* φ_1 φ_2) = *FOV* φ_1 ∪ *FOV* φ_2
| *FOV* (*FExists* φ) = (λ*x*. *x* − *1*) ‘ (*FOV* φ − {*0*})
| *FOV* (*FEXISTS* φ) = (λ*x*. *x* − *1*) ‘ *FOV* φ

primrec *SOV* :: *'a formula* ⇒ *nat set* **where**
  SOV (*FQ a m*) = {}
| *SOV* (*FLess m1 m2*) = {}
| *SOV* (*FIn m M*) = {*M*}
| *SOV* (*FNot* φ) = *SOV* φ
| *SOV* (*FOr* φ_1 φ_2) = *SOV* φ_1 ∪ *SOV* φ_2
| *SOV* (*FAnd* φ_1 φ_2) = *SOV* φ_1 ∪ *SOV* φ_2
| *SOV* (*FExists* φ) = (λ*x*. *x* − *1*) ‘ *SOV* φ
| *SOV* (*FEXISTS* φ) = (λ*x*. *x* − *1*) ‘ (*SOV* φ − {*0*})

definition σ = (λΣ *n*. *concat* (*map* (λ*bs*. *map* (λ*a*. (*a*, *bs*)) Σ) (*List.n-lists n* [*True*, *False*])))
definition π = (λ(*a*, *bs*). (*a*, *tl bs*))
definition ε = (λΣ (*a*::*'a*, *bs*). *if a* ∈ *set* Σ *then* [(*a*, *True* # *bs*), (*a*, *False* # *bs*)] *else* [])

datatype *'a atom* =
    Singleton 'a bool list
  | *AQ nat 'a*
  | *Arbitrary-Except nat bool*
  | *Arbitrary-Except2 nat nat*
derive *linorder atom*

fun *wf-atom* **where**
  wf-atom Σ *n* (*Singleton a bs*) = (*a* ∈ *set* Σ ∧ *length bs* = *n*)
| *wf-atom* Σ *n* (*AQ m a*) = (*a* ∈ *set* Σ ∧ *m* < *n*)
| *wf-atom* Σ *n* (*Arbitrary-Except m* -) = (*m* < *n*)
| *wf-atom* Σ *n* (*Arbitrary-Except2 m1 m2*) = (*m1* < *n* ∧ *m2* < *n*)

fun *lookup* **where**
  lookup (*Singleton a' bs'*) (*a*, *bs*) = (*a* = *a'* ∧ *bs* = *bs'*)
| *lookup* (*AQ m a'*) (*a*, *bs*) = (*a* = *a'* ∧ *bs* ! *m*)
| *lookup* (*Arbitrary-Except m b*) (-, *bs*) = (*bs* ! *m* = *b*)
| *lookup* (*Arbitrary-Except2 m1 m2*) (-, *bs*) = (*bs* ! *m1* ∧ *bs* ! *m2*)

lemma π-σ: π ‘ (*set* ∘ σ Σ) (*n* + *1*) = (*set* ∘ σ Σ) *n*
  unfolding π-*def* σ-*def set-map*[*symmetric*] *o-apply map-concat* **by** *auto*

locale *formula* = *embed2 set* ∘ (σ Σ) *wf-atom* Σ π *lookup* ε Σ *case-prod Singleton*
for Σ :: *'a* :: *linorder list* +
assumes *nonempty*: Σ ≠ []
begin

abbreviation Σ-*product-lists* $n \equiv$
$List.maps\ (\lambda bools.\ map\ (\lambda a.\ (a,\ bools))\ \Sigma)\ (bool\text{-}product\text{-}lists\ n)$

primrec *pre-wf-formula* :: $nat \Rightarrow 'a\ formula \Rightarrow bool$ **where**
$pre\text{-}wf\text{-}formula\ n\ (FQ\ a\ m) = (a \in set\ \Sigma \wedge m < n)$
$|\ pre\text{-}wf\text{-}formula\ n\ (FLess\ m1\ m2) = (m1 < n \wedge m2 < n)$
$|\ pre\text{-}wf\text{-}formula\ n\ (FIn\ m\ M) = (m < n \wedge M < n)$
$|\ pre\text{-}wf\text{-}formula\ n\ (FNot\ \varphi) = pre\text{-}wf\text{-}formula\ n\ \varphi$
$|\ pre\text{-}wf\text{-}formula\ n\ (FOr\ \varphi_1\ \varphi_2) = (pre\text{-}wf\text{-}formula\ n\ \varphi_1 \wedge pre\text{-}wf\text{-}formula\ n\ \varphi_2)$
$|\ pre\text{-}wf\text{-}formula\ n\ (FAnd\ \varphi_1\ \varphi_2) = (pre\text{-}wf\text{-}formula\ n\ \varphi_1 \wedge pre\text{-}wf\text{-}formula\ n\ \varphi_2)$
$|\ pre\text{-}wf\text{-}formula\ n\ (FExists\ \varphi) = (pre\text{-}wf\text{-}formula\ (n + 1)\ \varphi \wedge 0 \in FOV\ \varphi \wedge 0 \notin SOV\ \varphi)$
$|\ pre\text{-}wf\text{-}formula\ n\ (FEXISTS\ \varphi) = (pre\text{-}wf\text{-}formula\ (n + 1)\ \varphi \wedge 0 \notin FOV\ \varphi \wedge 0 \in SOV\ \varphi)$

abbreviation $closed \equiv pre\text{-}wf\text{-}formula\ 0$

definition [*simp*]: $wf\text{-}formula\ n\ \varphi \equiv pre\text{-}wf\text{-}formula\ n\ \varphi \wedge FOV\ \varphi \cap SOV\ \varphi = \{\}$

lemma *max-idx-vars*: $pre\text{-}wf\text{-}formula\ n\ \varphi \Longrightarrow \forall\, p \in FOV\ \varphi \cup SOV\ \varphi.\ p < n$
by (*induct* φ *arbitrary*: n)
(*auto split*: *if-split-asm*, (*metis Un-iff diff-Suc-less less-SucE less-imp-diff-less*)+)

lemma *finite-FOV*: *finite* $(FOV\ \varphi)$
by (*induct* φ) (*auto split*: *if-split-asm*)

9.3 ENC

definition *valid-ENC* :: $nat \Rightarrow nat \Rightarrow ('a\ atom)\ rexp$ **where**
$valid\text{-}ENC\ n\ p = (if\ n = 0\ then\ Full\ else$
$TIMES\ [$
$Star\ (Atom\ (Arbitrary\text{-}Except\ p\ False)),$
$Atom\ (Arbitrary\text{-}Except\ p\ True),$
$Star\ (Atom\ (Arbitrary\text{-}Except\ p\ False))])$

lemma *wf-rexp-valid-ENC*: $n = 0 \vee p < n \Longrightarrow wf\ n\ (valid\text{-}ENC\ n\ p)$
unfolding *valid-ENC-def* **by** *auto*

definition *ENC* :: $nat \Rightarrow nat\ set \Rightarrow ('a\ atom)\ rexp$ **where**
$ENC\ n\ V = flatten\ INTERSECT\ (valid\text{-}ENC\ n\ `\ V)$

lemma *wf-rexp-ENC*: $[\![finite\ V;\ n = 0 \vee (\forall\, v \in V.\ v < n)]\!] \Longrightarrow wf\ n\ (ENC\ n\ V)$
unfolding *ENC-def*
by (*intro iffD2*[*OF wf-flatten-INTERSECT*]) (*auto simp*: *wf-rexp-valid-ENC*)

lemma *enc-atom-σ-eq*: $i < length\ w \Longrightarrow$

(*length I = n ∧ p ∈ set Σ*) ⟷ *enc-atom I i p ∈ set* (*σ Σ n*)
by (*auto simp*: *σ-def set-n-lists intro*!: *exI*[*of - enc-atom-bool I i*] *imageI*)

lemmas *enc-atom-σ* = *iffD1*[*OF enc-atom-σ-eq*, *OF - conjI*]

lemma *enc-atom-bool-take-drop-True*:
⟦*r* < *length I*; *case I* ! *r of Inl p′* ⇒ *p* = *p′* | *Inr P* ⇒ *p* ∈ *P*⟧ ⟹
enc-atom-bool I p = *take r* (*enc-atom-bool I p*) @ *True* # *drop* (*Suc r*)
(*enc-atom-bool I p*)
by (*auto intro*: *trans*[*OF id-take-nth-drop*])

lemma *enc-atom-bool-take-drop-True2*:
⟦*r* < *length I*; *case I* ! *r of Inl p′* ⇒ *p* = *p′* | *Inr P* ⇒ *p* ∈ *P*;
s < *length I*; *case I* ! *s of Inl p′* ⇒ *p* = *p′* | *Inr P* ⇒ *p* ∈ *P*; *r* < *s*⟧ ⟹
enc-atom-bool I p = *take r* (*enc-atom-bool I p*) @ *True* #
take (*s* − *Suc r*) (*drop* (*Suc r*) (*enc-atom-bool I p*)) @ *True* #
drop (*Suc s*) (*enc-atom-bool I p*)
using *id-take-nth-drop*[*of r enc-atom-bool I p*]
id-take-nth-drop[*of s* − *r* − *1 drop* (*Suc r*) (*enc-atom-bool I p*)]
by *auto*

lemma *enc-atom-bool-take-drop-False*:
⟦*r* < *length I*; *case I* ! *r of Inl p′* ⇒ *p* ≠ *p′* | *Inr P* ⇒ *p* ∉ *P*⟧ ⟹
enc-atom-bool I p = *take r* (*enc-atom-bool I p*) @ *False* # *drop* (*Suc r*)
(*enc-atom-bool I p*)
by (*auto intro*: *trans*[*OF id-take-nth-drop*] *split*: *sum.splits*)

lemma *enc-atom-lang-AQ*: ⟦*r* < *length I*;
case I ! *r of Inl p′* ⇒ *p* = *p′* | *Inr P* ⇒ *p* ∈ *P*; *length I* = *n*; *a* ∈ *set Σ*⟧ ⟹
[*enc-atom I p a*] ∈ *lang n* (*Atom* (*AQ r a*))
unfolding *lang.simps*
by (*force intro*!: *enc-atom-bool-take-drop-True*
image-eqI[*of - -* (λ*J*. *take r J* @ *drop* (*r* + *1*) *J*) (*enc-atom-bool I p*)]
simp: *σ-def set-n-lists*)

lemma *enc-atom-lang-Arbitrary-Except-True*: ⟦*r* < *length I*;
case I ! *r of Inl p′* ⇒ *p* = *p′* | *Inr P* ⇒ *p* ∈ *P*; *length I* = *n*; *a* ∈ *set Σ*⟧ ⟹
[*enc-atom I p a*] ∈ *lang n* (*Atom* (*Arbitrary-Except r True*))
unfolding *lang.simps*
by (*force intro*!: *enc-atom-bool-take-drop-True*
image-eqI[*of - -* (λ*J*. *take r J* @ *drop* (*r* + *1*) *J*) (*enc-atom-bool I p*)]
simp: *σ-def set-n-lists*)

lemma *enc-atom-lang-Arbitrary-Except2*:⟦*r* < *length I*; *s* < *length I*;
case I ! *r of Inl p′* ⇒ *p* = *p′* | *Inr P* ⇒ *p* ∈ *P*;
case I ! *s of Inl p′* ⇒ *p* = *p′* | *Inr P* ⇒ *p* ∈ *P*; *length I* = *n*; *a* ∈ *set Σ*⟧ ⟹
[*enc-atom I p a*] ∈ *lang n* (*Atom* (*Arbitrary-Except2 r s*))
unfolding *lang.simps*
by (*force intro*!: *enc-atom-bool-take-drop-True2*

simp: *set-n-lists* σ*-def take-Cons' drop-Cons' numeral-eq-Suc*)

lemma *enc-atom-lang-Arbitrary-Except-False*: ⟦$r <$ *length* I;
case I ! r *of Inl* $p' \Rightarrow p \neq p' \mid$ *Inr* $P \Rightarrow p \notin P$; *length* $I = n$; $a \in$ *set* Σ⟧ ⟹
[*enc-atom* I p a] ∈ *lang* n (*Atom* (*Arbitrary-Except* r *False*))
unfolding *lang.simps*
by (*force intro*!: *enc-atom-bool-take-drop-False*
image-eqI[*of* - - (λJ. *take* r J @ *drop* (r + *1*) J) (*enc-atom-bool* I p)]
simp: *set-n-lists* σ*-def split*: *sum.splits*)

lemma *AQ-D*:
assumes $v \in$ *lang* n (*Atom* (*AQ* m a)) $m < n$ $a \in$ *set* Σ
shows ∃x. v = [x] ∧ *fst* x = a ∧ *snd* x ! m
using *assms* **by** *auto*

lemma *Arbitrary-ExceptD*:
assumes $v \in$ *lang* n (*Atom* (*Arbitrary-Except* r b)) $r < n$
shows ∃x. v = [x] ∧ *snd* x ! r = b
using *assms* **by** *auto*

lemma *Arbitrary-Except2D*:
assumes $v \in$ *lang* n (*Atom* (*Arbitrary-Except2* r s)) $r < n$ $s < n$
shows ∃x. v = [x] ∧ *snd* x ! r ∧ *snd* x ! s
using *assms* **by** *auto*

lemma *star-Arbitrary-ExceptD*:
⟦$v \in$ *star* (*lang* n (*Atom* (*Arbitrary-Except* r b))); $r < n$; $i <$ *length* v⟧ ⟹
snd (v ! i) ! r = b
proof (*induct arbitrary*: i *rule*: *star-induct*)
case (*append* u v) **thus** *?case* **by** (*cases* i) (*auto dest*: *Arbitrary-ExceptD*)
qed *simp*

end

end

10 M2L

10.1 Encodings

context *formula*
begin

fun *enc* :: 'a *interp* ⇒ ('a × *bool list*) *list* **where**
enc (w, I) = *map-index* (*enc-atom* I) w

abbreviation *wf-interp* w I ≡ (*length* $w > 0$ ∧
(∀ $a \in$ *set* w. $a \in$ *set* Σ) ∧
(∀ $x \in$ *set* I. *case* x *of Inl* $p \Rightarrow p <$ *length* $w \mid$ *Inr* $P \Rightarrow \forall\, p \in P.\ p <$ *length* w))

fun *wf-interp-for-formula* :: *'a interp* ⇒ *'a formula* ⇒ *bool* **where**
wf-interp-for-formula (*w*, *I*) *φ* =
(*wf-interp w I* ∧
(∀ *n* ∈ *FOV φ*. *case I* ! *n of Inl* - ⇒ *True* | - ⇒ *False*) ∧
(∀ *n* ∈ *SOV φ*. *case I* ! *n of Inl* - ⇒ *False* | - ⇒ *True*))

fun *satisfies* :: *'a interp* ⇒ *'a formula* ⇒ *bool* (**infix** ⊨ *50*) **where**
(*w*, *I*) ⊨ *FQ a m* = (*w* ! (*case I* ! *m of Inl p* ⇒ *p*) = *a*)
| (*w*, *I*) ⊨ *FLess m1 m2* = ((*case I* ! *m1 of Inl p* ⇒ *p*) < (*case I* ! *m2 of Inl p* ⇒ *p*))
| (*w*, *I*) ⊨ *FIn m M* = ((*case I* ! *m of Inl p* ⇒ *p*) ∈ (*case I* ! *M of Inr P* ⇒ *P*))
| (*w*, *I*) ⊨ *FNot φ* = (¬ (*w*, *I*) ⊨ *φ*)
| (*w*, *I*) ⊨ (*FOr φ*$_1$ *φ*$_2$) = ((*w*, *I*) ⊨ *φ*$_1$ ∨ (*w*, *I*) ⊨ *φ*$_2$)
| (*w*, *I*) ⊨ (*FAnd φ*$_1$ *φ*$_2$) = ((*w*, *I*) ⊨ *φ*$_1$ ∧ (*w*, *I*) ⊨ *φ*$_2$)
| (*w*, *I*) ⊨ (*FExists φ*) = (∃ *p*. *p* ∈ {*0* .. *length w* − *1*} ∧ (*w*, *Inl p* # *I*) ⊨ *φ*)
| (*w*, *I*) ⊨ (*FEXISTS φ*) = (∃ *P*. *P* ⊆ {*0* .. *length w* − *1*} ∧ (*w*, *Inr P* # *I*) ⊨ *φ*)

definition *lang*$_{M2L}$ :: *nat* ⇒ *'a formula* ⇒ (*'a* × *bool list*) *list set* **where**
lang$_{M2L}$ *n φ* = {*enc* (*w*, *I*) | *w I*.
length I = *n* ∧ *wf-interp-for-formula* (*w*, *I*) *φ* ∧ *satisfies* (*w*, *I*) *φ*}

definition *dec-word* ≡ *map fst*

definition *positions-in-row w i* =
Option.these (*set* (*map-index* (λ*p a-bs*. *if nth* (*snd a-bs*) *i then Some p else None*) *w*))

definition *dec-interp n FO* (*w* :: (*'a* × *bool list*) *list*) ≡ *map* (λ*i*.
if i ∈ *FO*
then Inl (*the-elem* (*positions-in-row w i*))
else Inr (*positions-in-row w i*)) [*0*..<*n*]

lemma *positions-in-row*: *positions-in-row w i* = {*p*. *p* < *length w* ∧ *snd* (*w* ! *p*) ! *i*}
unfolding *positions-in-row-def Option.these-def* **by** (*auto intro*!: *image-eqI*[*of* - *the*])

lemma *positions-in-row-unique*: ∃!*p*. *p* < *length w* ∧ *snd* (*w* ! *p*) ! *i* ⟹
the-elem (*positions-in-row w i*) = (*THE p*. *p* < *length w* ∧ *snd* (*w* ! *p*) ! *i*)
by (*rule the1I2*) (*auto simp*: *the-elem-def positions-in-row*)

lemma *positions-in-row-length*: ∃!*p*. *p* < *length w* ∧ *snd* (*w* ! *p*) ! *i* ⟹
the-elem (*positions-in-row w i*) < *length w*
unfolding *positions-in-row-unique* **by** (*rule the1I2*) *auto*

lemma *dec-interp-Inl*: ⟦*i* ∈ *FO*; *i* < *n*⟧ ⟹ ∃ *p*. *dec-interp n FO x* ! *i* = *Inl p*
unfolding *dec-interp-def* **using** *nth-map*[*of n* [*0*..<*n*]] **by** *auto*

lemma *dec-interp-not-Inr*: ⟦*dec-interp n FO x ! i = Inr P*; *i ∈ FO*; *i < n*⟧ ⟹ *False*
unfolding *dec-interp-def* **using** *nth-map*[*of n* [*0..<n*]] **by** *auto*

lemma *dec-interp-Inr*: ⟦*i ∉ FO*; *i < n*⟧ ⟹ ∃ *P*. *dec-interp n FO x ! i = Inr P*
unfolding *dec-interp-def* **using** *nth-map*[*of n* [*0..<n*]] **by** *auto*

lemma *dec-interp-not-Inl*: ⟦*dec-interp n FO x ! i = Inl p*; *i ∉ FO*; *i < n*⟧ ⟹ *False*
unfolding *dec-interp-def* **using** *nth-map*[*of n* [*0..<n*]] **by** *auto*

lemma *Inl-dec-interp-length*:
assumes ∀ *i ∈ FO*. ∃!*p*. *p < length w ∧ snd* (*w ! p*) *! i*
shows *Inl p ∈ set* (*dec-interp n FO w*) ⟹ *p < length w*
unfolding *dec-interp-def* **by** (*auto intro*: *positions-in-row-length*[*OF bspec*[*OF assms*]])

lemma *Inr-dec-interp-length*: ⟦*Inr P ∈ set* (*dec-interp n FO w*); *p ∈ P*⟧ ⟹ *p < length w*
unfolding *dec-interp-def* **by** (*auto simp*: *positions-in-row*)

lemma *the-elem-Collect*[*simp*]:
assumes ∃!*x*. *P x*
shows *the-elem* (*Collect P*) = (*The P*)
proof (*unfold the-elem-def*, *rule the-equality*)
from *assms* **have** *P* (*The P*) **by** (*rule theI'*)
with *assms* **show** *Collect P* = {*The P*} **by** *auto*

fix *x* **assume** *Collect P* = {*x*}
then show *x* = *The P* **by** (*auto intro*: *the-equality*[*symmetric*])
qed

lemma *enc-atom-dec*:
⟦*wf-word n w*; ∀ *i ∈ FO*. *i < n* ⟶ (∃!*p*. *p < length w ∧ snd* (*w ! p*) *! i*); *p < length w*⟧ ⟹
enc-atom (*dec-interp n FO w*) *p* (*fst* (*w ! p*)) = *w ! p*
unfolding *wf-word dec-interp-def map-filter-def Let-def*
by (*auto 0 4 simp*: *comp-def σ-def set-n-lists positions-in-row dest*: *nth-mem*[*of p w*]
intro!: *trans*[*OF iffD2*[*OF prod.inject*] *prod.collapse*] *nth-equalityI the-equality*[*symmetric*]
intro: *the1I2*[*of* λ*p*. *p < length w ∧ snd* (*w ! p*) *! i* λ*p*. *snd* (*w ! p*) *! i* **for** *i*]
elim!: *contrapos-np*[*of - False*])

lemma *enc-dec*:
⟦*wf-word n w*; ∀ *i ∈ FO*. *i < n* ⟶ (∃!*p*. *p < length w ∧ snd* (*w ! p*) *! i*)⟧ ⟹
enc (*dec-word w*, *dec-interp n FO w*) = *w*
unfolding *enc.simps dec-word-def*

by (*intro trans*[*OF map-index-map map-index-congL*] *allI impI enc-atom-dec*) *assumption*+

lemma *dec-word-enc*: *dec-word* (*enc* (*w*, *I*)) = *w*
unfolding *dec-word-def* **by** *auto*

lemma *enc-unique*:
assumes *wf-interp w I i* < *length I*
shows ∃ *p*. *I* ! *i* = *Inl p* ⟹ ∃!*p*. *p* < *length* (*enc* (*w*, *I*)) ∧ *snd* (*enc* (*w*, *I*) ! *p*) ! *i*
proof (*erule exE*)
fix *p* **assume** *I* ! *i* = *Inl p*
with *assms* **show** *?thesis* **by** (*auto simp*: *map-index all-set-conv-all-nth intro*!: *exI*[*of* - *p*])
qed

lemma *dec-interp-enc-Inl*:
⟦*dec-interp n FO* (*enc* (*w*, *I*)) ! *i* = *Inl p*′; *I* ! *i* = *Inl p*; *i* ∈ *FO*; *i* < *n*; *length I* = *n*; *p* < *length w*; *wf-interp w I*⟧ ⟹
p = *p*′
unfolding *dec-interp-def* **using** *nth-map*[*of* - [*0*..<*n*]] *positions-in-row-unique*[*OF enc-unique*]
by (*auto intro*: *sym*[*OF the-equality*])

lemma *dec-interp-enc-Inr*:
⟦*dec-interp n FO* (*enc* (*w*, *I*)) ! *i* = *Inr P*′; *I* ! *i* = *Inr P*; *i* ∉ *FO*; *i* < *n*; *length I* = *n*; ∀ *p* ∈ *P*. *p* < *length w*⟧ ⟹
P = *P*′
unfolding *dec-interp-def positions-in-row* **by** *auto*

lemma *length-dec-interp*[*simp*]: *length* (*dec-interp n FO x*) = *n*
by (*auto simp*: *dec-interp-def*)

lemma *nth-dec-interp*[*simp*]: *i* < *n* ⟹ *dec-interp n* {} *x* ! *i* = *Inr* (*positions-in-row x i*)
by (*auto simp*: *dec-interp-def*)

lemma *set-σD*[*simp*]: (*a*, *bs*) ∈ *set* (*σ* Σ *n*) ⟹ *a* ∈ *set* Σ
unfolding *σ-def* **by** *auto*

lemma *lang-ENC*:
assumes *FO* ⊆ {*0* ..< *n*} *SO* ⊆ {*0* ..< *n*} − *FO*
shows *lang n* (*ENC n FO*) − {[]} = {*enc* (*w*, *I*) | *w I* . *length I* = *n* ∧ *wf-interp w I* ∧
(∀ *i* ∈ *FO*. *case I* ! *i of Inl* - ⇒ *True* | *Inr* - ⇒ *False*) ∧
(∀ *i* ∈ *SO*. *case I* ! *i of Inl* - ⇒ *False* | *Inr* - ⇒ *True*)}
(**is** *?L* = *?R*)
proof (*cases FO* = {})
case *True* **with** *assms* **show** *?thesis*

```
    by (force simp: ENC-def dec-word-def wf-word
      in-set-conv-nth[of - dec-interp n {} x for x] positions-in-row Ball-def
      intro!: enc-atom-σ exI[of - dec-word x :: 'a list for x] exI[of - dec-interp n {}
x for x]
      enc-dec[of n - {}, symmetric, unfolded dec-word-def enc.simps map-index-map])
next
  case False
  hence nonempty: valid-ENC n ‘ FO ≠ {} by simp
  have finite: finite (valid-ENC n ‘ FO)
    by (intro finite-imageI[OF finite-subset[OF assms(1)]]) auto
  from False assms(1) have 0 < n by auto
  with wf-rexp-valid-ENC assms(1) have wf-rexp: ∀ x ∈ valid-ENC n ‘ FO. wf n
x by auto
  { fix r w I assume r ∈ FO and ∗: length I = n wf-interp w I
      (∀ i ∈ FO. case I ! i of Inl - ⇒ True | Inr - ⇒ False)
      (∀ i ∈ SO. case I ! i of Inl - ⇒ False | Inr - ⇒ True)
    then obtain p where p: I ! r = Inl p by (cases I ! r) auto
    moreover from ‹r ∈ FO› assms(1) ∗(1) have r < length I by auto
    ultimately have Inl p ∈ set I by (auto simp add: in-set-conv-nth)
    with ∗(2) have p < length w by auto
    with ∗(2) obtain a where a: w ! p = a a ∈ set Σ by auto
    from ‹r < length I› p ∗(1) ‹a ∈ set Σ›
      have [enc-atom I p a] ∈ lang n (Atom (Arbitrary-Except r True))
      by (intro enc-atom-lang-Arbitrary-Except-True) auto
    moreover
    from ‹r < length I› p ∗(1) ‹a ∈ set Σ› ∗(2) ‹p < length w›
      have take p (enc (w, I)) ∈ star (lang n (Atom (Arbitrary-Except r False)))
      by (auto simp: in-set-conv-nth simp del: lang.simps
        intro!: Ball-starI enc-atom-lang-Arbitrary-Except-False) auto
    moreover
    from ‹r < length I› p ∗(1) ‹a ∈ set Σ› ∗(2) ‹p < length w›
      have drop (Suc p) (enc (w, I)) ∈ star (lang n (Atom (Arbitrary-Except r
False)))
      by (auto simp: in-set-conv-nth simp del: lang.simps
        intro!: Ball-starI enc-atom-lang-Arbitrary-Except-False) auto
    ultimately have take p (enc (w, I)) @ [enc-atom I p a] @ drop (p + 1) (enc
(w, I)) ∈
      lang n (valid-ENC n r) using ‹0 < n› unfolding valid-ENC-def by (auto
simp del: append.simps)
    with ‹p < length w› a have enc (w, I) ∈ lang n (valid-ENC n r)
      using id-take-nth-drop[of p enc (w, I)] by auto
  }
  hence ?R ⊆ ?L using lang-flatten-INTERSECT[OF finite nonempty wf-rexp]
by (auto simp add: ENC-def)
  moreover
  { fix x assume x ∈ (⋂ r ∈ valid-ENC n ‘ FO. lang n r)
    hence r: ∀ r ∈ FO. x ∈ lang n (valid-ENC n r) by blast
    have length (dec-interp n FO x) = n unfolding dec-interp-def by simp
    moreover
```

```
  { fix r assume r ∈ FO
    with assms have r < n by auto
    from ‹r ∈ FO› r obtain u v w where uvw: x = u @ v @ w
      u ∈ star (lang n (Atom (Arbitrary-Except r False)))
      v ∈ lang n (Atom (Arbitrary-Except r True))
    w ∈ star (lang n (Atom (Arbitrary-Except r False))) using ‹0 < n› unfolding
valid-ENC-def
      by (fastforce simp del: lang.simps(4))
   hence length u < length x ⋀i. i < length x ⟶ snd (x ! i) ! r ⟷ i = length
u
      by (auto simp: nth-append nth-Cons' split: if-split-asm simp del: lang.simps
          dest!: Arbitrary-ExceptD[OF - ‹r < n›]
          dest: star-Arbitrary-ExceptD[OF - ‹r < n›, of u]
          elim!: iffD1[OF star-Arbitrary-ExceptD[OF - ‹r < n›, of w False]]) auto
   hence ∃!p. p < length x ∧ snd (x ! p) ! r by auto
  } note * = this
   have x-wf-word: wf-word n x using wf-lang-wf-word[OF wf-rexp-valid-ENC]
False r assms(1)
   by auto
   with * have x = enc (dec-word x, dec-interp n FO x) by (intro sym[OF
enc-dec]) auto
  moreover
  from * have wf-interp (dec-word x) (dec-interp n FO x)
    (∀ i ∈ FO. case dec-interp n FO x ! i of Inl - ⇒ True | Inr - ⇒ False)
    (∀ i ∈ SO. case dec-interp n FO x ! i of Inl - ⇒ False | Inr - ⇒ True)
     using False x-wf-word[unfolded wf-word, unfolded σ-def o-apply set-concat
set-map set-n-lists, simplified] assms
      Inl-dec-interp-length[OF ballI, of FO x - n] Inr-dec-interp-length[of - n FO
x]
      dec-interp-Inl[of - FO n x] dec-interp-Inr[of - FO n x]
   by (fastforce simp add: dec-word-def split: sum.split)+
  ultimately have x ∈ ?R by blast
 }
 with False lang-flatten-INTERSECT[OF finite nonempty wf-rexp] have ?L ⊆
?R by (auto simp add: ENC-def)
 ultimately show ?thesis by blast
qed

lemma lang-ENC-formula:
 assumes wf-formula n φ
 shows lang n (ENC n (FOV φ)) − {[]} = {enc (w, I) | w I . length I = n ∧
wf-interp-for-formula (w, I) φ}
proof −
 from assms max-idx-vars have *: FOV φ ⊆ {0 ..< n} SOV φ ⊆ {0 ..< n} −
FOV φ by auto
 show ?thesis unfolding lang-ENC[OF *] by simp
qed
```

10.2 Welldefinedness of enc wrt. Models

lemma *enc-alt-def*:
enc (*w*, *x* # *I*) = *map-index* (λ*n* (*a*, *bs*). (*a*, (*case x of Inl p* ⇒ *n* = *p* | *Inr P* ⇒ *n* ∈ *P*) # *bs*)) (*enc* (*w*, *I*))
by (*auto simp*: *comp-def*)

lemma *enc-extend-interp*: *enc* (*w*, *I*) = *enc* (*w*′, *I*′) ⟹ *enc* (*w*, *x* # *I*) = *enc* (*w*′, *x* # *I*′)
unfolding *enc-alt-def* **by** *auto*

lemma *wf-interp-for-formula-FExists*:
⟦*wf-formula* (*length I*) (*FExists* φ); *w* ≠ []⟧⟹
wf-interp-for-formula (*w*, *I*) (*FExists* φ) ⟷
(∀ *p* < *length w*. *wf-interp-for-formula* (*w*, *Inl p* # *I*) φ)
by (*auto simp*: *nth-Cons*′ *split*: *if-split-asm*)

lemma *wf-interp-for-formula-any-Inl*: *wf-interp-for-formula* (*w*, *Inl p* # *I*) φ ⟹
∀ *p* < *length w*. *wf-interp-for-formula* (*w*, *Inl p* # *I*) φ
by (*auto simp*: *nth-Cons*′ *split*: *if-split-asm*)

lemma *wf-interp-for-formula-FEXISTS*:
⟦*wf-formula* (*length I*) (*FEXISTS* φ); *w* ≠ []⟧⟹
wf-interp-for-formula (*w*, *I*) (*FEXISTS* φ) ⟷ (∀ *P* ⊆ {*0* .. *length w* − *1*}. *wf-interp-for-formula* (*w*, *Inr P* # *I*) φ)
by (*auto simp*: *neq-Nil-conv nth-Cons*′)

lemma *wf-interp-for-formula-any-Inr*: *wf-interp-for-formula* (*w*, *Inr P* # *I*) φ ⟹
∀ *P* ⊆ {*0* .. *length w* − *1*}. *wf-interp-for-formula* (*w*, *Inr P* # *I*) φ
by (*cases w*) (*auto simp*: *nth-Cons*′ *split*: *if-split-asm*)

lemma *enc-word-length*: *enc* (*w*, *I*) = *enc* (*w*′, *I*′) ⟹ *length w* = *length w*′
by (*auto elim*: *map-index-eq-imp-length-eq*)

lemma *enc-length*:
assumes *w* ≠ [] *enc* (*w*, *I*) = *enc* (*w*′, *I*′)
shows *length I* = *length I*′
using *assms*
length-map[*of* (λ*x*. *case x of Inl p* ⇒ *0* = *p* | *Inr P* ⇒ *0* ∈ *P*) *I*]
length-map[*of* (λ*x*. *case x of Inl p* ⇒ *0* = *p* | *Inr P* ⇒ *0* ∈ *P*) *I*′]
by (*induct rule*: *list-induct2*[*OF enc-word-length*[*OF assms*(*2*)]]) *auto*

lemma *wf-interp-for-formula-FOr*:
wf-interp-for-formula (*w*, *I*) (*FOr* φ*1* φ*2*) =
(*wf-interp-for-formula* (*w*, *I*) φ*1* ∧ *wf-interp-for-formula* (*w*, *I*) φ*2*)
by *auto*

lemma *wf-interp-for-formula-FAnd*:
wf-interp-for-formula (*w*, *I*) (*FAnd* φ*1* φ*2*) =
(*wf-interp-for-formula* (*w*, *I*) φ*1* ∧ *wf-interp-for-formula* (*w*, *I*) φ*2*)

by *auto*

lemma *enc-wf-interp*:
assumes *wf-formula* (*length I*) φ *wf-interp-for-formula* (*w*, *I*) φ
shows *wf-interp-for-formula* (*dec-word* (*enc* (*w*, *I*)), *dec-interp* (*length I*) (*FOV* φ) (*enc* (*w*, *I*))) φ
(**is** *wf-interp-for-formula* (-, *?dec*) φ)
unfolding *dec-word-enc*
proof −
{ fix *i* **assume** *i*: *i* ∈ *FOV* φ
with *assms*(*2*) **have** ∃ *p*. *I* ! *i* = *Inl p* **by** (*cases I* ! *i*) *auto*
with *i assms* **have** ∃!*p*. *p* < *length* (*enc* (*w*, *I*)) ∧ *snd* (*enc* (*w*, *I*) ! *p*) ! *i*
by (*intro enc-unique*[*of w I i*]) (*auto intro*!: *bspec*[*OF max-idx-vars*] *split*: *sum.splits*)
} note ∗ = *this*
have ∀ *x* ∈ *set ?dec*. *case-sum* (λ*p*. *p* < *length w*) (λ*P*. ∀ *p*∈*P*. *p* < *length w*) *x*
proof (*intro ballI*, *split sum.split*, *safe*)
fix *p* **assume** *Inl p* ∈ *set ?dec*
thus *p* < *length w* **using** *Inl-dec-interp-length*[*OF ballI*[*OF* ∗]] **by** *auto*
next
fix *p P* **assume** *Inr P* ∈ *set ?dec p* ∈ *P*
thus *p* < *length w* **using** *Inr-dec-interp-length* **by** *fastforce*
qed
thus *wf-interp-for-formula* (*w*, *?dec*) φ
using *assms*
dec-interp-Inl[*of - FOV* φ *length I enc* (*w*, *I*), *OF - bspec*[*OF max-idx-vars*]]
dec-interp-Inr[*of - FOV* φ *length I enc* (*w*, *I*), *OF - bspec*[*OF max-idx-vars*]]
by (*fastforce split*: *sum.splits*)
qed

lemma *enc-welldef*: ⟦*enc* (*w*, *I*) = *enc* (*w*′, *I*′); *wf-formula* (*length I*) φ;
wf-interp-for-formula (*w*, *I*) φ; *wf-interp-for-formula* (*w*′, *I*′) φ⟧ ⟹
satisfies (*w*, *I*) φ ⟷ *satisfies* (*w*′, *I*′) φ
proof (*induction* φ *arbitrary*: *I I*′)
case (*FQ a m*)
let *?dec* = λ*w I*. (*dec-word* (*enc* (*w*, *I*)), *dec-interp* (*length I*) (*FOV* (*FQ a m*)) (*enc* (*w*, *I*)))
from *FQ*(*2*,*3*) **have** *satisfies* (*w*, *I*) (*FQ a m*) = *satisfies* (*?dec w I*) (*FQ a m*)
unfolding *dec-word-enc*
using *dec-interp-enc-Inl*[*of length I* {*m*} *w I m*]
by (*auto intro*: *nth-mem dest*: *dec-interp-not-Inr split*: *sum.splits*) (*metis nth-mem*)+
moreover
from *FQ*(*3*) **have** ∗: *w* ≠ [] **by** *simp*
from *FQ*(*2*,*4*) **have** *satisfies* (*w*′, *I*′) (*FQ a m*) = *satisfies* (*?dec w*′ *I*′) (*FQ a m*)
unfolding *dec-word-enc enc-length*[*OF* ∗ *FQ*(*1*)]
using *dec-interp-enc-Inl*[*of length I*′ {*m*} *w*′ *I*′ *m*]
by (*auto dest*: *dec-interp-not-Inr split*: *sum.splits*) (*metis nth-mem*)+

ultimately show *?case* **unfolding** *FQ(1) enc-length*[*OF* ∗ *FQ(1)*] **..**
next
case (*FLess m n*)
let *?dec* = λ*w I*. (*dec-word* (*enc* (*w*, *I*)), *dec-interp* (*length I*) (*FOV* (*FLess m n*)) (*enc* (*w*, *I*)))
from *FLess(2,3)* **have** *satisfies* (*w*, *I*) (*FLess m n*) = *satisfies* (*?dec* (*TYPE* (*'a*)) *w I*) (*FLess m n*)
unfolding *dec-word-enc*
using *dec-interp-enc-Inl*[*of length I* {*m*, *n*} *w I m*] *dec-interp-enc-Inl*[*of length I* {*m*, *n*} *w I n*]
by (*fastforce intro*: *nth-mem dest*: *dec-interp-not-Inr split*: *sum.splits*)
moreover
from *FLess(3)* **have** ∗: *w* ≠ [] **by** *simp*
from *FLess(2,4)* **have** *satisfies* (*w'*, *I'*) (*FLess m n*) = *satisfies* (*?dec* (*TYPE* (*'a*)) *w' I'*) (*FLess m n*)
unfolding *dec-word-enc enc-length*[*OF* ∗ *FLess(1)*]
using *dec-interp-enc-Inl*[*of length I'* {*m*, *n*} *w' I' m*] *dec-interp-enc-Inl*[*of length I'* {*m*, *n*} *w' I' n*]
by (*fastforce intro*: *nth-mem dest*: *dec-interp-not-Inr split*: *sum.splits*)
ultimately show *?case* **unfolding** *FLess(1) enc-length*[*OF* ∗ *FLess(1)*] **..**
next
case (*FIn m M*)
let *?dec* = λ*w I*. (*dec-word* (*enc* (*w*, *I*)), *dec-interp* (*length I*) (*FOV* (*FIn m M*)) (*enc* (*w*, *I*)))
from *FIn(2,3)* **have** *satisfies* (*w*, *I*) (*FIn m M*) = *satisfies* (*?dec* (*TYPE* (*'a*)) *w I*) (*FIn m M*)
unfolding *dec-word-enc*
using *dec-interp-enc-Inl*[*of length I* {*m*} *w I m*] *dec-interp-enc-Inr*[*of length I* {*m*} *w I M*]
by (*auto dest*: *dec-interp-not-Inr dec-interp-not-Inl split*: *sum.splits*) (*metis nth-mem*)+
moreover
from *FIn(3)* **have** ∗: *w* ≠ [] **by** *simp*
from *FIn(2,4)* **have** *satisfies* (*w'*, *I'*) (*FIn m M*) = *satisfies* (*?dec* (*TYPE* (*'a*)) *w' I'*) (*FIn m M*)
unfolding *dec-word-enc enc-length*[*OF* ∗ *FIn(1)*]
using *dec-interp-enc-Inl*[*of length I'* {*m*} *w' I' m*] *dec-interp-enc-Inr*[*of length I'* {*m*} *w' I' M*]
by (*auto dest*: *dec-interp-not-Inr dec-interp-not-Inl split*: *sum.splits*) (*metis nth-mem*)+
ultimately show *?case* **unfolding** *FIn(1) enc-length*[*OF* ∗ *FIn(1)*] **..**
next
case (*FOr φ1 φ2*) **show** *?case* **unfolding** *satisfies.simps*(*5*)
proof (*intro disj-cong*)
from *FOr(3−6)* **show** *satisfies* (*w*, *I*) *φ1* = *satisfies* (*w'*, *I'*) *φ1*
by (*intro FOr(1)*) *auto*
next
from *FOr(3−6)* **show** *satisfies* (*w*, *I*) *φ2* = *satisfies* (*w'*, *I'*) *φ2*
by (*intro FOr(2)*) *auto*

```
  qed
next
  case (FAnd φ1 φ2) show ?case unfolding satisfies.simps(6)
  proof (intro conj-cong)
    from FAnd(3−6) show satisfies (w, I) φ1 = satisfies (w', I') φ1
      by (intro FAnd(1)) auto
  next
    from FAnd(3−6) show satisfies (w, I) φ2 = satisfies (w', I') φ2
      by (intro FAnd(2)) auto
  qed
next
  case (FExists φ)
  hence w ≠ [] w' ≠ [] by auto
  hence length: length w = length w' length I = length I'
   using enc-word-length[OF FExists.prems(1)] enc-length[OF - FExists.prems(1)]
by auto
  show ?case
  proof
    assume satisfies (w, I) (FExists φ)
    with FExists.prems(3) obtain p where p < length w satisfies (w, Inl p # I)
φ
      by (auto intro: ord-less-eq-trans[OF le-imp-less-Suc Suc-pred])
    moreover
    with FExists.prems have satisfies (w', Inl p # I') φ
    proof (intro iffD1[OF FExists.IH[of Inl p # I Inl p # I']])
      from FExists.prems(2,3) ‹p < length w› show wf-interp-for-formula (w, Inl
p # I) φ
        by (blast dest: wf-interp-for-formula-FExists[of I, OF - ‹w ≠ []›])
    next
      from FExists.prems(2,4) ‹p < length w› show wf-interp-for-formula (w', Inl
p # I') φ
        unfolding length by (blast dest: wf-interp-for-formula-FExists[of I', OF -
‹w' ≠ []›])
    qed (auto elim: enc-extend-interp simp del: enc.simps)
    ultimately show satisfies (w', I') (FExists φ) using length by (auto intro!:
exI[of - p])
  next
    assume satisfies (w', I') (FExists φ)
    with FExists.prems(1,2,4) obtain p where p < length w' satisfies (w', Inl p
# I') φ
      by (auto intro: ord-less-eq-trans[OF le-imp-less-Suc Suc-pred])
    moreover
    with FExists.prems have satisfies (w, Inl p # I) φ
    proof (intro iffD2[OF FExists.IH[of Inl p # I Inl p # I']])
      from FExists.prems(2,3) ‹p < length w'› show wf-interp-for-formula (w, Inl
p # I) φ
       unfolding length[symmetric] by (blast dest: wf-interp-for-formula-FExists[of
I, OF - ‹w ≠ []›])
    next
```

```
    from FExists.prems(2,4) ‹p < length w'› show wf-interp-for-formula (w', Inl
p # I') φ
      unfolding length by (blast dest: wf-interp-for-formula-FExists[of I', OF -
‹w' ≠ []›])
    qed (auto elim: enc-extend-interp simp del: enc.simps)
    ultimately show satisfies (w, I) (FExists φ) using length by (auto intro!:
exI[of - p])
  qed
next
  case (FEXISTS φ)
  hence w ≠ [] w' ≠ [] by auto
  hence length: length w = length w' length I = length I'
   using enc-word-length[OF FEXISTS.prems(1)] enc-length[OF - FEXISTS.prems(1)]
by auto
  show ?case
  proof
    assume satisfies (w, I) (FEXISTS φ)
    then obtain P where P: P ⊆ {0 .. length w − 1} satisfies (w, Inr P # I)
φ by auto
    moreover
    with FEXISTS.prems have satisfies (w', Inr P # I') φ
    proof (intro iffD1[OF FEXISTS.IH[of Inr P # I Inr P # I']])
      from FEXISTS.prems(2,3) P(1) show wf-interp-for-formula (w, Inr P #
I) φ
        by (blast dest: wf-interp-for-formula-FEXISTS[of I, OF - ‹w ≠ []›])
    next
      from FEXISTS.prems(2,4) P(1) show wf-interp-for-formula (w', Inr P #
I') φ
        unfolding length by (blast dest: wf-interp-for-formula-FEXISTS[of I', OF
- ‹w' ≠ []›])
    qed (auto elim: enc-extend-interp simp del: enc.simps)
   ultimately show satisfies (w', I') (FEXISTS φ) using length by (auto intro!:
exI[of - P])
  next
    assume satisfies (w', I') (FEXISTS φ)
    then obtain P where P: P ⊆ {0 .. length w' − 1} satisfies (w', Inr P # I')
φ by auto
    moreover
    with FEXISTS.prems have satisfies (w, Inr P # I) φ
    proof (intro iffD2[OF FEXISTS.IH[of Inr P # I Inr P # I']])
      from FEXISTS.prems(2,3) P(1) show wf-interp-for-formula (w, Inr P #
I) φ
     unfolding length[symmetric] by (blast dest: wf-interp-for-formula-FEXISTS[of
I, OF - ‹w ≠ []›])
    next
      from FEXISTS.prems(2,4) P(1) show wf-interp-for-formula (w', Inr P #
I') φ
        unfolding length by (blast dest: wf-interp-for-formula-FEXISTS[of I', OF
- ‹w' ≠ []›])
```

qed (*auto elim*: *enc-extend-interp simp del*: *enc.simps*)
ultimately show *satisfies* (*w*, *I*) (*FEXISTS* φ) **using** *length* **by** (*auto intro*!: *exI*[*of* - *P*])
qed
qed *auto*

lemma $lang_{M2L}$-*FOr*:
assumes *wf-formula n* (*FOr* φ_1 φ_2)
shows $lang_{M2L}$ *n* (*FOr* φ_1 φ_2) $\subseteq$
($lang_{M2L}$ *n* φ_1 $\cup$ $lang_{M2L}$ *n* φ_2) $\cap$ {*enc* (*w*, *I*) | *w I*. *length I* = *n* $\wedge$ *wf-interp-for-formula* (*w*, *I*) (*FOr* φ_1 φ_2)}
(**is** - $\subseteq$ (*?L1* $\cup$ *?L2*) $\cap$ *?ENC*)
proof (*intro equalityI subsetI*)
fix *x* **assume** *x* $\in$ $lang_{M2L}$ *n* (*FOr* φ_1 φ_2)
then obtain *w I* **where**
$*$: *x* = *enc* (*w*, *I*) *wf-interp-for-formula* (*w*, *I*) (*FOr* φ_1 φ_2) *length I* = *n length w* > *0* **and**
satisfies (*w*, *I*) φ_1 $\vee$ *satisfies* (*w*, *I*) φ_2 **unfolding** $lang_{M2L}$-*def* **by** *auto*
thus *x* $\in$ (*?L1* $\cup$ *?L2*) $\cap$ *?ENC*
proof (*elim disjE*)
assume *satisfies* (*w*, *I*) φ_1
with $*$ **have** *x* $\in$ *?L1* **using** *assms* **unfolding** $lang_{M2L}$-*def* **by** (*fastforce*)
with $*$ **show** *?thesis* **by** *auto*
next
assume *satisfies* (*w*, *I*) φ_2
with $*$ **have** *x* $\in$*?L2* **using** *assms* **unfolding** $lang_{M2L}$-*def* **by** (*fastforce*)
with $*$ **show** *?thesis* **by** *auto*
qed
qed

lemma $lang_{M2L}$-*FAnd*:
assumes *wf-formula n* (*FAnd* φ_1 φ_2)
shows $lang_{M2L}$ *n* (*FAnd* φ_1 φ_2) $\subseteq$
$lang_{M2L}$ *n* φ_1 $\cap$ $lang_{M2L}$ *n* φ_2 $\cap$ {*enc* (*w*, *I*) | *w I*. *length I* = *n* $\wedge$ *wf-interp-for-formula* (*w*, *I*) (*FAnd* φ_1 φ_2)}
(**is** - $\subseteq$ *?L1* $\cap$ *?L2* $\cap$ *?ENC*)
using *assms* **unfolding** $lang_{M2L}$-*def* **by** (*fastforce*)

10.3 From M2L to Regular expressions

fun *rexp-of* :: *nat* $\Rightarrow$ *'a formula* $\Rightarrow$ (*'a atom*) *rexp* **where**
rexp-of n (*FQ a m*) = *Inter* (*TIMES* [*Full*, *Atom* (*AQ m a*), *Full*]) (*ENC n* {*m*})
| *rexp-of n* (*FLess m1 m2*) = (*if m1* = *m2 then Zero else Inter*
(*TIMES* [*Full*, *Atom* (*Arbitrary-Except m1 True*), *Full*, *Atom* (*Arbitrary-Except m2 True*), *Full*])
(*ENC n* {*m1*, *m2*}))
| *rexp-of n* (*FIn m M*) =
Inter (*TIMES* [*Full*, *Atom* (*Arbitrary-Except2 m M*), *Full*]) (*ENC n* {*m*})
| *rexp-of n* (*FNot* φ) = *Inter* (*rexp.Not* (*rexp-of n* φ)) (*ENC n* (*FOV* (*FNot* φ)))

| *rexp-of n (FOr φ1 φ2) = Inter (Plus (rexp-of n φ1) (rexp-of n φ2)) (ENC n (FOV (FOr φ1 φ2)))*
| *rexp-of n (FAnd φ1 φ2) = INTERSECT [rexp-of n φ1, rexp-of n φ2, ENC n (FOV (FAnd φ1 φ2))]*
| *rexp-of n (FExists φ) = Pr (rexp-of (n + 1) φ)*
| *rexp-of n (FEXISTS φ) = Pr (rexp-of (n + 1) φ)*

fun *rexp-of-alt :: nat ⇒ 'a formula ⇒ ('a atom) rexp* **where**
rexp-of-alt n (FQ a m) = TIMES [Full, Atom (AQ m a), Full]
| *rexp-of-alt n (FLess m1 m2) = (if m1 = m2 then Zero else*
TIMES [Full, Atom (Arbitrary-Except m1 True), Full, Atom (Arbitrary-Except m2 True), Full])
| *rexp-of-alt n (FIn m M) = TIMES [Full, Atom (Arbitrary-Except2 m M), Full]*
| *rexp-of-alt n (FNot φ) = rexp.Not (rexp-of-alt n φ)*
| *rexp-of-alt n (FOr φ1 φ2) = Plus (rexp-of-alt n φ1) (rexp-of-alt n φ2)*
| *rexp-of-alt n (FAnd φ1 φ2) = Inter (rexp-of-alt n φ1) (rexp-of-alt n φ2)*
| *rexp-of-alt n (FExists φ) = Pr (Inter (rexp-of-alt (n + 1) φ) (ENC (n + 1) (FOV φ)))*
| *rexp-of-alt n (FEXISTS φ) = Pr (Inter (rexp-of-alt (n + 1) φ) (ENC (n + 1) (FOV φ)))*

definition *rexp-of' n φ = Inter (rexp-of-alt n φ) (ENC n (FOV φ))*

fun *rexp-of-alt' :: nat ⇒ 'a formula ⇒ ('a atom) rexp* **where**
rexp-of-alt' n (FQ a m) = TIMES [Full, Atom (AQ m a), Full]
| *rexp-of-alt' n (FLess m1 m2) = (if m1 = m2 then Zero else*
TIMES [Full, Atom (Arbitrary-Except m1 True), Full, Atom (Arbitrary-Except m2 True), Full])
| *rexp-of-alt' n (FIn m M) = TIMES [Full, Atom (Arbitrary-Except2 m M), Full]*
| *rexp-of-alt' n (FNot φ) = rexp.Not (rexp-of-alt' n φ)*
| *rexp-of-alt' n (FOr φ1 φ2) = Plus (rexp-of-alt' n φ1) (rexp-of-alt' n φ2)*
| *rexp-of-alt' n (FAnd φ1 φ2) = Inter (rexp-of-alt' n φ1) (rexp-of-alt' n φ2)*
| *rexp-of-alt' n (FExists φ) = Pr (Inter (rexp-of-alt' (n + 1) φ) (ENC (n + 1) {0}))*
| *rexp-of-alt' n (FEXISTS φ) = Pr (rexp-of-alt' (n + 1) φ)*

definition *rexp-of'' n φ = Inter (rexp-of-alt' n φ) (ENC n (FOV φ))*

theorem *lang$_{M2L}$-rexp-of: wf-formula n φ ⟹ lang$_{M2L}$ n φ = lang n (rexp-of n φ) − {[]}*
(**is** *- ⟹ - = ?L n φ*)
proof (*induct φ arbitrary: n*)
case (*FQ a m*)
show *?case*
proof (*intro equalityI subsetI*)
fix *x* **assume** *x ∈ lang$_{M2L}$ n (FQ a m)*
then obtain *w I* **where**
∗: x = enc (w, I) wf-interp-for-formula (w, I) (FQ a m) satisfies (w, I) (FQ a m)

```
    length I = n
    unfolding lang_M2L-def by blast
  with FQ(1) obtain p where p: p < length w I ! m = Inl p w ! p = a
    by (auto simp: all-set-conv-all-nth split: sum.splits)
  with *(1) have x = take p (enc (w, I)) @ [enc-atom I p a] @ drop (p + 1)
(enc (w, I))
    using id-take-nth-drop[of p enc (w, I)] by auto
  moreover from *(4) FQ(1) p(2)
   have [enc-atom I p a] ∈ lang n (Atom (AQ m a))
    by (intro enc-atom-lang-AQ) auto
  moreover from *(2,4) have take p (enc (w, I)) ∈ lang n (Full)
    by (auto intro!: enc-atom-σ dest!: in-set-takeD)
  moreover from *(2,4) have drop (Suc p) (enc (w, I)) ∈ lang n (Full)
    by (auto intro!: enc-atom-σ dest!: in-set-dropD)
  ultimately show x ∈ ?L n (FQ a m) using *(1,2,4)
    unfolding rexp-of.simps lang.simps(6,9) rexp-of-list.simps Int-Diff
      lang-ENC-formula[OF FQ, unfolded FOV.simps]
    by (auto elim: ssubst simp del: o-apply append.simps lang.simps)
 next
  fix x assume x: x ∈ ?L n (FQ a m)
  with FQ obtain w I p where m: I ! m = Inl p m < length I and
    wI: x = enc (w, I) length I = n wf-interp-for-formula (w, I) (FQ a m)
      unfolding rexp-of.simps lang.simps lang-ENC-formula[OF FQ, unfolded
FOV.simps] Int-Diff
    by atomize-elim (auto split: sum.splits)
  hence wf-interp-for-formula (dec-word x, dec-interp n {m} x) (FQ a m) un-
folding wI(1)
    using enc-wf-interp[OF FQ(1)[folded wI(2)]] by auto
  moreover
  from x obtain u1 u u2 where x = u1 @ u @ u2 u ∈ lang n (Atom (AQ m
a))
    unfolding rexp-of.simps lang.simps rexp-of-list.simps using concE by fast
  with FQ(1) obtain v where v: x = u1 @ [v] @ u2 snd v ! m fst v = a
    using AQ-D[of u n m a] by fastforce
  hence u: length u1 < length x by auto
  { from v have snd (x ! length u1) ! m by auto
    moreover
    from m wI have p < length x snd (x ! p) ! m
      by (fastforce intro: nth-mem split: sum.splits)+
    moreover
    from m wI have ex1: ∃!p. p < length x ∧ snd (x ! p) ! m unfolding wI(1)
by (intro enc-unique) auto
    ultimately have p = length u1 using u by auto
  } note * = this
  from v have v = enc (w, I) ! length u1 unfolding wI(1) by simp
   hence a = w ! length u1 using nth-map[OF u, of fst] unfolding wI(1)
v(3)[symmetric] by auto
  with * m wI have satisfies (dec-word x, dec-interp n {m} x) (FQ a m)
    unfolding dec-word-enc[of w I, folded wI(1)]
```

by (*auto simp del*: *enc.simps dest*: *dec-interp-not-Inr split*: *sum.splits*)
(*fastforce dest*!: *dec-interp-enc-Inl intro*: *nth-mem split*: *sum.splits*)
moreover from *wI* **have** *wf-word n x* **unfolding** *wf-word* **by** (*auto intro*!: *enc-atom-σ*)
ultimately show $x \in lang_{M2L}\ n\ (FQ\ a\ m)$ **unfolding** $lang_{M2L}$*-def* **using** *m wI*(*3*)
by (*auto simp del*: *enc.simps intro*!: *exI*[*of - dec-word x*] *exI*[*of - dec-interp n* {*m*} *x*]
intro: *sym*[*OF enc-dec*[*OF - ballI*[*OF impI*[*OF enc-unique*[*of w I*, *folded wI*(*1*)]]]]])
qed
next
case (*FLess m m'*)
show *?case*
proof (*cases m = m'*)
case *False*
thus *?thesis*
proof (*intro equalityI subsetI*)
fix *x* **assume** $x \in lang_{M2L}\ n\ (FLess\ m\ m')$
then obtain *w I* **where**
∗: $x = enc\ (w, I)$ *wf-interp-for-formula* (*w*, *I*) (*FLess m m'*) *satisfies* (*w*, *I*) (*FLess m m'*)
length I = n
unfolding $lang_{M2L}$*-def* **by** *blast*
with *FLess*(*1*) **obtain** *p q* **where** *pq*: *p < length w I* ! *m = Inl p q < length w I* ! *m' = Inl q p < q*
by (*auto simp*: *all-set-conv-all-nth split*: *sum.splits*)
with ∗(*1*) **have** *x = take p* (*enc* (*w*, *I*)) @ [*enc-atom I p* (*w* ! *p*)] @ *drop* (*p* + *1*) (*enc* (*w*, *I*))
using *id-take-nth-drop*[*of p enc* (*w*, *I*)] **by** *auto*
also have *drop* (*p* + *1*) (*enc* (*w*, *I*)) = *take* (*q* − *p* − *1*) (*drop* (*p* + *1*) (*enc* (*w*, *I*))) @
[*enc-atom I q* (*w* ! *q*)] @ *drop* (*q* − *p*) (*drop* (*p* + *1*) (*enc* (*w*, *I*))) (**is** - = *?LHS*)
using *id-take-nth-drop*[*of q* − *p* − *1 drop* (*p* + *1*) (*enc* (*w*, *I*))] *pq* **by** *auto*
finally have *x = take p* (*enc* (*w*, *I*)) @ [*enc-atom I p* (*w* ! *p*)] @ *?LHS* **.**
moreover from ∗(*2*,*4*) *FLess*(*1*) *pq*(*1*,*2*)
have [*enc-atom I p* (*w* ! *p*)] ∈ *lang n* (*Atom* (*Arbitrary-Except m True*))
by (*intro enc-atom-lang-Arbitrary-Except-True*) *auto*
moreover from ∗(*2*,*4*) *FLess*(*1*) *pq*(*3*,*4*)
have [*enc-atom I q* (*w* ! *q*)] ∈ *lang n* (*Atom* (*Arbitrary-Except m' True*))
by (*intro enc-atom-lang-Arbitrary-Except-True*) *auto*
moreover from ∗(*2*,*4*) **have** *take p* (*enc* (*w*, *I*)) ∈ *lang n* (*Full*)
by (*auto intro*!: *enc-atom-σ dest*!: *in-set-takeD*)
moreover from ∗(*2*,*4*) **have** *take* (*q* − *p* − *1*) (*drop* (*Suc p*) (*enc* (*w*, *I*))) ∈ *lang n* (*Full*)
by (*auto intro*!: *enc-atom-σ dest*!: *in-set-dropD in-set-takeD*)
moreover from ∗(*2*,*4*) **have** *drop* (*q* − *p*) (*drop* (*Suc p*) (*enc* (*w*, *I*))) ∈ *lang n* (*Full*)

```
    by (auto intro!: enc-atom-σ dest!: in-set-dropD)
  ultimately show x ∈ ?L n (FLess m m') using ∗(1,2,4)
    unfolding rexp-of.simps lang.simps(6,9) rexp-of-list.simps Int-Diff
      lang-ENC-formula[OF FLess, unfolded FOV.simps] if-not-P[OF False]
    by (auto elim: ssubst simp del: o-apply append.simps lang.simps)
next
  fix x assume x: x ∈ ?L n (FLess m m')
  with FLess obtain w I where
    wI: x = enc (w, I) length I = n wf-interp-for-formula (w, I) (FLess m m')
    unfolding rexp-of.simps lang.simps lang-ENC-formula[OF FLess, unfolded
FOV.simps] Int-Diff
      if-not-P[OF False]
    by (fastforce split: sum.splits)
  with FLess obtain p p' where m: I ! m = Inl p m < length I I ! m' = Inl
p' m' < length I
    by (auto split: sum.splits)
  with wI have wf-interp-for-formula (dec-word x, dec-interp n {m, m'} x)
(FLess m m') unfolding wI(1)
    using enc-wf-interp[OF FLess(1)[folded wI(2)]] by auto
  moreover
  from x obtain u1 u u2 u' u3 where x = u1 @ u @ u2 @ u' @ u3
    u ∈ lang n (Atom (Arbitrary-Except m True))
    u' ∈ lang n (Atom (Arbitrary-Except m' True))
    unfolding rexp-of.simps lang.simps rexp-of-list.simps if-not-P[OF False]
using concE by fast
  with FLess(1) obtain v v' where v: x = u1 @ [v] @ u2 @ [v'] @ u3 snd v
! m snd v' ! m'
  using Arbitrary-ExceptD[of u n m True] Arbitrary-ExceptD[of u' n m' True]
by fastforce
  hence u: length u1 < length x and u': Suc (length u1 + length u2) < length
x (is ?u' < -) by auto
  { from v have snd (x ! length u1) ! m by auto
    moreover
    from m wI have p < length x snd (x ! p) ! m
      by (fastforce intro: nth-mem split: sum.splits)+
    moreover
    from m wI have ex1: ∃!p. p < length x ∧ snd (x ! p) ! m unfolding wI(1)
by (intro enc-unique) auto
    ultimately have p = length u1 using u by auto
  }
  { from v have snd (x ! ?u') ! m' by (auto simp: nth-append)
    moreover
    from m wI have p' < length x snd (x ! p') ! m'
      by (fastforce intro: nth-mem split: sum.splits)+
    moreover
    from m wI have ex1: ∃!p. p < length x ∧ snd (x ! p) ! m' unfolding
wI(1) by (intro enc-unique) auto
    ultimately have p' = ?u' using u' by auto
  } note ∗ = this ‹p = length u1›
```

```
      with ∗ m wI have satisfies (dec-word x, dec-interp n {m, m'} x) (FLess m
m')
       unfolding dec-word-enc[of w I, folded wI(1)]
       by (auto simp del: enc.simps dest: dec-interp-not-Inr split: sum.splits)
         (fastforce dest!: dec-interp-enc-Inl intro: nth-mem split: sum.splits)
      moreover from wI have wf-word n x unfolding wf-word by (auto intro!:
enc-atom-σ)
       ultimately show x ∈ lang_M2L n (FLess m m') unfolding lang_M2L-def
using m wI(3)
       by (auto simp del: enc.simps intro!: exI[of - dec-word x] exI[of - dec-interp
n {m, m'} x]
          intro: sym[OF enc-dec[OF - ballI[OF impI[OF enc-unique[of w I, folded
wI(1)]]]]])
    qed
  qed (simp add: lang_M2L-def del: o-apply)
next
  case (FIn m M)
  show ?case
  proof (intro equalityI subsetI)
    fix x assume x ∈ lang_M2L n (FIn m M)
    then obtain w I where
      ∗: x = enc (w, I) wf-interp-for-formula (w, I) (FIn m M) satisfies (w, I)
(FIn m M)
       length I = n
      unfolding lang_M2L-def by blast
    with FIn(1) obtain p P where p: p < length w I ! m = Inl p I ! M = Inr
P p ∈ P
      by (auto simp: all-set-conv-all-nth split: sum.splits)
    with ∗(1) have x = take p (enc (w, I)) @ [enc-atom I p (w ! p)] @ drop (p
+ 1) (enc (w, I))
      using id-take-nth-drop[of p enc (w, I)] by auto
    moreover
   from ∗(2,4) FIn(1) p have [enc-atom I p (w ! p)] ∈ lang n (Atom (Arbitrary-Except2
m M))
      by (intro enc-atom-lang-Arbitrary-Except2) auto
    moreover from ∗(2,4) have take p (enc (w, I)) ∈ lang n (Full)
      by (auto intro!: enc-atom-σ dest!: in-set-takeD)
    moreover from ∗(2,4) have drop (Suc p) (enc (w, I)) ∈ lang n (Full)
      by (auto intro!: enc-atom-σ dest!: in-set-dropD)
    ultimately show x ∈ ?L n (FIn m M) using ∗(1,2,4)
      unfolding rexp-of.simps lang.simps(6,9) rexp-of-list.simps Int-Diff
        lang-ENC-formula[OF FIn, unfolded FOV.simps]
      by (auto elim: ssubst simp del: o-apply append.simps lang.simps)
  next
    fix x assume x: x ∈ ?L n (FIn m M)
   with FIn obtain w I where wI: x = enc (w, I) length I = n wf-interp-for-formula
(w, I) (FIn m M)
       unfolding rexp-of.simps lang.simps lang-ENC-formula[OF FIn, unfolded
FOV.simps] Int-Diff
```

```
    by (fastforce split: sum.splits)
   with FIn obtain p P where m: I ! m = Inl p m < length I I ! M = Inr P
M < length I by (auto split: sum.splits)
   with wI have wf-interp-for-formula (dec-word x, dec-interp n {m} x) (FIn m
M) unfolding wI(1)
    using enc-wf-interp[OF FIn(1)[folded wI(2)]] by auto
   moreover
   from x obtain u1 u u2 where x = u1 @ u @ u2
    u ∈ lang n (Atom (Arbitrary-Except2 m M))
    unfolding rexp-of.simps lang.simps rexp-of-list.simps using concE by fast
   with FIn(1) obtain v where v: x = u1 @ [v] @ u2 snd v ! m snd v ! M
    using Arbitrary-Except2D[of u n m M] by fastforce
   from v have u: length u1 < length x by auto
   { from v have snd (x ! length u1) ! m by auto
    moreover
    from m wI have p < length x snd (x ! p) ! m
     by (fastforce intro: nth-mem split: sum.splits)+
    moreover
    from m wI have ex1: ∃!p. p < length x ∧ snd (x ! p) ! m unfolding wI(1)
by (intro enc-unique) auto
    ultimately have p = length u1 using u by auto
   } note ∗ = this
   from v have v = enc (w, I) ! length u1 unfolding wI(1) by simp
   with v(3) m(3,4) u wI(1) have length u1 ∈ P by auto
   with ∗ m wI have satisfies (dec-word x, dec-interp n {m} x) (FIn m M)
    unfolding dec-word-enc[of w I, folded wI(1)]
    by (auto simp del: enc.simps dest: dec-interp-not-Inr dec-interp-not-Inl split:
sum.splits)
      (auto simp del: enc.simps dest!: dec-interp-enc-Inl dec-interp-enc-Inr dest:
nth-mem split: sum.splits)
    moreover from wI have wf-word n x unfolding wf-word by (auto intro!:
enc-atom-σ)
   ultimately show x ∈ lang_M2L n (FIn m M) unfolding lang_M2L-def using
m wI(3)
    by (auto simp del: enc.simps intro!: exI[of - dec-word x] exI[of - dec-interp n
{m} x]
      intro: sym[OF enc-dec[OF - ballI[OF impI[OF enc-unique[of w I, folded
wI(1)]]]]])
 qed
next
 case (FOr φ1 φ2)
 from FOr(3) have IH1: lang_M2L n φ1 = lang n (rexp-of n φ1) − {[]}
  by (intro FOr(1)) auto
 from FOr(3) have IH2: lang_M2L n φ2 = lang n (rexp-of n φ2) − {[]}
  by (intro FOr(2)) auto
 show ?case
 proof (intro equalityI subsetI)
  fix x assume x ∈ lang_M2L n (FOr φ1 φ2) thus x ∈ lang n (rexp-of n (FOr
φ1 φ2)) − {[]}
```

using $lang_{M2L}$-*FOr*[*OF FOr*(*3*)] **unfolding** *lang-ENC-formula*[*OF FOr*(*3*)] *rexp-of.simps lang.simps*
IH1 IH2 Int-Diff **by** *auto*
next
fix x **assume** $x \in lang\ n\ (rexp\text{-}of\ n\ (FOr\ \varphi_1\ \varphi_2)) - \{[]\}$
then obtain $w\ I$ **where** *or*: $x \in lang_{M2L}\ n\ \varphi_1 \vee x \in lang_{M2L}\ n\ \varphi_2$ **and** *wI*: $x = enc\ (w, I)\ length\ I = n$
wf-interp-for-formula (w, I) $(FOr\ \varphi_1\ \varphi_2)$
unfolding *lang-ENC-formula*[*OF FOr*(*3*)] *rexp-of.simps lang.simps IH1 IH2 Int-Diff* **by** *auto*
have $satisfies\ (w, I)\ \varphi_1 \vee satisfies\ (w, I)\ \varphi_2$
proof (*intro mp*[*OF disj-mono*[*OF impI impI*] *or*])
assume $x \in lang_{M2L}\ n\ \varphi_1$
with *wI*(*2*,*3*) *FOr*(*3*) **show** $satisfies\ (w, I)\ \varphi_1$
unfolding $lang_{M2L}$-*def wI*(*1*) *wf-interp-for-formula-FOr*
by (*auto simp del*: *enc.simps dest*!: *iffD2*[*OF enc-welldef*[*of - - - - φ_1*]])
next
assume $x \in lang_{M2L}\ n\ \varphi_2$
with *wI*(*2*,*3*) *FOr*(*3*) **show** $satisfies\ (w, I)\ \varphi_2$
unfolding $lang_{M2L}$-*def wI*(*1*) *wf-interp-for-formula-FOr*
by (*auto simp del*: *enc.simps dest*!: *iffD2*[*OF enc-welldef*[*of - - - - φ_2*]])
qed
with *wI* **show** $x \in lang_{M2L}\ n\ (FOr\ \varphi_1\ \varphi_2)$ **unfolding** $lang_{M2L}$-*def* **by** *auto*
qed
next
case $(FAnd\ \varphi_1\ \varphi_2)$
from *FAnd*(*3*) **have** *IH1*: $lang_{M2L}\ n\ \varphi_1 = lang\ n\ (rexp\text{-}of\ n\ \varphi_1) - \{[]\}$
by (*intro FAnd*(*1*)) *auto*
from *FAnd*(*3*) **have** *IH2*: $lang_{M2L}\ n\ \varphi_2 = lang\ n\ (rexp\text{-}of\ n\ \varphi_2) - \{[]\}$
by (*intro FAnd*(*2*)) *auto*
show *?case*
proof (*intro equalityI subsetI*)
fix x **assume** $x \in lang_{M2L}\ n\ (FAnd\ \varphi_1\ \varphi_2)$ **thus** $x \in lang\ n\ (rexp\text{-}of\ n\ (FAnd\ \varphi_1\ \varphi_2)) - \{[]\}$
using $lang_{M2L}$-*FAnd*[*OF FAnd*(*3*)] **unfolding** *lang-ENC-formula*[*OF FAnd*(*3*)] *rexp-of.simps*
rexp-of-list.simps lang.simps IH1 IH2 Int-Diff **by** *auto*
next
fix x **assume** $x \in lang\ n\ (rexp\text{-}of\ n\ (FAnd\ \varphi_1\ \varphi_2)) - \{[]\}$
then obtain $w\ I$ **where** *and*: $x \in lang_{M2L}\ n\ \varphi_1 \wedge x \in lang_{M2L}\ n\ \varphi_2$ **and** *wI*: $x = enc\ (w, I)\ length\ I = n$
wf-interp-for-formula (w, I) $(FAnd\ \varphi_1\ \varphi_2)$
unfolding *lang-ENC-formula*[*OF FAnd*(*3*)] *rexp-of.simps rexp-of-list.simps lang.simps IH1 IH2*
Int-Diff **by** *auto*
have $satisfies\ (w, I)\ \varphi_1 \wedge satisfies\ (w, I)\ \varphi_2$
proof (*intro mp*[*OF conj-mono*[*OF impI impI*] *and*])
assume $x \in lang_{M2L}\ n\ \varphi_1$
with *wI*(*2*,*3*) *FAnd*(*3*) **show** $satisfies\ (w, I)\ \varphi_1$

```
      unfolding lang_M2L-def wI(1) wf-interp-for-formula-FAnd
      by (auto simp del: enc.simps dest!: iffD2[OF enc-welldef[of - - - - φ1]])
    next
     assume x ∈ lang_M2L n φ2
     with wI(2,3) FAnd(3) show satisfies (w, I) φ2
      unfolding lang_M2L-def wI(1) wf-interp-for-formula-FAnd
      by (auto simp del: enc.simps dest!: iffD2[OF enc-welldef[of - - - - φ2]])
    qed
   with wI show x ∈ lang_M2L n (FAnd φ1 φ2) unfolding lang_M2L-def by auto
  qed
next
  case (FNot φ)
  hence IH: ?L n φ =  lang_M2L n φ by simp
  show ?case
  proof (intro equalityI subsetI)
    fix x assume x ∈ lang_M2L n (FNot φ)
    then obtain w I where
      ∗: x = enc (w, I) wf-interp-for-formula (w, I) φ length I = n length w > 0
       and unsat: ¬ (satisfies (w, I) φ)
       unfolding lang_M2L-def by auto
    { assume x ∈ ?L n φ
      with IH have satisfies (w, I) φ using enc-welldef[of - - w I φ, OF - - -
∗(2)] FNot(2)
       unfolding ∗(1,3) lang_M2L-def by auto
    }
    with unsat have x ∉ ?L n φ by blast
    with ∗ show x ∈ ?L n (FNot φ) unfolding rexp-of.simps lang.simps
        using lang-ENC-formula[OF FNot(2)] by (auto simp: comp-def intro!:
enc-atom-σ)
  next
    fix x assume x ∈ ?L n (FNot φ)
     with IH have x ∈ lang n (ENC n (FOV (FNot φ))) − {[]} and x: x ∉
lang_M2L n φ by (auto simp del: o-apply)
    then obtain w I where ∗: x = enc (w, I) wf-interp-for-formula (w, I) (FNot
φ) length I = n
      unfolding lang-ENC-formula[OF FNot(2)] by blast
    { assume ¬ satisfies (w, I) (FNot φ)
      with ∗ have x ∈ lang_M2L n φ unfolding lang_M2L-def by auto
    }
    with x ∗ show x ∈ lang_M2L n (FNot φ)  unfolding lang_M2L-def by blast
  qed
next
  case (FExists φ)
  show ?case
  proof (intro equalityI subsetI)
    fix x assume x ∈ lang_M2L n (FExists φ)
    then obtain w I p where
      ∗: x = enc (w, I) wf-interp-for-formula (w, I) (FExists φ)
       length I = n length w > 0 p ∈ {0 .. length w − 1} satisfies (w, Inl p # I) φ
```

```
      unfolding lang_M2L-def by auto
    with FExists(2) have enc (w, Inl p # I) ∈ ?L (Suc n) φ
      by (intro subsetD[OF equalityD1[OF FExists(1)], of Suc n enc (w, Inl p #
I)])
         (auto simp: lang_M2L-def nth-Cons' ord-less-eq-trans[OF le-imp-less-Suc
Suc-pred[OF ∗(4)]]
        split: if-split-asm sum.splits intro!: exI[of - w] exI[of - Inl p # I])
    with ∗(1) show x ∈ ?L n (FExists φ)
      by (auto simp: map-index intro!: image-eqI[of - map π] simp del: o-apply)
(auto simp: π-def)
  next
    fix x assume x ∈ ?L n (FExists φ)
   then obtain x' where x: x = map π x' and x' ∈ ?L (Suc n) φ by (auto simp
del: o-apply)
    with FExists(2) have x' ∈ lang_M2L (Suc n) φ
      by (intro subsetD[OF equalityD2[OF FExists(1)], of Suc n x'])
         (auto split: if-split-asm sum.splits)
    then obtain w I' where
     ∗: x' = enc (w, I') wf-interp-for-formula (w, I') φ length I' = Suc n satisfies
(w, I') φ
      unfolding lang_M2L-def by auto
    moreover then obtain I_0 I where I' = I_0 # I by (cases I') auto
    moreover with FExists(2) ∗(2) obtain p where I_0 = Inl p p < length w
      by (auto simp: nth-Cons' split: sum.splits if-split-asm)
     ultimately have x = enc (w, I) wf-interp-for-formula (w, I) (FExists φ)
length I = n
      length w > 0 satisfies (w, I) (FExists φ) using FExists(2) unfolding x
      by (auto simp: map-tl nth-Cons' split: if-split-asm simp del: o-apply) (auto
simp: π-def)
     thus x ∈ lang_M2L n (FExists φ) unfolding lang_M2L-def by (auto intro!:
exI[of - w] exI[of - I])
  qed
next
  case (FEXISTS φ)
  show ?case
  proof (intro equalityI subsetI)
    fix x assume x ∈ lang_M2L n (FEXISTS φ)
    then obtain w I P where
     ∗: x = enc (w, I) wf-interp-for-formula (w, I) (FEXISTS φ)
      length I = n length w > 0 P ⊆ {0 .. length w − 1} satisfies (w, Inr P # I)
φ
      unfolding lang_M2L-def by auto
    from ∗(4,5) have ∀ p ∈ P. p < length w by (cases w) auto
    with ∗(2−4,6) FEXISTS(2) have enc (w, Inr P # I) ∈ ?L (Suc n) φ
      by (intro subsetD[OF equalityD1[OF FEXISTS(1)], of Suc n enc (w, Inr P
# I)])
         (auto simp: lang_M2L-def nth-Cons' split: if-split-asm sum.splits
          intro!: exI[of - w] exI[of - Inr P # I])
    with ∗(1) show x ∈ ?L n (FEXISTS φ)
```

by (*auto simp*: *map-index intro*!: *image-eqI*[*of* - *map* π] *simp del*: *o-apply*) (*auto simp*: π-*def*)
next
fix x **assume** $x \in ?L\ n$ (*FEXISTS* φ)
then obtain x' **where** x: $x = map\ \pi\ x'$ **and** x': *length* $x' > 0$ **and** $x' \in ?L$ (*Suc* n) φ **by** (*auto simp del*: *o-apply*)
with *FEXISTS*(*2*) **have** $x' \in lang_{M2L}$ (*Suc* n) φ
by (*intro subsetD*[*OF equalityD2*[*OF FEXISTS*(*1*)], *of Suc n x'*])
(*auto split*: *if-split-asm sum.splits*)
then obtain $w\ I'$ **where**
$*$: $x' = enc\ (w, I')$ *wf-interp-for-formula* (w, I') φ *length* $I' =$ *Suc n satisfies* (w, I') φ
unfolding $lang_{M2L}$-*def* **by** *auto*
moreover then obtain $I_0\ I$ **where** $I' = I_0\ \#\ I$ **by** (*cases* I') *auto*
moreover with *FEXISTS*(*2*) $*$(*2*) **obtain** P **where** $I_0 = Inr\ P$
by (*auto simp*: *nth-Cons' split*: *sum.splits if-split-asm*)
moreover have *length* $w \geq 1$ **using** x' $*$(*1*) **by** (*cases* w) *auto*
ultimately have $x = enc\ (w, I)$ *wf-interp-for-formula* (w, I) (*FEXISTS* φ) *length* $I = n$
length $w > 0$ *satisfies* (w, I) (*FEXISTS* φ) **using** *FEXISTS*(*2*) **unfolding** x
by (*auto simp add*: *map-tl nth-Cons' split*: *if-split-asm*
intro!: *exI*[*of* - *P*] *simp del*: *o-apply*) (*auto simp*: π-*def*)
thus $x \in lang_{M2L}\ n$ (*FEXISTS* φ) **unfolding** $lang_{M2L}$-*def* **by** (*auto intro*!: *exI*[*of* - w] *exI*[*of* - I])
qed
qed

lemma *wf-rexp-of*: *wf-formula* $n\ \varphi \Longrightarrow$ *wf* n (*rexp-of* $n\ \varphi$)
by (*induct* φ *arbitrary*: n) (*auto intro*: *wf-rexp-ENC simp*: *finite-FOV max-idx-vars*)

lemma *wf-rexp-of-alt*: *wf-formula* $n\ \varphi \Longrightarrow$ *wf* n (*rexp-of-alt* $n\ \varphi$)
by (*induct* φ *arbitrary*: n) (*auto simp*: *wf-rexp-ENC finite-FOV max-idx-vars*)

lemma *wf-rexp-of'*: *wf-formula* $n\ \varphi \Longrightarrow$ *wf* n (*rexp-of'* $n\ \varphi$)
unfolding *rexp-of'-def* **by** (*auto simp*: *wf-rexp-ENC wf-rexp-of-alt finite-FOV max-idx-vars*)

lemma *wf-rexp-of-alt'*: *wf-formula* $n\ \varphi \Longrightarrow$ *wf* n (*rexp-of-alt'* $n\ \varphi$)
by (*induct* φ *arbitrary*: n) (*auto simp*: *wf-rexp-ENC*)

lemma *wf-rexp-of''*: *wf-formula* $n\ \varphi \Longrightarrow$ *wf* n (*rexp-of''* $n\ \varphi$)
unfolding *rexp-of''-def* **by** (*auto simp*: *wf-rexp-ENC wf-rexp-of-alt' finite-FOV max-idx-vars*)

lemma *ENC-Not*: *ENC* n (*FOV* (*FNot* φ)) = *ENC* n (*FOV* φ)
unfolding *ENC-def* **by** *auto*

lemma *ENC-And*:
wf-formula n (*FAnd* $\varphi\ \psi$) $\Longrightarrow$ *lang* n (*ENC* n (*FOV* (*FAnd* $\varphi\ \psi$))) $-$ $\{[]\} \subseteq$ *lang*

n (*ENC* n (*FOV* φ)) ∩ *lang* n (*ENC* n (*FOV* ψ)) − {[]}
proof
fix x **assume** *wf*: *wf-formula* n (*FAnd* φ ψ) **and** x: $x \in$ *lang* n (*ENC* n (*FOV* (*FAnd* φ ψ))) − {[]}
hence *wf1*: *wf-formula* n φ **and** *wf2*: *wf-formula* n ψ **by** *auto*
from x **obtain** w I **where** *wI*: x = *enc* (w, I) *wf-interp-for-formula* (w, I) (*FAnd* φ ψ) *length* I = n
using *lang-ENC-formula*[*OF wf*] **by** *blast*
hence *wf-interp-for-formula* (w, I) φ *wf-interp-for-formula* (w, I) ψ
unfolding *wf-interp-for-formula-FAnd* **by** *auto*
hence $x \in$ (*lang* n (*ENC* n (*FOV* φ)) − {[]}) ∩ (*lang* n (*ENC* n (*FOV* ψ)) − {[]})
unfolding *lang-ENC-formula*[*OF wf1*] *lang-ENC-formula*[*OF wf2*] **using** *wI* **by** *auto*
thus $x \in$ *lang* n (*ENC* n (*FOV* φ)) ∩ *lang* n (*ENC* n (*FOV* ψ)) − {[]} **by** *blast*
qed

lemma *ENC-Or*:
wf-formula n (*FOr* φ ψ) ⟹ *lang* n (*ENC* n (*FOV* (*FOr* φ ψ))) − {[]} ⊆ *lang* n (*ENC* n (*FOV* φ)) ∩ *lang* n (*ENC* n (*FOV* ψ)) − {[]}
proof
fix x **assume** *wf*: *wf-formula* n (*FOr* φ ψ) **and** x: $x \in$ *lang* n (*ENC* n (*FOV* (*FOr* φ ψ))) − {[]}
hence *wf1*: *wf-formula* n φ **and** *wf2*: *wf-formula* n ψ **by** *auto*
from x **obtain** w I **where** *wI*: x = *enc* (w, I) *wf-interp-for-formula* (w, I) (*FOr* φ ψ) *length* I = n
using *lang-ENC-formula*[*OF wf*] **by** *blast*
hence *wf-interp-for-formula* (w, I) φ *wf-interp-for-formula* (w, I) ψ
unfolding *wf-interp-for-formula-FOr* **by** *auto*
hence $x \in$ (*lang* n (*ENC* n (*FOV* φ)) − {[]}) ∩ (*lang* n (*ENC* n (*FOV* ψ)) − {[]})
unfolding *lang-ENC-formula*[*OF wf1*] *lang-ENC-formula*[*OF wf2*] **using** *wI* **by** *auto*
thus $x \in$ *lang* n (*ENC* n (*FOV* φ)) ∩ *lang* n (*ENC* n (*FOV* ψ)) − {[]} **by** *blast*
qed

lemma *project-enc*: *map* π (*enc* (w, x # I)) = *enc* (w, I)
unfolding π-*def* **by** *auto*

lemma *list-list-eqI*:
assumes ∀ (-, x) ∈ *set xs*. $x \neq$ [] ∀ (-, y) ∈ *set ys*. $y \neq$ []
map (λ(-, x). *hd* x) *xs* = *map* (λ(-, x). *hd* x) *ys map* π *xs* = *map* π *ys*
shows *xs* = *ys*
proof −
from *assms*(*4*) **have** *length xs* = *length ys* **by** (*metis length-map*)
then show *?thesis* **using** *assms* **by** (*induct rule*: *list-induct2*) (*auto simp*: π-*def neq-Nil-conv*)
qed

lemma *project-enc-extend*:
assumes *map* π *x* = *enc* (*w*, *I*) $\forall$(-, *x*) $\in$ *set x*. *x* $\neq$ []
shows *x* = *enc* (*w*, *Inr* (*positions-in-row x 0*) # *I*)
proof −
from *arg-cong*[*OF assms*(*1*), *of map fst*] **have** *w*: *w* = *map fst x* **by** (*auto simp*: π*-def*)
show *?thesis*
proof (*rule list-list-eqI*[*OF assms*(*2*)], *unfold project-enc*)
show *map* (λ(-, *x*). *hd x*) *x* = *map* (λ(-, *x*). *hd x*) (*enc* (*w*, *Inr* (*positions-in-row x 0*) # *I*))
using *assms*(*2*) **unfolding** *enc.simps map-index positions-in-row w*
by (*intro nth-equalityI*) (*auto dest*!: *nth-mem simp*: *hd-conv-nth*)
qed (*auto simp*: *assms*(*1*))
qed

lemma *ENC-Exists*:
wf-formula n (*FExists* φ) $\Longrightarrow$ *lang n* (*ENC n* (*FOV* (*FExists* φ))) − {[]} = *map* π ‘ *lang* (*Suc n*) (*ENC* (*Suc n*) (*FOV* φ)) − {[]}
proof (*intro equalityI subsetI*)
fix *x* **assume** *wf*: *wf-formula n* (*FExists* φ) **and** *x*: *x* $\in$ *lang n* (*ENC n* (*FOV* (*FExists* φ))) − {[]}
hence *wf1*: *wf-formula* (*Suc n*) φ **by** *auto*
from *x* **obtain** *w I* **where** *wI*: *x* = *enc* (*w*, *I*) *wf-interp-for-formula* (*w*, *I*) (*FExists* φ) *length I* = *n*
using *lang-ENC-formula*[*OF wf*] **by** *blast*
with *x* **have** *w* $\neq$ [] **by** (*cases w*) *auto*
from *wI*(*2*) **obtain** *p* **where** *p* < *length w wf-interp-for-formula* (*w*, *Inl p* # *I*) φ
using *wf-interp-for-formula-FExists*[*OF wf*[*folded wI*(*3*)] ‹*w* $\neq$ []›] **by** *auto*
with *wI*(*3*) **have** *x* $\in$ *map* π ‘ (*lang* (*Suc n*) (*ENC* (*Suc n*) (*FOV* φ)) − {[]})
unfolding *wI*(*1*) *lang-ENC-formula*[*OF wf1*] *project-enc*[*symmetric*, *of w I Inl p*]
by (*intro imageI CollectI exI*[*of - w*] *exI*[*of - Inl p* # *I*]) *auto*
thus *x* $\in$ *map* π ‘ *lang* (*Suc n*) (*ENC* (*Suc n*) (*FOV* φ)) − {[]} **by** *blast*
next
fix *x* **assume** *wf*: *wf-formula n* (*FExists* φ) **and** *x* $\in$ *map* π ‘ *lang* (*Suc n*) (*ENC* (*Suc n*) (*FOV* φ)) − {[]}
hence *wf1*: *wf-formula* (*Suc n*) φ **and** *0* $\in$ *FOV* φ **and** *x*: *x* $\in$ *map* π ‘ (*lang* (*Suc n*) (*ENC* (*Suc n*) (*FOV* φ)) − {[]}) **by** *auto*
from *x* **obtain** *w I* **where** *wI*: *x* = *map* π (*enc* (*w*, *I*)) *wf-interp-for-formula* (*w*, *I*) φ *length I* = *Suc n*
using *lang-ENC-formula*[*OF wf1*] **by** *auto*
with ‹*0* $\in$ *FOV* φ› **obtain** *p I*′ **where** *I*: *I* = *Inl p* # *I*′ **by** (*cases I*) (*fastforce split*: *sum.splits*)+
with *wI* **have** *wtlI*: *x* = *enc* (*w*, *I*′) *length I*′ = *n* **unfolding** π*-def* **by** *auto*
with *x* **have** *w* $\neq$ [] **by** (*cases w*) *auto*
have *wf-interp-for-formula* (*w*, *I*′) (*FExists* φ)
using *wf-interp-for-formula-FExists*[*OF wf*[*folded wtlI*(*2*)] ‹*w* $\neq$ []›]
wf-interp-for-formula-any-Inl[*OF wI*(*2*)[*unfolded I*]] **..**

with *wtlI* **show** $x \in$ *lang* n (*ENC* n (*FOV* (*FExists* φ))) $-$ {[]} **unfolding** *lang-ENC-formula*[*OF wf*] **by** *blast*
qed

lemma *ENC-EXISTS*:
wf-formula n (*FEXISTS* φ) $\Longrightarrow$ *lang* n (*ENC* n (*FOV* (*FEXISTS* φ))) $-$ {[]} = *map* π ‘ *lang* (*Suc* n) (*ENC* (*Suc* n) (*FOV* φ)) $-$ {[]}
proof (*intro equalityI subsetI*)
fix x **assume** *wf*: *wf-formula* n (*FEXISTS* φ) **and** x: $x \in$ *lang* n (*ENC* n (*FOV* (*FEXISTS* φ))) $-$ {[]}
hence *wf1*: *wf-formula* (*Suc* n) φ **by** *auto*
from x **obtain** w I **where** *wI*: x = *enc* (w, I) *wf-interp-for-formula* (w, I) (*FEXISTS* φ) *length* I = n
using *lang-ENC-formula*[*OF wf*] **by** *blast*
with x **have** $w \neq$ [] **by** (*cases* w) *auto*
from *wI*(*2*) **obtain** P **where** $\forall\, p \in P.\ p <$ *length* w *wf-interp-for-formula* (w, *Inr* P # I) φ
using *wf-interp-for-formula-FEXISTS*[*OF wf*[*folded wI*(*3*)] ‹$w \neq$ []›] **by** *auto*
with *wI*(*3*) **have** $x \in$ *map* π ‘ (*lang* (*Suc* n) (*ENC* (*Suc* n) (*FOV* φ)) $-$ {[]})
unfolding *wI*(*1*) *lang-ENC-formula*[*OF wf1*] *project-enc*[*symmetric*, *of w I Inr P*]
by (*intro imageI CollectI exI*[*of - w*] *exI*[*of - Inr P # I*]) *auto*
thus $x \in$ *map* π ‘ *lang* (*Suc* n) (*ENC* (*Suc* n) (*FOV* φ)) $-$ {[]} **by** *blast*
next
fix x **assume** *wf*: *wf-formula* n (*FEXISTS* φ) **and** $x \in$ *map* π ‘ *lang* (*Suc* n) (*ENC* (*Suc* n) (*FOV* φ)) $-$ {[]}
hence *wf1*: *wf-formula* (*Suc* n) φ **and** $0 \in$ *SOV* φ **and** x: $x \in$ *map* π ‘ (*lang* (*Suc* n) (*ENC* (*Suc* n) (*FOV* φ)) $-$ {[]}) **by** *auto*
from x **obtain** w I **where** *wI*: x = *map* π (*enc* (w, I)) *wf-interp-for-formula* (w, I) φ *length* I = *Suc* n
using *lang-ENC-formula*[*OF wf1*] **by** *auto*
with ‹$0 \in$ *SOV* φ› **obtain** P I' **where** I: I = *Inr* P # I' **by** (*cases* I) (*fastforce split*: *sum.splits*)+
with *wI* **have** *wtlI*: x = *enc* (w, I') *length* I' = n **unfolding** π-*def* **by** *auto*
with x **have** $w \neq$ [] **by** (*cases* w) *auto*
have *wf-interp-for-formula* (w, I') (*FEXISTS* φ)
using *wf-interp-for-formula-FEXISTS*[*OF wf*[*folded wtlI*(*2*)] ‹$w \neq$ []›]
wf-interp-for-formula-any-Inr[*OF wI*(*2*)[*unfolded I*]] **..**
with *wtlI* **show** $x \in$ *lang* n (*ENC* n (*FOV* (*FEXISTS* φ))) $-$ {[]} **unfolding** *lang-ENC-formula*[*OF wf*] **by** *blast*
qed

lemma *map-project-empty*: *map* π ‘ A $-$ {[]} = *map* π ‘ (A $-$ {[]})
by *auto*

lemma $lang_{M2L}$-*rexp-of-rexp-of'*:
wf-formula n φ $\Longrightarrow$ *lang* n (*rexp-of* n φ) $-$ {[]} = *lang* n (*rexp-of'* n φ) $-$ {[]}
unfolding *rexp-of'-def* **proof** (*induction* φ *arbitrary*: n)

case (*FNot* φ)
hence *wf-formula n* φ **by** *simp*
with *FNot.IH* **show** *?case* **unfolding** *rexp-of.simps rexp-of-alt.simps lang.simps ENC-Not* **by** *blast*
next
case (*FAnd* φ_1 φ_2)
hence *wf1*: *wf-formula n* φ_1 **and** *wf2*: *wf-formula n* φ_2 **by** *force+*
from *FAnd.IH*(*1*)[*OF wf1*] *FAnd.IH*(*2*)[*OF wf2*] **show** *?case* **using** *ENC-And*[*OF FAnd.prems*]
unfolding *rexp-of.simps rexp-of-alt.simps lang.simps rexp-of-list.simps* **by** *blast*
next
case (*FOr* φ_1 φ_2)
hence *wf1*: *wf-formula n* φ_1 **and** *wf2*: *wf-formula n* φ_2 **by** *force+*
from *FOr.IH*(*1*)[*OF wf1*] *FOr.IH*(*2*)[*OF wf2*] **show** *?case* **using** *ENC-Or*[*OF FOr.prems*]
unfolding *rexp-of.simps rexp-of-alt.simps lang.simps* **by** *blast*
next
case (*FExists* φ)
hence *wf*: *wf-formula* (*n* + *1*) φ **by** *auto*
show *?case* **using** *ENC-Exists*[*OF FExists.prems*]
unfolding *rexp-of.simps rexp-of-alt.simps lang.simps map-project-empty FExists.IH*[*OF wf*] **by** *auto*
next
case (*FEXISTS* φ)
hence *wf*: *wf-formula* (*n* + *1*) φ **by** *auto*
show *?case* **using** *ENC-EXISTS*[*OF FEXISTS.prems*]
unfolding *rexp-of.simps rexp-of-alt.simps lang.simps map-project-empty FEXISTS.IH*[*OF wf*] **by** *auto*
qed *auto*

lemma *Int-Diff-both*: $A \cap B - C = (A - C) \cap (B - C)$
by *auto*

lemma *lang-ENC-split*:
assumes *finite X* $X = Y1 \cup Y2$ $n = 0 \vee (\forall p \in X.\ p < n)$
shows *lang n* (*ENC n X*) = *lang n* (*ENC n Y1*) $\cap$ *lang n* (*ENC n Y2*)
unfolding *ENC-def lang-INTERSECT* **using** *assms lang-subset-lists*[*OF wf-rexp-valid-ENC*, *of n*] **by** *auto*

lemma *map-project-Int-ENC*:
assumes $0 \notin X$ $X \subseteq \{0\ ..<\ n + 1\}$ $Z \subseteq$ *lists* ((*set* $\circ$ σ Σ) (*n* + *1*))
shows *map* π ‘ ($Z \cap$ *lang* (*n* + *1*) (*ENC* (*n* + *1*) *X*) − {[]}) =
map π ‘ *Z* $\cap$ *lang n* (*ENC n* ((λ*x*. *x* − *1*) ‘ *X*)) − {[]}
proof −
let *?Y* = {*0* ..< *n* + *1*} − *X*
let *?fX* = (λ*x*. *x* − *1*) ‘ *X*
let *?fY* = {*0* ..< *n*} − (λ*x*. *x* − *1*) ‘ *X*
from *assms* **have** ∗: (λ*x*. *x* − *1*) ‘ *X* ⊆ {*0* ..< *n*} **by** (*cases n*) *auto*
show *?thesis* **unfolding** *Int-Diff lang-ENC*[*OF assms*(*2*) *subset-refl*] *lang-ENC*[*OF*

```
∗ subset-refl]
  proof (safe elim!: imageI)
    fix w I
    assume ∗: length I = n + 1 w ≠ []
      ∀ i∈X. case I ! i of Inl x ⇒ True | Inr x ⇒ False
      ∀ i∈?Y. case I ! i of Inl x ⇒ False | Inr x ⇒ True
      ∀ a∈set w. a ∈ set Σ Ball (set I) (case-sum (λp. p < length w) (λP. ∀ p∈P.
p < length w))
    then obtain p Is where I = p # Is by (cases I) auto
    then show ∃ w' I'.
      map π (enc (w, I)) = enc (w', I') ∧
      length I' = n ∧ (0 < length w' ∧ (∀ a∈set w'. a ∈ set Σ) ∧
      Ball (set I') (case-sum (λp. p < length w') (λP. ∀ p∈P. p < length w'))) ∧
      (∀ i∈?fX. case I' ! i of Inl x ⇒ True | Inr x ⇒ False) ∧
      (∀ i∈?fY. case I' ! i of Inl x ⇒ False | Inr x ⇒ True)
    proof (hypsubst, intro exI[of - w] exI[of - Is] conjI ballI project-enc)
      fix i assume i ∈ ?fY
      then show case Is ! i of Inl x ⇒ False | Inr x ⇒ True
        using ∗[unfolded ‹I = p # Is›] assms(1)
          by (cases i = 0) (fastforce simp: nth-Cons' image-iff split: sum.splits
if-splits)+
     qed (insert ∗[unfolded ‹I = p # Is›] assms(1), auto simp: nth-Cons' split:
sum.splits if-splits)
  next
    fix x w I
    assume ∗: w ≠ [] x ∈ Z map π x = enc (w, I)
      ∀ i∈?fX. case I ! i of Inl x ⇒ True | Inr x ⇒ False
      ∀ i∈{0 ..< length I} − ?fX. case I ! i of Inl x ⇒ False | Inr x ⇒ True
      ∀ a∈set w. a ∈ set Σ Ball (set I) (case-sum (λp. p < length w) (λP. ∀ p∈P.
p < length w))
    moreover from assms(1) have ∀ x ∈ X. x > 0 ⋀x y. x − Suc 0 = y − Suc
0 ⟷
      x = y ∨ (x = 0 ∧ y = Suc 0) ∨ (x = Suc 0 ∧ y = 0)
       by (metis neq0-conv) (metis One-nat-def Suc-diff-1 diff-0-eq-0 diff-self-eq-0
neq0-conv)
    moreover from ∗(2) assms(3) have x = enc (w, Inr (positions-in-row x 0)
# I)
      apply (intro project-enc-extend [OF ∗(3)])
      apply (simp only: σ-def)
      apply auto
      done
    moreover from arg-cong[OF ∗(3), of length] have length w = length x by
simp
    ultimately show  map π x ∈ map π ‘
      (Z ∩ {enc (w, I') |w I'. length I' = length I + 1 ∧ (0 < length w ∧ (∀ a∈set
w. a ∈ set Σ) ∧
        Ball (set I') (case-sum (λp. p < length w) (λP. ∀ p∈P. p < length w))) ∧
          (∀ i∈X. case I' ! i of Inl x ⇒ True | Inr x ⇒ False) ∧
            (∀ i∈{0..<length I + 1} − X. case I' ! i of Inl x ⇒ False | Inr x ⇒
```

```
True)})
      by (intro imageI CollectI conjI IntI exI[of - w] exI[of - Inr (positions-in-row
x 0) # I])
        (auto simp: nth-Cons' positions-in-row elim!: bspec simp del: enc.simps)
  qed
qed

lemma map-project-ENC:
  assumes X ⊆ {0 ..< n + 1} Z ⊆ lists ((set o σ Σ) (n + 1))
  shows map π ` (Z ∩ lang (n + 1) (ENC (n + 1) X) − {[]}) =
    (if 0 ∈ X
    then map π ` (Z ∩ lang (n + 1) (ENC (n + 1) {0})) ∩ lang n (ENC n ((λx.
x − 1) ` (X − {0}))) − {[]}
    else map π ` Z ∩ lang n (ENC n ((λx. x − 1) ` (X − {0}))) − {[]})
    (is ?L = (if - then ?R1 else ?R2))
proof (split if-splits, intro conjI impI)
  assume 0: 0 ∉ X
  from assms have fin: finite X finite ((λx. x − 1) ` X)
    by (auto elim: finite-subset intro!: finite-imageI[of X])
  from 0 show ?L = ?R2 using map-project-Int-ENC[OF 0 assms]
    unfolding lists-image[symmetric] π-σ
      Int-absorb1[OF lang-subset-lists[OF wf-rexp-ENC[OF fin(1)]], of n + 1]
      Int-absorb1[OF lang-subset-lists[OF wf-rexp-ENC[OF fin(2)]], of n]
    by auto
next
  assume 0 ∈ X
  hence 0: 0 ∉ X − {0} and X: X = {0} ∪ (X − {0}) by auto
  from assms have fin: finite X
    by (auto elim: finite-subset intro!: finite-imageI[of X])
  have ?L = map π ` ((Z ∩ lang (n + 1) (ENC (n + 1) {0})) ∩ lang (n + 1)
(ENC (n + 1) (X − {0})) − {[]})
    unfolding Int-assoc using assms by (subst lang-ENC-split[OF fin X, of n +
1]) auto
  also have ... = ?R1
    using 0 assms by (elim map-project-Int-ENC) auto
  finally show ?L = ?R1 .
qed

abbreviation 𝔏 ≡ project.lang (set ∘ σ Σ) π

lemma lang_M2L-rexp-of'-rexp-of'':
  wf-formula n φ ⟹ lang n (rexp-of' n φ) − {[]} = lang n (rexp-of'' n φ) − {[]}
unfolding rexp-of'-def rexp-of''-def
proof (induction φ arbitrary: n)
  case (FNot φ)
  hence wf-formula n φ by simp
 with FNot.IH show ?case unfolding rexp-of-alt.simps rexp-of-alt'.simps lang.simps
ENC-Not by blast
next
```

case (*FAnd* φ_1 φ_2)
hence *wf1*: *wf-formula n* φ_1 **and** *wf2*: *wf-formula n* φ_2 **by** *force+*
from *FAnd.IH*(*1*)[*OF wf1*] *FAnd.IH*(*2*)[*OF wf2*] **show** *?case* **using** *ENC-And*[*OF FAnd.prems*]
unfolding *rexp-of-alt.simps rexp-of-alt'.simps lang.simps rexp-of-list.simps* **by** *blast*
next
case (*FOr* φ_1 φ_2)
hence *wf1*: *wf-formula n* φ_1 **and** *wf2*: *wf-formula n* φ_2 **by** *force+*
from *FOr.IH*(*1*)[*OF wf1*] *FOr.IH*(*2*)[*OF wf2*] **show** *?case* **using** *ENC-Or*[*OF FOr.prems*]
unfolding *rexp-of-alt.simps rexp-of-alt'.simps lang.simps rexp-of-list.simps* **by** *blast*
next
case (*FExists* φ)
hence *wf*: *wf-formula* $(n + 1)$ φ **and** *0*: $0 \in FOV\ \varphi$ **by** *auto*
then show *?case*
using *ENC-Exists*[*OF FExists.prems*] *map-project-ENC*[*of FOV* φ *n*] *max-idx-vars*[*of* $n + 1$ φ]
wf-rexp-of-alt'[*OF wf*] *0*
unfolding *rexp-of-alt.simps rexp-of-alt'.simps lang.simps Int-Diff-both*
unfolding *map-project-empty FExists.IH*[*OF wf*, *unfolded lang.simps*]
by (*intro trans*[*OF arg-cong2*[*of - - - -* $(\cap)$, *OF map-project-ENC*[*OF - lang-subset-lists*] *refl*]])
fastforce+
next
case (*FEXISTS* φ)
hence *wf*: *wf-formula* $(n + 1)$ φ **and** *0*: $0 \notin FOV\ \varphi$ **by** *auto*
then show *?case*
using *ENC-EXISTS*[*OF FEXISTS.prems*] *map-project-ENC*[*of FOV* φ *n*] *max-idx-vars*[*of* $n + 1$ φ]
wf-rexp-of-alt'[*OF wf*] *0*
unfolding *rexp-of-alt.simps rexp-of-alt'.simps lang.simps Int-Diff-both*
unfolding *map-project-empty FEXISTS.IH*[*OF wf*, *unfolded lang.simps*]
by (*intro trans*[*OF arg-cong2*[*of - - - -* $(\cap)$, *OF map-project-ENC*[*OF - lang-subset-lists*] *refl*]])
fastforce+
qed *simp-all*

theorem $lang_{M2L}$*-rexp-of'*: *wf-formula n* $\varphi \Longrightarrow lang_{M2L}\ n\ \varphi = lang\ n\ (rexp\text{-}of'\ n\ \varphi) - \{[]\}$
unfolding $lang_{M2L}$*-rexp-of-rexp-of'*[*symmetric*] **by** (*rule* $lang_{M2L}$*-rexp-of*)

theorem $lang_{M2L}$*-rexp-of''*: *wf-formula n* $\varphi \Longrightarrow lang_{M2L}\ n\ \varphi = lang\ n\ (rexp\text{-}of''\ n\ \varphi) - \{[]\}$
unfolding $lang_{M2L}$*-rexp-of'-rexp-of''*[*symmetric*] **by** (*rule* $lang_{M2L}$*-rexp-of'*)

end

11 Normalization of M2L Formulas

fun *nNot* **where**
  nNot (*FNot* φ) = φ
| *nNot* (*FAnd* $\varphi 1$ $\varphi 2$) = *FOr* (*nNot* $\varphi 1$) (*nNot* $\varphi 2$)
| *nNot* (*FOr* $\varphi 1$ $\varphi 2$) = *FAnd* (*nNot* $\varphi 1$) (*nNot* $\varphi 2$)
| *nNot* φ = *FNot* φ

primrec *norm* **where**
  norm (*FQ* *a* *m*) = *FQ* *a* *m*
| *norm* (*FLess* *m* *n*) = *FLess* *m* *n*
| *norm* (*FIn* *m* *M*) = *FIn* *m* *M*
| *norm* (*FOr* φ ψ) = *FOr* (*norm* φ) (*norm* ψ)
| *norm* (*FAnd* φ ψ) = *FAnd* (*norm* φ) (*norm* ψ)
| *norm* (*FNot* φ) = *nNot* (*norm* φ)
| *norm* (*FExists* φ) = *FExists* (*norm* φ)
| *norm* (*FEXISTS* φ) = *FEXISTS* (*norm* φ)

context *formula*
begin

lemma *satisfies-nNot*[*simp*]: *satisfies* (*w*, *I*) (*nNot* φ) = *satisfies* (*w*,*I*) (*FNot* φ)
  by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *FOV-nNot*[*simp*]: *FOV* (*nNot* φ) = *FOV* (*FNot* φ)
  by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *SOV-nNot*[*simp*]: *SOV* (*nNot* φ) = *SOV* (*FNot* φ)
  by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *pre-wf-formula-nNot*[*simp*]: *pre-wf-formula* *n* (*nNot* φ) = *pre-wf-formula* *n* (*FNot* φ)
  by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *FOV-norm*[*simp*]: *FOV* (*norm* φ) = *FOV* φ
  by (*induct* φ) *auto*

lemma *SOV-norm*[*simp*]: *SOV* (*norm* φ) = *SOV* φ
  by (*induct* φ) *auto*

lemma *pre-wf-formula-norm*[*simp*]: *pre-wf-formula* *n* (*norm* φ) = *pre-wf-formula* *n* φ
  by (*induct* φ *arbitrary*: *n*) *auto*

lemma *satisfies-norm*[*simp*]: *satisfies* (*w*, *I*) (*norm* φ) = *satisfies* (*w*, *I*) φ
  by (*induct* φ *arbitrary*: *I*) *auto*

lemma $lang_{M2L}$-*norm*[*simp*]: $lang_{M2L}$ *n* (*norm* φ) = $lang_{M2L}$ *n* φ

unfolding $lang_{M2L}$-*def* **by** *auto*

end

12 Deciding Equivalence of M2L Formulas

global-interpretation *embed set o* σ Σ *wf-atom* Σ π *lookup* ε Σ
for Σ :: $'a$ :: *linorder list*
defines
$\mathfrak{D}$ = *embed.lderiv lookup* (ε Σ)
and *Co*$\mathfrak{D}$ = *embed.lderiv-dual lookup* (ε Σ)
by *unfold-locales* (*auto simp*: σ-*def* π-*def* ε-*def set-n-lists*)

lemma *enum-not-empty*[*simp*]: *Enum.enum* $\neq$ [] (**is** *?enum* $\neq$ [])
proof (*rule notI*)
assume *?enum* = []
hence *set ?enum* = {} **by** *simp*
thus *False* **unfolding** *UNIV-enum*[*symmetric*] **by** *simp*
qed

global-interpretation Φ: *formula Enum.enum* :: $'a$ :: {*enum*, *linorder*} *list*
defines
pre-wf-formula = Φ.*pre-wf-formula*
and *wf-formula* = Φ.*wf-formula*
and *rexp-of* = Φ.*rexp-of*
and *rexp-of-alt* = Φ.*rexp-of-alt*
and *rexp-of-alt*$'$ = Φ.*rexp-of-alt*$'$
and *rexp-of*$'$ = Φ.*rexp-of*$'$
and *rexp-of*$''$ = Φ.*rexp-of*$''$
and *valid-ENC* = Φ.*valid-ENC*
and *ENC* = Φ.*ENC*
and *dec-interp* = Φ.*dec-interp*
by *unfold-locales* (*auto simp*: σ-*def* π-*def set-n-lists*)

lemma *lang-Plus-Zero*: *lang* Σ *n* (*Plus r One*) = *lang* Σ *n* (*Plus s One*) $\longleftrightarrow$ *lang* Σ *n r* $-$ {[]} = *lang* Σ *n s* $-$ {[]}
by *auto*

lemmas $lang_{M2L}$-*rexp-of-norm* = *trans*[*OF sym*[*OF* Φ.$lang_{M2L}$-*norm*] Φ.$lang_{M2L}$-*rexp-of*]
lemmas $lang_{M2L}$-*rexp-of*$'$-*norm* = *trans*[*OF sym*[*OF* Φ.$lang_{M2L}$-*norm*] Φ.$lang_{M2L}$-*rexp-of*$'$]
lemmas $lang_{M2L}$-*rexp-of*$''$-*norm* = *trans*[*OF sym*[*OF* Φ.$lang_{M2L}$-*norm*] Φ.$lang_{M2L}$-*rexp-of*$''$]

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path slow*)›

global-interpretation *D*: *rexp-DFA* σ Σ *wf-atom* Σ π *lookup* $\lambda x.$ «*pnorm* (*inorm x*)»
$\lambda a\ r.$ «$\mathfrak{D}$ Σ *a r*» *final alphabet.wf* (*wf-atom* Σ) *n pnorm lang* Σ *n n*

for Σ :: *'a* :: *linorder list* **and** *n* :: *nat*
defines
test = *rexp-DA.test* (*final* :: *'a atom rexp* ⇒ *bool*)
and *step* = *rexp-DA.step* (σ Σ) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *pnorm n*
and *closure* = *rexp-DA.closure* (σ Σ) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *final pnorm n*
and *check-eqvRE* = *rexp-DA.check-eqv* (σ Σ) ($\lambda x.$ ≪*pnorm* (*inorm x*)≫) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *final pnorm n*
and *test-invariant* = *rexp-DA.test-invariant* (*final* :: *'a atom rexp* ⇒ *bool*) ::
((*'a* × *bool list*) *list* × -) *list* × - ⇒ *bool*
and *step-invariant* = *rexp-DA.step-invariant* (σ Σ) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *pnorm n*
and *closure-invariant* = *rexp-DA.closure-invariant* (σ Σ) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *final pnorm n*
and *counterexampleRE* = *rexp-DA.counterexample* (σ Σ) ($\lambda x.$ ≪*pnorm* (*inorm x*)≫) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *final pnorm n*
and *reachable* = *rexp-DA.reachable* (σ Σ) ($\lambda x.$ ≪*pnorm* (*inorm x*)≫) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *pnorm n*
and *automaton* = *rexp-DA.automaton* (σ Σ) ($\lambda x.$ ≪*pnorm* (*inorm x*)≫) ($\lambda a\ r.$ ≪$\mathfrak{D}$ Σ *a r*≫) *pnorm n*
by *unfold-locales* (*auto simp only*: *comp-apply*
ACI-norm-wf ACI-norm-lang wf-inorm lang-inorm wf-pnorm lang-pnorm wf-lderiv lang-lderiv
lang-final finite-fold-lderiv dest!: *lang-subset-lists*)

definition *check-eqv* **where**
check-eqv n φ ψ ⟷ *wf-formula n* (*FOr* φ ψ) ∧
slow.check-eqvRE Enum.enum n (*Plus* (*rexp-of'' n* (*norm* φ)) *One*) (*Plus* (*rexp-of'' n* (*norm* ψ)) *One*)

definition *counterexample* **where**
counterexample n φ ψ =
map-option ($\lambda w.$ *dec-interp n* (*FOV* (*FOr* φ ψ)) *w*)
(*slow.counterexampleRE Enum.enum n* (*Plus* (*rexp-of'' n* (*norm* φ)) *One*) (*Plus* (*rexp-of'' n* (*norm* ψ)) *One*))

lemma *soundness*: *slow.check-eqv n* φ ψ ⟹ Φ.$lang_{M2L}$ *n* φ = Φ.$lang_{M2L}$ *n* ψ
by (*rule box-equals*[*OF iffD1*[*OF lang-Plus-Zero*, *OF slow.D.check-eqv-sound*]
sym[*OF trans*[*OF* $lang_{M2L}$*-rexp-of''-norm*]] *sym*[*OF trans*[*OF* $lang_{M2L}$*-rexp-of''-norm*]]])
(*auto simp*: *slow.check-eqv-def intro*!: Φ.*wf-rexp-of''*)

lemma *completeness*:
assumes Φ.$lang_{M2L}$ *n* φ = Φ.$lang_{M2L}$ *n* ψ *wf-formula n* (*FOr* φ ψ)
shows *slow.check-eqv n* φ ψ
using *assms*(*2*) **unfolding** *slow.check-eqv-def*
by (*intro conjI*[*OF assms*(*2*) *slow.D.check-eqv-complete*[*OF iffD2*[*OF lang-Plus-Zero*]],
OF box-equals[*OF assms*(*1*) $lang_{M2L}$*-rexp-of''-norm* $lang_{M2L}$*-rexp-of''-norm*]])
(*auto intro*!: Φ.*wf-rexp-of''*)

setup ‹*Sign.map-naming Name-Space.parent-path*›

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path fast*)›

global-interpretation *D*: *rexp-DA-no-post* σ Σ *wf-atom* Σ π *lookup* $\lambda x.$ *pnorm* (*inorm x*)
$\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*) *final alphabet.wf* (*wf-atom* Σ) *n lang* Σ *n n*
for Σ :: *'a* :: *linorder list* **and** *n* :: *nat*
defines
test = *rexp-DA.test* (*final* :: *'a atom rexp* ⇒ *bool*)
and *step* = *rexp-DA.step* (σ Σ) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *id n*
and *closure* = *rexp-DA.closure* (σ Σ) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *final id n*
and *check-eqvRE* = *rexp-DA.check-eqv* (σ Σ) ($\lambda x.$ *pnorm* (*inorm x*)) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *final id n*
and *test-invariant* = *rexp-DA.test-invariant* (*final* :: *'a atom rexp* ⇒ *bool*) ::
(('*a* × *bool list*) *list* × -) *list* × - ⇒ *bool*
and *step-invariant* = *rexp-DA.step-invariant* (σ Σ) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *id n*
and *closure-invariant* = *rexp-DA.closure-invariant* (σ Σ) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *final id n*
and *counterexampleRE* = *rexp-DA.counterexample* (σ Σ) ($\lambda x.$ *pnorm* (*inorm x*)) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *final id n*
and *reachable* = *rexp-DA.reachable* (σ Σ) ($\lambda x.$ *pnorm* (*inorm x*)) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *id n*
and *automaton* = *rexp-DA.automaton* (σ Σ) ($\lambda x.$ *pnorm* (*inorm x*)) ($\lambda a\ r.$ *pnorm* ($\mathfrak{D}$ Σ *a r*)) *id n*
by *unfold-locales* (*auto simp only*: *comp-apply*
ACI-norm-wf ACI-norm-lang wf-inorm lang-inorm wf-pnorm lang-pnorm wf-lderiv lang-lderiv id-apply
lang-final dest!: *lang-subset-lists*)

definition *check-eqv* **where**
check-eqv n φ ψ ⟷ *wf-formula n* (*FOr* φ ψ) ∧
fast.check-eqvRE Enum.enum n (*Plus* (*rexp-of'' n* (*norm* φ)) *One*) (*Plus* (*rexp-of'' n* (*norm* ψ)) *One*)

definition *counterexample* **where**
counterexample n φ ψ =
map-option ($\lambda w.$ *dec-interp n* (*FOV* (*FOr* φ ψ)) *w*)
(*fast.counterexampleRE Enum.enum n* (*Plus* (*rexp-of'' n* (*norm* φ)) *One*) (*Plus* (*rexp-of'' n* (*norm* ψ)) *One*))

lemma *soundness*: *fast.check-eqv n* φ ψ ⟹ Φ.$lang_{M2L}$ *n* φ = Φ.$lang_{M2L}$ *n* ψ
by (*rule box-equals*[*OF iffD1*[*OF lang-Plus-Zero*, *OF fast.D.check-eqv-sound*]
sym[*OF trans*[*OF* $lang_{M2L}$*-rexp-of''-norm*]] *sym*[*OF trans*[*OF* $lang_{M2L}$*-rexp-of''-norm*]]])
(*auto simp*: *fast.check-eqv-def intro*!: Φ.*wf-rexp-of''*)

setup ‹*Sign.map-naming Name-Space.parent-path*›

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path dual*)›

global-interpretation *D*: *rexp-DA-no-post* σ Σ *wf-atom* Σ π *lookup*
λx. *pnorm-dual* (*rexp-dual-of* (*inorm x*)) $\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*) *final-dual*
alphabet.wf-dual (*wf-atom* Σ) *n lang-dual* Σ *n n*
for Σ :: *'a* :: *linorder list* **and** *n* :: *nat*
defines
test = *rexp-DA.test* (*final-dual* :: *'a atom rexp-dual* $\Rightarrow$ *bool*)
and *step* = *rexp-DA.step* (σ Σ) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *id n*
and *closure* = *rexp-DA.closure* (σ Σ) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *final-dual id n*
and *check-eqvRE* = *rexp-DA.check-eqv* (σ Σ) (λx. *pnorm-dual* (*rexp-dual-of* (*inorm x*))) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *final-dual id n*
and *test-invariant* = *rexp-DA.test-invariant* (*final-dual* :: *'a atom rexp-dual* $\Rightarrow$ *bool*) ::
(('*a* $\times$ *bool list*) *list* $\times$ -) *list* $\times$ - $\Rightarrow$ *bool*
and *step-invariant* = *rexp-DA.step-invariant* (σ Σ) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *id n*
and *closure-invariant* = *rexp-DA.closure-invariant* (σ Σ) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *final-dual id n*
and *counterexampleRE* = *rexp-DA.counterexample* (σ Σ) (λx. *pnorm-dual* (*rexp-dual-of* (*inorm x*))) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *final-dual id n*
and *reachable* = *rexp-DA.reachable* (σ Σ) (λx. *pnorm-dual* (*rexp-dual-of* (*inorm x*))) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *id n*
and *automaton* = *rexp-DA.automaton* (σ Σ) (λx. *pnorm-dual* (*rexp-dual-of* (*inorm x*))) ($\lambda a\ r$. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ *a r*)) *id n*
by *unfold-locales* (*auto simp only*: *comp-apply id-apply*
wf-inorm lang-inorm
wf-dual-pnorm-dual lang-dual-pnorm-dual
wf-dual-rexp-dual-of lang-dual-rexp-dual-of
wf-dual-lderiv-dual lang-dual-lderiv-dual
lang-dual-final-dual dest!: *lang-dual-subset-lists*)

definition *check-eqv* **where**
check-eqv n φ ψ $\longleftrightarrow$ *wf-formula n* (*FOr* φ ψ) $\wedge$
dual.check-eqvRE Enum.enum n (*Plus* (*rexp-of'' n* (*norm* φ)) *One*) (*Plus* (*rexp-of'' n* (*norm* ψ)) *One*)

definition *counterexample* **where**
counterexample n φ ψ =
map-option (λw. *dec-interp n* (*FOV* (*FOr* φ ψ)) *w*)
(*dual.counterexampleRE Enum.enum n* (*Plus* (*rexp-of'' n* (*norm* φ)) *One*) (*Plus* (*rexp-of'' n* (*norm* ψ)) *One*))

lemma *soundness*: *dual.check-eqv n* φ ψ $\Longrightarrow$ Φ.*lang*$_{M2L}$ *n* φ = Φ.*lang*$_{M2L}$ *n* ψ
by (*rule box-equals*[*OF iffD1*[*OF lang-Plus-Zero*, *OF dual.D.check-eqv-sound*]
sym[*OF trans*[*OF lang*$_{M2L}$*-rexp-of''-norm*]] *sym*[*OF trans*[*OF lang*$_{M2L}$*-rexp-of''-norm*]]])
(*auto simp*: *dual.check-eqv-def intro*!: Φ.*wf-rexp-of''*)

setup ‹*Sign.map-naming Name-Space.parent-path*›

13 WS1S

13.1 Encodings

definition *cut-same x s = stake* (*LEAST n. sdrop n s = sconst x*) *s*

abbreviation *poss I* ≡ (⋃*x*∈*set I*. *case x of Inl p* ⇒ {*p*} | *Inr P* ⇒ *P*)

declare *smap-sconst*[*simp*]

lemma (**in** *wellorder*) *min-Least*:
⟦∃ *n*. *P n*; ∃ *n*. *Q n*⟧ ⟹ *min* (*Least P*) (*Least Q*) = (*LEAST n*. *P n* ∨ *Q n*)
proof (*intro sym*[*OF Least-equality*])
fix *y* **assume** *P y* ∨ *Q y*
thus *min* (*Least P*) (*Least Q*) ≤ *y*
proof (*elim disjE*)
assume *P y*
hence *Least P* ≤ *y* **by** (*auto intro*: *LeastI2-wellorder*)
thus *min* (*Least P*) (*Least Q*) ≤ *y* **unfolding** *min-def* **by** *auto*
next
assume *Q y*
hence *Least Q* ≤ *y* **by** (*auto intro*: *LeastI2-wellorder*)
thus *min* (*Least P*) (*Least Q*) ≤ *y* **unfolding** *min-def* **by** *auto*
qed
qed (*metis LeastI-ex min-def*)

lemma *sconst-collapse*: *y* ## *sconst y* = *sconst y*
by (*subst* (*2*) *siterate.ctr*) *auto*

lemma *shift-sconst-inj*: ⟦*length x* = *length y*; *x* @− *sconst z* = *y* @− *sconst z*⟧
⟹ *x* = *y*
by (*induct rule*: *list-induct2*) *auto*

context *formula*
begin

definition *any* ≡ *hd* Σ

lemma *any-*Σ[*simp*]: *any* ∈ *set* Σ
unfolding *any-def* **by** (*auto simp*: *nonempty intro*: *someI*[*of - hd* Σ])

lemma *any-σ*[*simp*]: *length bs* = *n* ⟹ (*any*, *bs*) ∈ *set* (*σ* Σ *n*)
by (*auto simp*: *σ-def set-n-lists*)

fun *stream-enc* :: *'a interp* ⇒ (*'a* × *bool list*) *stream* **where**
stream-enc (*w*, *I*) = *smap2* (*enc-atom I*) *nats* (*w* @− *sconst any*)

lemma *tl-stream-enc*[*simp*]: *smap π* (*stream-enc* (*w*, *x* # *I*)) = *stream-enc* (*w*, *I*)
by (*auto simp*: *comp-def π-def*)

lemma *enc-atom-max*: ⟦∀ *x*∈*set I*. *case x of Inl p* ⇒ *p* ≤ *n* | *Inr P* ⇒ ∀ *p*∈*P*. *p* ≤ *n*; *n* ≤ *n'*⟧ ⟹
enc-atom I (*Suc n'*) *a* = (*a*, *replicate* (*length I*) *False*)
by (*induct I*) (*auto split*: *sum.splits*)

lemma *ex-Loop-stream-enc*:
assumes ∀ *x* ∈ *set I*. *case x of Inr P* ⇒ *finite P* | - ⇒ *True*
shows ∃ *n*. *sdrop n* (*stream-enc* (*w*, *I*)) = *sconst* (*any*, *replicate* (*length I*) *False*)
proof −
from *assms* **have** ∃ *n* > *length w*. ∀ *x*∈*set I*. *case x of Inl p* ⇒ *p* ≤ *n* | *Inr P* ⇒ ∀ *p*∈*P*. *p* ≤ *n*
proof (*induct I*)
case (*Cons x I*)
then obtain *n* **where** *IH*: *length w* < *n*
∀ *x*∈*set I*. *case x of Inl p* ⇒ *p* ≤ *n* | *Inr P* ⇒ ∀ *p*∈*P*. *p* ≤ *n* **by** *auto*
thus *?case*
proof (*cases x*)
case (*Inl p*)
with *IH* **show** *?thesis*
by (*intro exI*[*of* - *max p n*]) (*fastforce split*: *sum.splits*)
next
case (*Inr P*)
with *IH Cons*(*2*) **show** *?thesis*
by (*intro exI*[*of* - *max* (*Max P*) *n*]) (*fastforce dest*: *Max-ge split*: *sum.splits*)
qed
qed *auto*
then obtain *n* **where** *length w* < *n* ∀ *x*∈*set I*. *case x of Inl p* ⇒ *p* ≤ *n* | *Inr P* ⇒ ∀ *p*∈*P*. *p* ≤ *n*
by (*elim exE conjE*)
hence *sdrop* (*Suc n*) (*stream-enc* (*w*, *I*)) = *sconst* (*any*, *replicate* (*length I*) *False*)
(**is** *?s1 n* = *?s2*)
by (*intro stream.coinduct*[*of* λ*s1 s2*. ∃ *n'*≥ *n*. *s1* = *?s1 n'* ∧ *s2* = *?s2*])
(*auto simp*: *enc-atom-max dest*: *le-SucI*)
thus *?thesis* **by** *blast*
qed

lemma *length-snth-enc*[*simp*]: *length* (*snd* (*stream-enc* (*w*, *I*) !! *n*)) = *length I*
by *auto*

lemma *sset-singleton*[*simp*]: *sset s* ⊆ {*x*} ⟷ *sset s* = {*x*}
by (*cases s*) *auto*

lemma *drop-sconstE*: ⟦*drop n w* @− *sconst y* = *sconst y*; *p* < *length w*; ¬ *p* < *n*⟧ ⟹ *w* ! *p* = *y*
unfolding *not-less sconst-alt* **proof** (*induct p arbitrary*: *w n*)
case (*Suc p*)
with *Suc*(*1*)[*of 0 tl w*] **show** *?case*
by (*cases w n rule*: *list.exhaust*[*case-product nat.exhaust*]) *auto*

qed (*auto simp add*: *neq-Nil-conv*)

lemma *less-length-cut-same*:
⟦(*w* @− *sconst y*) !! *p* = *a*⟧ ⟹ *a* = *y* ∨ (*p* < *length* (*cut-same y* (*w* @− *sconst y*)) ∧ *w* ! *p* = *a*)
unfolding *cut-same-def length-stake*
by (*rule LeastI2-ex*[*OF exI*[*of* - *length w*]])
(*auto simp*: *sdrop-shift shift-snth split*: *if-split-asm elim*!: *drop-sconstE*)

lemma *less-length-cut-same-Inl*:
⟦(∀ *x* ∈ *set I*. *case x of Inr P* ⇒ *finite P* | - ⇒ *True*); *r* < *length I*; *I* ! *r* = *Inl p*⟧ ⟹
p < *length* (*cut-same* (*any*, *replicate* (*length I*) *False*) (*stream-enc* (*w*, *I*)))
unfolding *cut-same-def length-stake*
by (*erule LeastI2-ex*[*OF ex-Loop-stream-enc ccontr*],
auto simp: *smap2-alt list-eq-iff-nth-eq add.commute dest*!: *add-diff-inverse split*: *sum.splits*,
metis)

lemma *less-length-cut-same-Inr*:
⟦(∀ *x* ∈ *set I*. *case x of Inr P* ⇒ *finite P* | - ⇒ *True*); *r* < *length I*; *I* ! *r* = *Inr P*⟧ ⟹
∀ *p* ∈ *P*. *p* < *length* (*cut-same* (*any*, *replicate* (*length I*) *False*) (*stream-enc* (*w*, *I*)))
unfolding *cut-same-def length-stake*
by (*rule ballI*, *erule LeastI2-ex*[*OF ex-Loop-stream-enc ccontr*],
auto simp: *smap2-alt list-eq-iff-nth-eq add.commute dest*!: *add-diff-inverse split*: *sum.splits*,
metis)

fun *enc* :: *'a interp* ⇒ (*'a* × *bool list*) *list set* **where**
enc (*w*, *I*) = {*x*. ∃ *n*. *x* = (*cut-same* (*any*, *replicate* (*length I*) *False*) (*stream-enc* (*w*, *I*)) @
replicate n (*any*, *replicate* (*length I*) *False*))}

lemma *cut-same-all*[*simp*]: *cut-same x* (*sconst x*) = []
unfolding *cut-same-def* **by** (*auto intro*: *Least-equality*)

lemma *cut-same-stop*[*simp*]:
assumes *x* ≠ *y*
shows *cut-same x* (*xs* @− *y* ## *sconst x*) = *xs* @ [*y*] (**is** *cut-same x ?s* = -)
proof −
have (*LEAST n*. *sdrop n ?s* = *sconst x*) = *Suc* (*length xs*)
proof (*rule Least-equality*)
show *sdrop* (*Suc* (*length xs*)) *?s* = *sconst x* **by** (*auto simp*: *sdrop-shift*)
next
fix *m* **assume** ∗: *sdrop m ?s* = *sconst x*
{ assume *m* < *Suc* (*length xs*)
hence *m* ≤ *length xs* **by** *simp*

```
    then obtain ys where sdrop m ?s = ys @− y ## sconst x
      by atomize-elim (induct m arbitrary: xs, auto)
    with ∗ obtain ys @− y ## sconst x = sconst x by simp
    from arg-cong[OF this, of sdrop (length ys)] have y ## sconst x = sconst x
      by (auto simp: sdrop-shift)
    with assms have False by (metis siterate.code stream.inject)
   }
   thus Suc (length xs) ≤ m by (blast intro: leI)
  qed
  thus ?thesis unfolding cut-same-def stake-shift by simp
qed

lemma cut-same-shift-sconst: ∃ n. w = cut-same x (w @− sconst x) @ replicate n
x
proof (induct w rule: rev-induct)
  case (snoc a w)
  then obtain n where w = cut-same x (w @− sconst x) @ replicate n x by blast
  thus ?case
    by (cases a = x)
     (auto simp: id-def[symmetric] siterate.code[of id, simplified id-apply, symmet-
ric]
       replicate-append-same[symmetric] intro!: exI[of - Suc n])
qed (simp add: id-def[symmetric])

lemma set-cut-same: set (cut-same x (w @− sconst x)) ⊆ set w
proof (induct w rule: rev-induct)
  case (snoc a w)
  thus ?case by (cases a = x)
    (auto simp: id-def[symmetric] siterate.code[of id, simplified id-apply, symmet-
ric])
qed (simp add: id-def[symmetric])

lemma stream-enc-cut-same:
  assumes (∀ x ∈ set I. case x of Inr P ⇒ finite P | - ⇒ True)
  shows stream-enc (w, I) = cut-same (any, replicate (length I) False) (stream-enc
(w, I)) @−
    sconst (any, replicate (length I) False)
  unfolding cut-same-def
  by (rule trans[OF sym[OF stake-sdrop] arg-cong2[of - - - - (@−), OF refl]])
     (rule LeastI-ex[OF ex-Loop-stream-enc[OF assms]])

lemma stream-enc-enc:
  assumes (∀ x ∈ set I. case x of Inr P ⇒ finite P | - ⇒ True) and  v: v ∈ enc
(w, I)
  shows stream-enc (w, I) = v @− sconst (any, replicate (length I) False)
  (is ?s = ?v @− sconst ?F)
proof −
  from assms(1) obtain n where sdrop n (stream-enc (w, I)) = sconst ?F by
(metis ex-Loop-stream-enc)
```

moreover from v **obtain** m **where** *?v = cut-same ?F ?s @ replicate m ?F* **by** *auto*
ultimately show *?s = v @− sconst ?F*
by (*auto simp del*: *stream-enc.simps intro*: *stream-enc-cut-same*[*OF assms*(*1*)])
qed

lemma *stream-enc-enc-some*:
assumes (∀ *x* ∈ *set I*. *case x of Inr P* ⇒ *finite P* | - ⇒ *True*)
shows *stream-enc* (*w*, *I*) = (*SOME v*. *v* ∈ *enc* (*w*, *I*)) @− *sconst* (*any*, *replicate* (*length I*) *False*)
by (*rule stream-enc-enc*[*OF assms*], *rule someI-ex*) *auto*

lemma *enc-unique-length*: *v* ∈ *enc* (*w*, *I*) ⟹ ∀ *v'*. *length v'* = *length v* ∧ *v'* ∈ *enc* (*w*, *I*) ⟶ *v* = *v'*
by *auto*

lemma *sdrop-sconst*: *sdrop n s* = *sconst x* ⟹ *n* ≤ *m* ⟹ *s* !! *m* = *x*
by (*metis le-iff-add sdrop-snth snth-siterate*[*of id*, *simplified id-funpow id-apply*])

lemma *fin-cut-same-tl*:
assumes ∃ *n*. *sdrop n s* = *sconst x*
shows *fin-cut-same* (π *x*) (*map* π (*cut-same x s*)) = *cut-same* (π *x*) (*smap* π *s*)
proof −
define *min* **where** *min* = (*LEAST n*. *sdrop n s* = *sconst x*)
from *assms* **have** *min*: *sdrop min s* = *sconst x* ⋀*m*. *sdrop m s* = *sconst x* ⟹ *min* ≤ *m*
unfolding *min-def* **by** (*auto intro*: *LeastI Least-le*)
have *Ex*: ∃ *n*. *drop n* (*map* π (*stake min s*)) = *replicate* (*length* (*map* π (*stake min s*)) − *n*) (π *x*)
by (*auto intro*: *exI*[*of - length* (*map* π (*stake min s*))])
have *fin-cut-same* (π *x*) (*map* π (*cut-same x s*)) =
map π (*stake* (*LEAST n*.
map π (*stake* (*min* − *n*) (*sdrop n s*)) = *replicate* (*min* − *n*) (π *x*) ∨ *sdrop n s* = *sconst x*) *s*)
unfolding *fin-cut-same-def cut-same-def take-map take-stake min-Least*[*OF Ex assms*, *folded min-def*]
min-def[*symmetric*] **by** (*auto simp*: *drop-map drop-stake*)
also have (λ*n*. *map* π (*stake* (*min* − *n*) (*sdrop n s*)) = *replicate* (*min* − *n*) (π *x*) ∨ *sdrop n s* = *sconst x*) =
(λ*n*. *smap* π (*sdrop n s*) = *sconst* (π *x*))
proof (*rule ext*, *unfold smap-alt snth-siterate*[*of id*, *simplified id-funpow id-apply*], *safe*)
fix *n m*
assume *map* π (*stake* (*min* − *n*) (*sdrop n s*)) = *replicate* (*min* − *n*) (π *x*)
hence ∀ *y*∈*set* (*stake* (*min* − *n*) (*sdrop n s*)). π *y* = π *x*
by (*intro iffD1*[*OF map-eq-conv*]) (*metis length-stake map-replicate-const*)
hence ∀ *i*<*min* − *n*. π (*sdrop n s* !! *i*) = π *x*
unfolding *all-set-conv-all-nth* **by** (*auto simp*: *sdrop-snth*)
thus π (*sdrop n s* !! *m*) = π *x*

```
    proof (cases m < min − n)
      case False
      hence min ≤ n + m by linarith
       hence sdrop n s !! m = x unfolding sdrop-snth by (rule sdrop-sconst[OF
min(1)])
      thus ?thesis by simp
    qed auto
  next
    fix n
    assume ∀ m. π (sdrop n s !! m) = π x
    thus map π (stake (min − n) (sdrop n s)) = replicate (min − n) (π x)
      unfolding stake-smap[symmetric] smap-alt[symmetric, of π sdrop n s sconst
(π x), simplified]
        by (auto simp: map-replicate-const)
  qed auto
  finally show ?thesis unfolding cut-same-def sdrop-smap stake-smap .
qed

lemma tl-enc[simp]:
  assumes ∀ x ∈ set (x # I). case x of Inr P ⇒ finite P | - ⇒ True
  shows SAMEQUOT (any, replicate (length I) False) (map π ‘ enc (w, x # I))
= enc (w, I)
  unfolding SAMEQUOT-def
  by (fastforce simp: assms π-def
     fin-cut-same-tl[OF ex-Loop-stream-enc[OF assms], unfolded π-def, simplified,
symmetric])

lemma encD:
  ⟦v ∈ enc (w, I); (∀ x ∈ set I. case x of Inr P ⇒ finite P | - ⇒ True)⟧ ⟹
  v = map (case-prod (enc-atom I)) (zip [0 ..< length v] (stake (length v) (w @−
sconst any)))
  by (erule box-equals[OF sym[OF arg-cong[of - - stake (length v) ,OF stream-enc-enc]]])
    (auto simp: stake-shift sdrop-shift stake-add[symmetric] map-replicate-const)

lemma enc-Inl: ⟦x ∈ enc (w, I); (∀ x ∈ set I. case x of Inr P ⇒ finite P | - ⇒
True);
  m < length I; I ! m = Inl p⟧ ⟹ p < length x ∧ snd (x ! p) ! m
  by (auto dest!: less-length-cut-same-Inl[of - - - w] simp: nth-append cut-same-def)

lemma enc-Inr: assumes x ∈ enc (w, I) ∀ x ∈ set I. case x of Inr P ⇒ finite P
| - ⇒ True
  M < length I I ! M = Inr P
  shows p ∈ P ⟷ p < length x ∧ snd (x ! p) ! M
proof
  assume p ∈ P with assms show p < length x ∧ snd (x ! p) ! M
    by (auto dest!: less-length-cut-same-Inr[of - - - w] simp: nth-append cut-same-def)
next
  assume p < length x ∧ snd (x ! p) ! M
  thus p ∈ P using assms by (subst (asm) (2) encD[OF assms(1,2)]) auto
```

qed

lemma *enc-length*:
assumes *enc* (*w*, *I*) = *enc* (*w'*, *I'*)
shows *length I* = *length I'*
proof −
let *?cL* = λ*w I*. *cut-same* (*any*, *replicate* (*length I*) *False*) (*stream-enc* (*w*, *I*))
let *?w* = λ*w I m*. *?cL w I* @ *replicate* (*m* − *length* (*?cL w I*)) (*any*, *replicate* (*length I*) *False*)
let *?max* = *max* (*length* (*?cL w I*)) (*length* (*?cL w' I'*)) + *1*
from *assms* **have** *?w w I ?max* ∈ *enc* (*w*, *I*) *?w w' I' ?max* ∈ *enc* (*w'*, *I'*) **by** *auto*
hence *?w w I ?max* = *?w w' I' ?max* **using** *enc-unique-length assms* **by** (*simp del*: *enc.simps*)
moreover have *last* (*?w w I ?max*) = (*any*, *replicate* (*length I*) *False*)
last (*?w w' I' ?max*) = (*any*, *replicate* (*length I'*) *False*) **by** *auto*
ultimately show *length I* = *length I'* **by** *auto*
qed

lemma *enc-stream-enc*:
⟦(∀ *x* ∈ *set I*. *case x of Inr P* ⇒ *finite P* | - ⇒ *True*);
(∀ *x* ∈ *set I'*. *case x of Inr P* ⇒ *finite P* | - ⇒ *True*);
enc (*w*, *I*) = *enc* (*w'*, *I'*)⟧ ⟹ *stream-enc* (*w*, *I*) = *stream-enc* (*w'*, *I'*)
by (*rule box-equals*[*OF - sym*[*OF stream-enc-enc-some*] *sym*[*OF stream-enc-enc-some*]])
(*auto dest*: *enc-length simp del*: *enc.simps*)

abbreviation *wf-interp w I* ≡
((∀ *a* ∈ *set w*. *a* ∈ *set* Σ) ∧ (∀ *x* ∈ *set I*. *case x of Inr P* ⇒ *finite P* | - ⇒ *True*))

fun *wf-interp-for-formula* :: *'a interp* ⇒ *'a formula* ⇒ *bool* **where**
wf-interp-for-formula (*w*, *I*) φ =
(*wf-interp w I* ∧
(∀ *n* ∈ *FOV* φ. *case I* ! *n of Inl* - ⇒ *True* | - ⇒ *False*) ∧
(∀ *n* ∈ *SOV* φ. *case I* ! *n of Inl* - ⇒ *False* | *Inr* - ⇒ *True*))

fun *satisfies* :: *'a interp* ⇒ *'a formula* ⇒ *bool* (**infix** ⊨ *50*) **where**
(*w*, *I*) ⊨ *FQ a m* = ((*case I* ! *m of Inl p* ⇒ *if p* < *length w then w* ! *p else any*) = *a*)
| (*w*, *I*) ⊨ *FLess m1 m2* = ((*case I* ! *m1 of Inl p* ⇒ *p*) < (*case I* ! *m2 of Inl p* ⇒ *p*))
| (*w*, *I*) ⊨ *FIn m M* = ((*case I* ! *m of Inl p* ⇒ *p*) ∈ (*case I* ! *M of Inr P* ⇒ *P*))
| (*w*, *I*) ⊨ *FNot* φ = (¬ (*w*, *I*) ⊨ φ)
| (*w*, *I*) ⊨ (*FOr* φ$_1$ φ$_2$) = ((*w*, *I*) ⊨ φ$_1$ ∨ (*w*, *I*) ⊨ φ$_2$)
| (*w*, *I*) ⊨ (*FAnd* φ$_1$ φ$_2$) = ((*w*, *I*) ⊨ φ$_1$ ∧ (*w*, *I*) ⊨ φ$_2$)
| (*w*, *I*) ⊨ (*FExists* φ) = (∃ *p*. (*w*, *Inl p* # *I*) ⊨ φ)
| (*w*, *I*) ⊨ (*FEXISTS* φ) = (∃ *P*. *finite P* ∧ (*w*, *Inr P* # *I*) ⊨ φ)

definition *lang$_{WS1S}$* :: *nat* ⇒ *'a formula* ⇒ (*'a* × *bool list*) *list set* **where**
lang$_{WS1S}$ n φ = ⋃{*enc* (*w*, *I*) | *w I* . *length I* = *n* ∧ *wf-interp-for-formula* (*w*,

$I)\ \varphi \wedge (w,\ I) \models \varphi\}$

lemma *encD-ex*: $[\![x \in enc\ (w,\ I);\ (\forall x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True)]\!] \Longrightarrow$
$\exists n.\ x = map\ (case\text{-}prod\ (enc\text{-}atom\ I))\ (zip\ [0\ ..<\ n]\ (stake\ n\ (w\ @-\ sconst\ any)))$
by (*auto dest*!: *encD simp del*: *enc.simps*)

lemma *enc-set-σ*: $[\![x \in enc\ (w,\ I);\ (\forall x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True);$
$length\ I = n;\ a \in set\ x;\ set\ w \subseteq set\ \Sigma]\!] \Longrightarrow a \in set\ (\sigma\ \Sigma\ n)$
by (*force dest*: *encD-ex intro*: *enc-atom-σ simp*: *in-set-zip shift-snth simp del*: *enc.simps*)

definition *positions-in-row s i* =
Option.these (*sset* (*smap2* (λ*p* (-, *bs*). *if nth bs i then Some p else None*) *nats s*))

lemma *positions-in-row*: *positions-in-row s i* = {*p*. *snd* (*s* !! *p*) ! *i*}
unfolding *positions-in-row-def Option.these-def smap2-szip stream.set-map sset-range*
by (*auto split*: *if-split-asm intro*!: *image-eqI*[*of* - *the*] *split*: *prod.splits*)

lemma *positions-in-row-unique*: ∃!*p*. *snd* (*s* !! *p*) ! *i* ⟹
the-elem (*positions-in-row s i*) = (*THE p*. *snd* (*s* !! *p*) ! *i*)
by (*rule the1I2*) (*auto simp*: *the-elem-def positions-in-row*)

lemma *positions-in-row-nth*: ∃!*p*. *snd* (*s* !! *p*) ! *i* ⟹
snd (*s* !! *the-elem* (*positions-in-row s i*)) ! *i*
unfolding *positions-in-row-unique* **by** (*rule the1I2*) *auto*

definition *dec-word s* = *cut-same any* (*smap fst s*)

lemma *dec-word-stream-enc*: *dec-word* (*stream-enc* (*w*, *I*)) = *cut-same any* (*w* @− *sconst any*)
unfolding *dec-word-def* **by** (*auto intro*!: *arg-cong*[*of* - - *cut-same any*] *simp*: *smap2-alt*)

definition *stream-dec n FO* (*s* :: ('*a* × *bool list*) *stream*) = *map* (λ*i*.
if i ∈ *FO*
then Inl (*the-elem* (*positions-in-row s i*))
else Inr (*positions-in-row s i*)) [*0*..<*n*]

lemma *stream-dec-Inl*: $[\![i \in FO;\ i < n]\!] \Longrightarrow \exists p.\ stream\text{-}dec\ n\ FO\ s\ !\ i = Inl\ p$
unfolding *stream-dec-def* **using** *nth-map*[*of n* [*0*..<*n*]] **by** *auto*

lemma *stream-dec-not-Inr*: $[\![stream\text{-}dec\ n\ FO\ s\ !\ i = Inr\ P;\ i \in FO;\ i < n]\!] \Longrightarrow$ *False*
unfolding *stream-dec-def* **using** *nth-map*[*of n* [*0*..<*n*]] **by** *auto*

lemma *stream-dec-Inr*: $[\![i \notin FO;\ i < n]\!] \Longrightarrow \exists P.\ stream\text{-}dec\ n\ FO\ s\ !\ i = Inr\ P$
unfolding *stream-dec-def* **using** *nth-map*[*of n* [*0..<n*]] **by** *auto*

lemma *stream-dec-not-Inl*: $[\![stream\text{-}dec\ n\ FO\ s\ !\ i = Inl\ p;\ i \notin FO;\ i < n]\!] \Longrightarrow$ *False*
unfolding *stream-dec-def* **using** *nth-map*[*of n* [*0..<n*]] **by** *auto*

lemma *Inr-dec-finite*: $[\![\forall i{<}n.\ finite\ \{p.\ snd\ (s\ !!\ p)\ !\ i\};\ Inr\ P \in set\ (stream\text{-}dec\ n\ FO\ s)]\!] \Longrightarrow$
finite P
unfolding *stream-dec-def* **by** (*auto simp*: *positions-in-row*)

lemma *enc-atom-dec*:
$[\![\forall p.\ length\ (snd\ (s\ !!\ p)) = n;\ \forall i \in FO.\ i < n \longrightarrow (\exists!p.\ snd\ (s\ !!\ p)\ !\ i);\ a = fst\ (s\ !!\ p)]\!] \Longrightarrow$
enc-atom (*stream-dec n FO s*) *p a* = *s* !! *p*
unfolding *stream-dec-def*
by (*rule sym*, *subst surjective-pairing*[*of s* !! *p*])
(*auto intro*!: *nth-equalityI simp*: *positions-in-row simp del*: *prod.collapse split*: *if-split-asm*,
(*metis positions-in-row positions-in-row-nth*)+)

lemma *length-stream-dec*[*simp*]: *length* (*stream-dec n FO x*) = *n*
unfolding *stream-dec-def* **by** *auto*

lemma *stream-enc-dec*:
$[\![\exists n.\ sdrop\ n\ (smap\ fst\ s) = sconst\ any;$
stream-all ($\lambda x.$ *length* (*snd x*) = *n*) *s*; $\forall i \in FO.\ (\exists!p.\ snd\ (s\ !!\ p)\ !\ i)]\!] \Longrightarrow$
stream-enc (*dec-word s*, *stream-dec n FO s*) = *s*
unfolding *dec-word-def*
by (*drule LeastI-ex*)
(*auto intro*!: *enc-atom-dec simp*: *smap2-alt cut-same-def*
simp del: *stake-smap sdrop-smap*
intro!: *trans*[*OF arg-cong2*[*of - - - -* (!!)] *snth-smap*]
trans[*OF arg-cong2*[*of - - - -* (@−)] *stake-sdrop*])

lemma *stream-enc-unique*:
i < *length I* $\Longrightarrow \exists p.\ I\ !\ i = Inl\ p \Longrightarrow \exists!p.\ snd\ (stream\text{-}enc\ (w, I)\ !!\ p)\ !\ i$
by *auto*

lemma *stream-dec-enc-Inl*:
$[\![stream\text{-}dec\ n\ FO\ (stream\text{-}enc\ (w, I))\ !\ i = Inl\ p';\ I\ !\ i = Inl\ p;\ i \in FO;\ i < n;$ *length I* = *n*$]\!] \Longrightarrow$
p = *p*′
unfolding *stream-dec-def*
by (*auto intro*!: *trans*[*OF - sym*[*OF positions-in-row-unique*[*OF stream-enc-unique*]]]
simp del: *stream-enc.simps*) *simp*

lemma *stream-dec-enc-Inr*:

⟦*stream-dec n FO* (*stream-enc* (*w*, *I*)) ! *i* = *Inr P'*; *I* ! *i* = *Inr P*; *i* ∉ *FO*; *i* < *n*; *length I* = *n*⟧ ⟹
P = *P'*
unfolding *stream-dec-def positions-in-row* **by** *auto*

lemma *Collect-snth*: {*p*. *P* ((*x* ## *s*) !! *p*)} ⊆ {*0*} ∪ *Suc* ' {*p*. *P* (*s* !! *p*)}
unfolding *image-def* **by** (*auto simp*: *gr0-conv-Suc*)

lemma *finite-True-in-row*: ∀ *i* < *n*. *finite* {*p*. *snd* ((*w* @− *sconst* (*any*, *replicate n False*)) !! *p*) ! *i*}
by (*induct w*) (*auto simp*: *id-def*[*symmetric*] *intro*: *finite-subset*[*OF Collect-snth*])

lemma *lang-ENC*:
assumes *FO* ⊆ {*0* ..< *n*} *SO* ⊆ {*0* ..< *n*} − *FO*
shows *lang n* (*ENC n FO*) = ⋃{*enc* (*w*, *I*) | *w I* . *length I* = *n* ∧ *wf-interp w I* ∧
(∀ *i* ∈ *FO*. *case I* ! *i of Inl* - ⇒ *True* | *Inr* - ⇒ *False*) ∧
(∀ *i* ∈ *SO*. *case I* ! *i of Inl* - ⇒ *False* | *Inr* - ⇒ *True*)}
(**is** *?L* = *?R*)
proof (*intro equalityI subsetI*)
fix *x* **assume** *L*: *x* ∈ *?L*
from *assms*(*1*) **have** *fin*: *finite FO* **by** (*auto simp*: *finite-subset*)
have ∗: *set x* ⊆ *set* (*σ* Σ *n*) **using** *subsetD*[*OF assms*(*1*)]
bspec[*OF wf-lang-wf-word*[*OF wf-rexp-ENC*[*OF fin*]] *L*]
by (*cases n*) (*auto simp*: *wf-word*)
let *?s* = *x* @− *sconst* (*any*, *replicate n False*)
from *assms* **have** *list-all* (λ*bs*. *length* (*snd bs*) = *n*) *x*
using ∗ **by** (*auto simp*: *list-all-iff σ-def set-n-lists*)
hence *stream-all* (λ*x*. *length* (*snd x*) = *n*) (*x* @− *sconst* (*any*, *replicate n False*))
by (*auto simp only*: *stream-all-shift sset-sconst length-replicate snd-conv*)
moreover
{ fix *m* **assume** *m* ∈ *FO*
with *assms* **have** *m* < *n* **by** (*auto simp*: *max-idx-vars*)
with *L* ‹*m* ∈ *FO*› *assms* **obtain** *u z v* **where** *uzv*: *x* = *u* @ *z* @ *v*
u ∈ *star* (*lang n* (*Atom* (*Arbitrary-Except m False*)))
z ∈ *lang n* (*Atom* (*Arbitrary-Except m True*))
v ∈ *star* (*lang n* (*Atom* (*Arbitrary-Except m False*))) **unfolding** *ENC-def*
by (*cases n*)
(*auto simp*: *not-less max-idx-vars valid-ENC-def fin intro*!: *wf-rexp-valid-ENC finite-FOV*
dest!: *iffD1*[*OF lang-flatten-INTERSECT*, *rotated* −*1*], *fast*)
with ‹*m* < *n*› **have** ∃!*p*. *snd* (*x* ! *p*) ! *m* ∧ *p* < *length x*
proof (*intro ex1I*[*of* - *length u*])
fix *p* **assume** *m* < *n snd* (*x* ! *p*) ! *m* ∧ *p* < *length x*
with *star-Arbitrary-ExceptD*[*OF uzv*(*2*)] *Arbitrary-ExceptD*[*OF uzv*(*3*)] *star-Arbitrary-ExceptD*[*OF uzv*(*4*)]
show *p* = *length u* **by** (*cases rule*: *linorder-cases*) (*auto simp*: *nth-append uzv*(*1*))
qed (*auto dest*!: *Arbitrary-ExceptD*)

```
    then obtain p where p: p < length x snd (x ! p) ! m
      ⋀q. snd (x ! q) ! m ∧ q < length x ⟶ q = p by auto
    from this(1,2) have ∃!p. snd (?s !! p) ! m
    proof (intro ex1I[of - p])
      fix q from p(1,2) p(3)[of q] ‹m < n› show snd (?s !! q) ! m ⟹ q = p
        by (cases q < length x)  auto
    qed auto
  }
  moreover have sdrop (length x) (smap fst (x @− sconst (any, replicate n
False))) = sconst any
    unfolding sdrop-smap by (simp add: sdrop-shift)
  ultimately have enc-dec: stream-enc (dec-word ?s, stream-dec n FO ?s) =
    x @− sconst (any, replicate n False) by (intro stream-enc-dec) auto
  define I where I = stream-dec n FO ?s
  with assms have wf-interp (dec-word ?s) I ∧
   (∀ i ∈ FO. case I ! i of Inl - ⇒ True | Inr - ⇒ False) ∧
  (∀ i ∈ SO. case I ! i of Inl - ⇒ False | Inr - ⇒ True) unfolding I-def dec-word-def
    by (auto dest: stream-dec-not-Inr stream-dec-not-Inl simp: σ-def  max-idx-vars
        dest!: subsetD[OF set-cut-same[of any map fst x]] subsetD[OF ∗]  split:
sum.splits)
      (auto simp: stream-dec-def positions-in-row finite-True-in-row)
  moreover have length I = n unfolding I-def by simp
  moreover have x ∈ enc (dec-word ?s, I) unfolding I-def
    by (simp add: enc-dec cut-same-shift-sconst del: stream-enc.simps)
  ultimately show x ∈ ?R by blast
next
  fix x assume x ∈ ?R
  then obtain w I where I: x ∈ enc (w, I) wf-interp w I ∧
   (∀ i ∈ FO. case I ! i of Inl - ⇒ True | Inr - ⇒ False) ∧
   (∀ i ∈ SO. case I ! i of Inl - ⇒ False | Inr - ⇒ True) length I = n by blast
  { fix i from I(2) have (w @− sconst any) !! i ∈ set Σ by (cases i < length w)
auto } note ∗ = this
  from I have x @− sconst (any, replicate (length I) False) = stream-enc (w, I)
(is x @− ?F = ?s)
    by (intro stream-enc-enc[symmetric]) auto
  with ∗ ‹length I = n› have ∀ x ∈ set x. length (snd x) = n ∧ fst x ∈ set Σ
    by (auto dest!: shift-snth-less[of - - ?F, symmetric] simp: in-set-conv-nth)
  thus x ∈ ?L
  proof (cases FO = {})
    case False
    hence nonempty: valid-ENC n ‘ FO ≠ {} by simp
   have finite: finite (valid-ENC n ‘ FO) by (rule finite-imageI[OF finite-subset[OF
assms(1)]]) simp
    from False assms(1) have 0 < n by (cases n) (auto split: dest!: max-idx-vars)
    with wf-rexp-valid-ENC assms have wf-rexp: ∀ x ∈ valid-ENC n ‘ FO. wf n x
      by (auto simp: max-idx-vars)
    { fix r assume r ∈ FO
      with I(2) obtain p where p: I ! r = Inl p by (cases I ! r) auto
    from ‹r ∈ FO› assms I(2,3) have r: r < length I by (auto dest!: max-idx-vars)
```

from *p I*(*1*,*2*) *r* **have** *p* < *length x*
using *less-length-cut-same-Inl*[*of I r p w*] **by** *auto*
with *p I r* ∗
have [*x* ! *p*] ∈ *lang n* (*Atom* (*Arbitrary-Except r True*))
by (*subst encD*[*of x w I*]) (*auto simp del*: *lang.simps intro*!: *enc-atom-lang-Arbitrary-Except-True*)
moreover
from *p I r* ∗ **have** *take p x* ∈ *star* (*lang n* (*Atom* (*Arbitrary-Except r False*)))
by (*subst encD*[*of x*]) (*auto simp del*: *lang.simps simp*: *in-set-conv-nth intro*!: *Ball-starI enc-atom-lang-Arbitrary-Except-False*)
moreover
from *p I r* ∗ **have** *drop* (*Suc p*) *x* ∈ *star* (*lang n* (*Atom* (*Arbitrary-Except r False*)))
by (*subst encD*[*of x*]) (*auto simp*: *in-set-conv-nth simp del*: *lang.simps snth.simps intro*!: *Ball-starI enc-atom-lang-Arbitrary-Except-False*)
ultimately have *take p x* @ [*x* ! *p*] @ *drop* (*p* + *1*) *x* ∈ *lang n* (*valid-ENC n r*)
using ‹*0* < *n*› **unfolding** *valid-ENC-def* **by** (*auto simp del*: *append.simps*)
hence *x* ∈ *lang n* (*valid-ENC n r*) **using** *id-take-nth-drop*[*OF* ‹*p* < *length x*›]
by *auto*
}
with *False lang-flatten-INTERSECT*[*OF finite nonempty wf-rexp*] **show** *?thesis*
by (*auto simp*: *ENC-def*)
qed (*simp add*: *ENC-def*, *auto simp*: *σ-def set-n-lists image-iff*)
qed

lemma *lang-ENC-formula*:
assumes *wf-formula n φ*
shows *lang n* (*ENC n* (*FOV φ*)) = ⋃{*enc* (*w*, *I*) | *w I* . *length I* = *n* ∧ *wf-interp-for-formula* (*w*, *I*) *φ*}
proof −
from *assms max-idx-vars* **have** ∗: *FOV φ* ⊆ {*0* ..< *n*} *SOV φ* ⊆ {*0* ..< *n*} − *FOV φ* **by** *auto*
show *?thesis* **unfolding** *lang-ENC*[*OF* ∗] **by** *simp*
qed

13.2 Welldefinedness of enc wrt. Models

lemma *wf-interp-for-formula-FExists*:
⟦*wf-formula* (*length I*) (*FExists φ*)⟧⟹
wf-interp-for-formula (*w*, *I*) (*FExists φ*) ⟷ (∀ *p*. *wf-interp-for-formula* (*w*, *Inl p* # *I*) *φ*)
by (*auto simp*: *nth-Cons′ split*: *if-split-asm*)

lemma *wf-interp-for-formula-any-Inl*: *wf-interp-for-formula* (*w*, *Inl p* # *I*) *φ* ⟹
∀ *p*. *wf-interp-for-formula* (*w*, *Inl p* # *I*) *φ*
by (*auto simp*: *nth-Cons′ split*: *if-split-asm*)

lemma *wf-interp-for-formula-FEXISTS*:
⟦*wf-formula* (*length I*) (*FEXISTS φ*)⟧⟹

wf-interp-for-formula (*w*, *I*) (*FEXISTS* φ) $\longleftrightarrow$ ($\forall$ *P*. *finite P* $\longrightarrow$ *wf-interp-for-formula* (*w*, *Inr P # I*) φ)
by (*auto simp*: *nth-Cons' split*: *if-split-asm*)

lemma *wf-interp-for-formula-any-Inr*: *wf-interp-for-formula* (*w*, *Inr P # I*) φ $\Longrightarrow$
$\forall$ *P*. *finite P* $\longrightarrow$ *wf-interp-for-formula* (*w*, *Inr P # I*) φ
by (*auto simp*: *nth-Cons' split*: *if-split-asm*)

lemma *wf-interp-for-formula-FOr*:
wf-interp-for-formula (*w*, *I*) (*FOr* φ*1* φ*2*) =
(*wf-interp-for-formula* (*w*, *I*) φ*1* $\wedge$ *wf-interp-for-formula* (*w*, *I*) φ*2*)
by *auto*

lemma *wf-interp-for-formula-FAnd*:
wf-interp-for-formula (*w*, *I*) (*FAnd* φ*1* φ*2*) =
(*wf-interp-for-formula* (*w*, *I*) φ*1* $\wedge$ *wf-interp-for-formula* (*w*, *I*) φ*2*)
by *auto*

lemma *enc-wf-interp*:
$[\![$*wf-formula* (*length I*) φ; *wf-interp-for-formula* (*w*, *I*) φ; *x* $\in$ *enc* (*w*, *I*)$]\!] \Longrightarrow$
wf-interp-for-formula (*dec-word* (*x* @− *sconst* (*any*, *replicate* (*length I*) *False*)),
stream-dec (*length I*) (*FOV* φ) (*x* @− *sconst* (*any*, *replicate* (*length I*) *False*)))
φ
using
stream-dec-Inl[*of - FOV* φ *length I stream-enc* (*w*, *I*), *OF - bspec*[*OF max-idx-vars*]]
stream-dec-Inr[*of - FOV* φ *length I stream-enc* (*w*, *I*), *OF - bspec*[*OF max-idx-vars*]]
by (*auto split*: *sum.splits intro*: *Inr-dec-finite*[*OF finite-True-in-row*] *simp*: *max-idx-vars dec-word-def*
dest!: *stream-dec-not-Inl stream-dec-not-Inr subsetD*[*OF set-cut-same*] *simp del*: *stream-enc.simps*)
(*auto simp*: *cut-same-def in-set-zip smap2-alt shift-snth*)

lemma *enc-atom-welldef*: $\forall$ *x a*. *enc-atom I x a* = *enc-atom I' x a* $\Longrightarrow$ *m* < *length I* $\Longrightarrow$
(*case* (*I* ! *m*, *I'* ! *m*) *of* (*Inl p*, *Inl q*) $\Rightarrow$ *p* = *q* | (*Inr P*, *Inr Q*) $\Rightarrow$ *P* = *Q* | - $\Rightarrow$ *True*)
proof (*induct length I arbitrary*: *I I' m*)
case (*Suc n I I'*)
then obtain *x xs x' xs'* **where** $*$: *I* = *x* # *xs I'* = *x'* # *xs'*
by (*fastforce simp*: *Suc-length-conv map-eq-Cons-conv*)
with *Suc* **show** *?case*
proof (*cases m*)
case *0* **thus** *?thesis* **using** *Suc*(*3*) **unfolding** $*$
by (*cases x x' rule*: *sum.exhaust*[*case-product sum.exhaust*]) *auto*
qed *auto*
qed *simp*

lemma *stream-enc-welldef*: $[\![$*stream-enc* (*w*, *I*) = *stream-enc* (*w'*, *I'*); *wf-formula* (*length I*) φ;

wf-interp-for-formula (w, I) φ; *wf-interp-for-formula* (w', I') φ⟧ ⟹
$(w, I) \models \varphi \longleftrightarrow (w', I') \models \varphi$
proof (*induction* φ *arbitrary*: w w' I I')
case (*FQ a m*) **thus** *?case* **using** *enc-atom-welldef*[*of I I' m*]
by (*simp split*: *sum.splits add*: *smap2-alt shift-snth*)
(*metis snth-siterate*[*of id*, *simplified id-funpow id-apply*])
next
case (*FLess m1 m2*) **thus** *?case* **using** *enc-atom-welldef*[*of I I' m1*] *enc-atom-welldef*[*of I I' m2*]
by (*auto split*: *sum.splits simp add*: *smap2-alt*)
next
case (*FIn m M*) **thus** *?case* **using** *enc-atom-welldef*[*of I I' m*] *enc-atom-welldef*[*of I I' M*]
by (*auto split*: *sum.splits simp add*: *smap2-alt*)
next
case (*FOr* $\varphi 1$ $\varphi 2$) **show** *?case* **unfolding** *satisfies.simps*(5)
proof (*intro disj-cong*)
from *FOr*(3−6) **show** $(w, I) \models \varphi 1 \longleftrightarrow (w', I') \models \varphi 1$
by (*intro FOr*(1)) *auto*
next
from *FOr*(3−6) **show** $(w, I) \models \varphi 2 \longleftrightarrow (w', I') \models \varphi 2$
by (*intro FOr*(2)) *auto*
qed
next
case (*FAnd* $\varphi 1$ $\varphi 2$) **show** *?case* **unfolding** *satisfies.simps*(6)
proof (*intro conj-cong*)
from *FAnd*(3−6) **show** $(w, I) \models \varphi 1 \longleftrightarrow (w', I') \models \varphi 1$
by (*intro FAnd*(1)) *auto*
next
from *FAnd*(3−6) **show** $(w, I) \models \varphi 2 \longleftrightarrow (w', I') \models \varphi 2$
by (*intro FAnd*(2)) *auto*
qed
next
case (*FExists* φ)
hence *length*: *length* I' = *length* I **by** (*metis length-snth-enc*)
show *?case*
proof
assume $(w, I) \models$ *FExists* φ
with *FExists.prems*(3) **obtain** p **where** $(w, \mathit{Inl}\ p\ \#\ I) \models \varphi$ **by** *auto*
moreover
with *FExists.prems*(1,2) **have** $(w', \mathit{Inl}\ p\ \#\ I') \models \varphi$
proof (*intro iffD1*[*OF FExists.IH*[*of w Inl p # I w' Inl p # I'*]])
from *FExists.prems*(2,3) **show** *wf-interp-for-formula* $(w, \mathit{Inl}\ p\ \#\ I)$ φ
by (*blast dest*: *wf-interp-for-formula-FExists*[*of I*])
next
from *FExists.prems*(2,4) **show** *wf-interp-for-formula* $(w', \mathit{Inl}\ p\ \#\ I')$ φ
by (*blast dest*: *wf-interp-for-formula-FExists*[*of I'*, *unfolded length*])
qed (*auto simp*: *smap2-alt split*: *sum.splits if-split-asm*)
ultimately show $(w', I') \models$ *FExists* φ **by** *auto*

```
  next
    assume (w', I') ⊨ FExists φ
    with FExists.prems(1,2,4) obtain p where (w', Inl p # I') ⊨ φ by auto
    moreover
    with FExists.prems(1,2) have (w, Inl p # I) ⊨ φ
    proof (intro iffD2[OF FExists.IH[of w Inl p # I w' Inl p # I']])
      from FExists.prems(2,3) show wf-interp-for-formula (w, Inl p # I) φ
        by (blast dest: wf-interp-for-formula-FExists[of I])
    next
      from FExists.prems(2,4) show wf-interp-for-formula (w', Inl p # I') φ
        by (blast dest: wf-interp-for-formula-FExists[of I', unfolded length])
    qed (auto simp: smap2-alt split: sum.splits if-split-asm)
    ultimately show (w, I) ⊨ FExists φ by auto
  qed
next
  case (FEXISTS φ)
  hence length: length I' = length I by (metis length-snth-enc)
  show ?case
  proof
    assume (w, I) ⊨ FEXISTS φ
    with FEXISTS.prems(3) obtain P where finite P (w, Inr P # I) ⊨ φ by
auto
    moreover
    with FEXISTS.prems(1,2) have (w', Inr P # I') ⊨ φ
    proof (intro iffD1[OF FEXISTS.IH[of w Inr P # I w' Inr P # I']])
      from FEXISTS.prems(2,3) ‹finite P› show wf-interp-for-formula (w, Inr P
# I) φ
        by (blast dest: wf-interp-for-formula-FEXISTS[of I])
    next
      from FEXISTS.prems(2,4) ‹finite P› show wf-interp-for-formula (w', Inr P
# I') φ
        by (blast dest: wf-interp-for-formula-FEXISTS[of I', unfolded length])
    qed (auto simp: smap2-alt split: sum.splits if-split-asm)
    ultimately show (w', I') ⊨ FEXISTS φ by auto
  next
    assume (w', I') ⊨ FEXISTS φ
    with FEXISTS.prems(1,2,4) obtain P where finite P (w', Inr P # I') ⊨ φ
by auto
    moreover
    with FEXISTS.prems have (w, Inr P # I) ⊨ φ
    proof (intro iffD2[OF FEXISTS.IH[of w Inr P # I w' Inr P # I']])
      from FEXISTS.prems(2,3) ‹finite P› show wf-interp-for-formula (w, Inr P
# I) φ
        by (blast dest: wf-interp-for-formula-FEXISTS[of I])
    next
      from FEXISTS.prems(2,4) ‹finite P› show wf-interp-for-formula (w', Inr P
# I') φ
        by (blast dest: wf-interp-for-formula-FEXISTS[of I', unfolded length])
    qed (auto simp: smap2-alt split: sum.splits if-split-asm)
```

ultimately show $(w, I) \models FEXISTS\ \varphi$ **by** *auto*
qed
qed *auto*

lemma $lang_{WS1S}$-*FOr*:
assumes *wf-formula n* (*FOr* φ_1 φ_2)
shows $lang_{WS1S}$ *n* (*FOr* φ_1 φ_2) $\subseteq$
($lang_{WS1S}$ *n* φ_1 $\cup$ $lang_{WS1S}$ *n* φ_2) $\cap$ $\bigcup$ {*enc* (*w*, *I*) | *w I*. *length I* = *n* $\wedge$ *wf-interp-for-formula* (*w*, *I*) (*FOr* φ_1 φ_2)}
(**is** - $\subseteq$ (*?L1* $\cup$ *?L2*) $\cap$ *?ENC*)
proof (*intro equalityI subsetI*)
fix *x* **assume** *x* $\in$ $lang_{WS1S}$ *n* (*FOr* φ_1 φ_2)
then obtain *w I* **where**
$*$: *x* $\in$ *enc* (*w*, *I*) *wf-interp-for-formula* (*w*, *I*) (*FOr* φ_1 φ_2) *length I* = *n* **and**
satisfies (*w*, *I*) φ_1 $\vee$ *satisfies* (*w*, *I*) φ_2 **unfolding** $lang_{WS1S}$-*def* **by** *auto*
thus *x* $\in$ (*?L1* $\cup$ *?L2*) $\cap$ *?ENC*
proof (*elim disjE*)
assume *satisfies* (*w*, *I*) φ_1
with $*$ **have** *x* $\in$ *?L1* **using** *assms* **unfolding** $lang_{WS1S}$-*def* **by** (*fastforce simp del*: *enc.simps*)
with $*$ **show** *?thesis* **by** *auto*
next
assume *satisfies* (*w*, *I*) φ_2
with $*$ **have** *x* $\in$*?L2* **using** *assms* **unfolding** $lang_{WS1S}$-*def* **by** (*fastforce simp del*: *enc.simps*)
with $*$ **show** *?thesis* **by** *auto*
qed
qed

lemma $lang_{WS1S}$-*FAnd*:
assumes *wf-formula n* (*FAnd* φ_1 φ_2)
shows $lang_{WS1S}$ *n* (*FAnd* φ_1 φ_2) $\subseteq$
$lang_{WS1S}$ *n* φ_1 $\cap$ $lang_{WS1S}$ *n* φ_2 $\cap$ $\bigcup$ {*enc* (*w*, *I*) | *w I*. *length I* = *n* $\wedge$ *wf-interp-for-formula* (*w*, *I*) (*FAnd* φ_1 φ_2)}
using *assms* **unfolding** $lang_{WS1S}$-*def* **by** (*fastforce simp del*: *enc.simps*)

13.3 From WS1S to Regular expressions

fun *rexp-of* :: *nat* $\Rightarrow$ *'a formula* $\Rightarrow$ (*'a atom*) *rexp* **where**
rexp-of n (*FQ a m*) =
Inter (*TIMES* [*rexp.Not Zero*, *Atom* (*AQ m a*), *rexp.Not Zero*])
(*ENC n* (*FOV* (*FQ a m*)))
| *rexp-of n* (*FLess m1 m2*) = (*if m1* = *m2 then Zero else*
Inter (*TIMES* [*rexp.Not Zero*, *Atom* (*Arbitrary-Except m1 True*),
rexp.Not Zero, *Atom* (*Arbitrary-Except m2 True*),
rexp.Not Zero]) (*ENC n* (*FOV* (*FLess m1 m2* :: *'a formula*))))
| *rexp-of n* (*FIn m M*) =
Inter (*TIMES* [*rexp.Not Zero*, *Atom* (*Arbitrary-Except2 m M*), *rexp.Not Zero*])
(*ENC n* (*FOV* (*FIn m M* :: *'a formula*)))

| *rexp-of n* (*FNot* φ) = *Inter* (*rexp.Not* (*rexp-of n* φ)) (*ENC n* (*FOV* (*FNot* φ)))
| *rexp-of n* (*FOr* φ_1 φ_2) = *Inter* (*Plus* (*rexp-of n* φ_1) (*rexp-of n* φ_2)) (*ENC n* (*FOV* (*FOr* φ_1 φ_2)))
| *rexp-of n* (*FAnd* φ_1 φ_2) = *INTERSECT* [*rexp-of n* φ_1, *rexp-of n* φ_2, *ENC n* (*FOV* (*FAnd* φ_1 φ_2))]
| *rexp-of n* (*FExists* φ) = *samequot-exec* (*any*, *replicate n False*) (*Pr* (*rexp-of* (*n* + *1*) φ))
| *rexp-of n* (*FEXISTS* φ) = *samequot-exec* (*any*, *replicate n False*) (*Pr* (*rexp-of* (*n* + *1*) φ))

fun *rexp-of-alt* :: *nat* ⇒ *'a formula* ⇒ (*'a atom*) *rexp* **where**
  rexp-of-alt n (*FQ a m*) =
    TIMES [*rexp.Not Zero*, *Atom* (*AQ m a*), *rexp.Not Zero*]
| *rexp-of-alt n* (*FLess m1 m2*) = (*if m1 = m2 then Zero else*
    TIMES [*rexp.Not Zero*, *Atom* (*Arbitrary-Except m1 True*),
      rexp.Not Zero, *Atom* (*Arbitrary-Except m2 True*),
      rexp.Not Zero])
| *rexp-of-alt n* (*FIn m M*) =
    TIMES [*rexp.Not Zero*, *Atom* (*Arbitrary-Except2 m M*), *rexp.Not Zero*]
| *rexp-of-alt n* (*FNot* φ) = *rexp.Not* (*rexp-of-alt n* φ)
| *rexp-of-alt n* (*FOr* φ_1 φ_2) = *Plus* (*rexp-of-alt n* φ_1) (*rexp-of-alt n* φ_2)
| *rexp-of-alt n* (*FAnd* φ_1 φ_2) = *Inter* (*rexp-of-alt n* φ_1) (*rexp-of-alt n* φ_2)
| *rexp-of-alt n* (*FExists* φ) = *samequot-exec* (*any*, *replicate n False*) (*Pr* (*Inter* (*rexp-of-alt* (*n* + *1*) φ) (*ENC* (*Suc n*) (*FOV* φ))))
| *rexp-of-alt n* (*FEXISTS* φ) = *samequot-exec* (*any*, *replicate n False*) (*Pr* (*Inter* (*rexp-of-alt* (*n* + *1*) φ) (*ENC* (*Suc n*) (*FOV* φ))))

definition *rexp-of'* *n* φ = *Inter* (*rexp-of-alt n* φ) (*ENC n* (*FOV* φ))

fun *rexp-of-alt'* :: *nat* ⇒ *'a formula* ⇒ (*'a atom*) *rexp* **where**
  rexp-of-alt' n (*FQ a m*) = *TIMES* [*Full*, *Atom* (*AQ m a*), *Full*]
| *rexp-of-alt' n* (*FLess m1 m2*) = (*if m1 = m2 then Zero else*
    TIMES [*Full*, *Atom* (*Arbitrary-Except m1 True*), *Full*, *Atom* (*Arbitrary-Except m2 True*), *Full*])
| *rexp-of-alt' n* (*FIn m M*) = *TIMES* [*Full*, *Atom* (*Arbitrary-Except2 m M*), *Full*]
| *rexp-of-alt' n* (*FNot* φ) = *rexp.Not* (*rexp-of-alt' n* φ)
| *rexp-of-alt' n* (*FOr* φ_1 φ_2) = *Plus* (*rexp-of-alt' n* φ_1) (*rexp-of-alt' n* φ_2)
| *rexp-of-alt' n* (*FAnd* φ_1 φ_2) = *Inter* (*rexp-of-alt' n* φ_1) (*rexp-of-alt' n* φ_2)
| *rexp-of-alt' n* (*FExists* φ) = *samequot-exec* (*any*, *replicate n False*) (*Pr* (*Inter* (*rexp-of-alt'* (*n* + *1*) φ) (*ENC* (*n* + *1*) {*0*})))
| *rexp-of-alt' n* (*FEXISTS* φ) = *samequot-exec* (*any*, *replicate n False*) (*Pr* (*rexp-of-alt'* (*n* + *1*) φ))

definition *rexp-of''* *n* φ = *Inter* (*rexp-of-alt' n* φ) (*ENC n* (*FOV* φ))

lemma *enc-eqI*:
  assumes *x* ∈ *enc* (*w*, *I*) *x* ∈ *enc* (*w'*, *I'*) *wf-interp-for-formula* (*w*, *I*) φ *wf-interp-for-formula* (*w'*, *I'*) φ
    length I = *length I'*

shows *enc* $(w, I) = enc\ (w', I')$
proof −
from *assms* **have** *stream-enc* $(w, I) = stream\text{-}enc\ (w', I')$
by (*intro box-equals*[*OF - stream-enc-enc*[*symmetric*] *stream-enc-enc*[*symmetric*]])
auto
thus *?thesis* **using** *assms*(*5*) **by** *auto*
qed

lemma *enc-eq-welldef*:
⟦*enc* $(w, I) = enc\ (w', I')$; *wf-formula* (*length I*) φ; *wf-interp-for-formula* (w, I)
φ ;*wf-interp-for-formula* (w', I') φ⟧ $\Longrightarrow$
$(w, I) \models \varphi \longleftrightarrow (w', I') \models \varphi$
by (*intro stream-enc-welldef*) (*auto simp del*: *stream-enc.simps intro*!: *enc-stream-enc*)

lemma *enc-welldef*:
⟦$x \in enc\ (w, I)$; $x \in enc\ (w', I')$; *length I* = *length I'*; *wf-formula* (*length I*) φ;
wf-interp-for-formula (w, I) φ ;*wf-interp-for-formula* (w', I') φ⟧ $\Longrightarrow$
$(w, I) \models \varphi \longleftrightarrow (w', I') \models \varphi$
by (*intro enc-eq-welldef*[*OF enc-eqI*])

lemma *wf-rexp-of*: *wf-formula n* $\varphi \Longrightarrow$ *wf n* (*rexp-of n* φ)
by (*induct* φ *arbitrary*: *n*)
(*auto intro*!: *wf-samequot-exec wf-rexp-ENC*,
auto simp: *max-idx-vars finite-FOV*)

theorem $lang_{WS1S}$*-rexp-of*: *wf-formula n* $\varphi \Longrightarrow lang_{WS1S}\ n\ \varphi$ = *lang n* (*rexp-of*
n φ)
(**is** - $\Longrightarrow$ - = *?L n* φ)
proof (*induct* φ *arbitrary*: *n*)
case (*FQ a m*)
show *?case*
proof (*intro equalityI subsetI*)
fix *x* **assume** $x \in lang_{WS1S}\ n\ (FQ\ a\ m)$
then obtain *w I* **where**
∗: $x \in enc\ (w, I)$ *wf-interp-for-formula* (w, I) (*FQ a m*) *length I* = *n* (w, I)
$\models$ *FQ a m*
unfolding $lang_{WS1S}$*-def* **by** *blast*
hence *x-alt*: *x* = *map* (*case-prod* (*enc-atom I*)) (*zip* [*0* ..< *length x*] (*stake*
(*length x*) (*w* @− *sconst any*)))
by (*intro encD*) *auto*
from *FQ*(*1*) ∗(*2*,*4*) **obtain** *p* **where** *p*: *I* ! *m* = *Inl p*
by (*auto simp*: *all-set-conv-all-nth split*: *sum.splits*)
with *FQ*(*1*) ∗ **have** *p-less*: *p* < *length x*
by (*auto simp del*: *stream-enc.simps intro*: *trans-less-add1*[*OF less-length-cut-same-Inl*])
hence *enc-atom*: *x* ! *p* = *enc-atom I p* ((*w* @− *sconst any*) !! *p*) (**is** - =
enc-atom - - *?p*)
by (*subst x-alt*, *simp*)
with ∗(*1*) *p-less*(*1*) **have** *x* = *take p x* @ [*enc-atom I p ?p*] @ *drop* (*p* + *1*) *x*
using *id-take-nth-drop*[*of p x*] **by** *auto*

moreover
from ∗(2,3,4) *FQ*(1) *p* **have** [*enc-atom I p ?p*] ∈ *lang n* (*Atom* (*AQ m a*))
by (*auto simp del*: *lang.simps intro*!: *enc-atom-lang-AQ*)
moreover from ∗(2,3) **have** *take p x* ∈ *lang n* (*rexp.Not Zero*)
by (*subst x-alt*) (*auto simp*: *in-set-zip shift-snth intro*!: *enc-atom-σ dest*!: *in-set-takeD*)
moreover from ∗(2,3) **have** *drop* (*Suc p*) *x* ∈ *lang n* (*rexp.Not Zero*)
by (*subst x-alt*) (*auto simp*: *in-set-zip shift-snth intro*!: *enc-atom-σ dest*!: *in-set-dropD*)
ultimately show *x* ∈ *?L n* (*FQ a m*) **using** ∗(1,2,3)
unfolding *rexp-of.simps lang.simps*(6,9) *rexp-of-list.simps lang-ENC-formula*[*OF FQ*]
by (*auto elim*: *ssubst simp del*: *o-apply append.simps lang.simps enc.simps*)
next
fix *x* **let** *?x* = *x* @− *sconst* (*any*, *replicate n False*)
assume *x*: *x* ∈ *?L n* (*FQ a m*)
with *FQ* **obtain** *w I* **where**
I: *x* ∈ *enc* (*w*, *I*) *length I* = *n wf-interp-for-formula* (*w*, *I*) (*FQ a m*)
unfolding *rexp-of.simps lang.simps lang-ENC-formula*[*OF FQ*] **by** *fastforce*
hence *stream-enc*: *stream-enc* (*w*, *I*) = *?x* **using** *stream-enc-enc* **by** *auto*
from *I FQ* **obtain** *p* **where** *m*: *I* ! *m* = *Inl p m* < *length I* **by** (*auto split*: *sum.splits*)
with *I* **have** *wf-interp-for-formula* (*dec-word ?x*, *stream-dec n* {*m*} *?x*) (*FQ a m*) **unfolding** *I*(1)
using *enc-wf-interp*[*OF FQ*(1)[*folded I*(2)]] **by** *auto*
moreover
from *x* **obtain** *u1 u u2* **where** *x* = *u1* @ *u* @ *u2 u* ∈ *lang n* (*Atom* (*AQ m a*))
unfolding *rexp-of.simps lang.simps rexp-of-list.simps* **using** *concE* **by** *fast*
with *FQ*(1) **obtain** *v* **where** *v*: *x* = *u1* @ [*v*] @ *u2 snd v* ! *m fst v* = *a*
using *AQ-D*[*of u n m a*] **by** *fastforce*
from *v* **have** *u*: *length u1* < *length x* **by** *auto*
{ from *v* **have** *snd* (*x* ! *length u1*) ! *m* **by** *auto*
moreover
from *m I* **have** *p* < *length x snd* (*x* ! *p*) ! *m* **by** (*auto dest*: *enc-Inl simp del*: *enc.simps*)
moreover
from *m I* **have** *ex1*: ∃!*p*. *snd* (*stream-enc* (*w*, *I*) !! *p*) ! *m* **by** (*intro stream-enc-unique*) *auto*
ultimately have *p* = *length u1* **unfolding** *stream-enc* **using** *u I*(3) **by** *auto*
} note ∗ = *this*
from *v* **have** *v* = *x* ! *length u1* **by** *simp*
with *u* **have** *?x* !! *length u1* = *v* **by** (*auto simp*: *shift-snth*)
with ∗ *m I v*(3) **have** (*dec-word ?x*, *stream-dec n* {*m*} *?x*) ⊨ *FQ a m*
using *stream-enc-enc*[*OF* - *I*(1), *symmetric*] *less-length-cut-same*[*of w any length u1 a*]
by (*auto simp del*: *enc.simps stream-enc.simps simp*: *dec-word-stream-enc dest*!:
stream-dec-enc-Inl stream-dec-not-Inr split: *sum.splits*)

```
    (auto simp: smap2-alt cut-same-def)
  moreover from m I(2)
   have stream-enc-dec: stream-enc (dec-word (stream-enc (w, I)), stream-dec n
{m} (stream-enc (w, I))) = stream-enc (w, I)
    by (intro stream-enc-dec)
     (auto simp: smap2-alt sdrop-snth shift-snth intro: stream-enc-unique,
      auto simp: smap2-szip stream.set-map)
   moreover from I have wf-word n x unfolding wf-word by (auto elim:
enc-set-σ simp del: enc.simps)
  ultimately show x ∈ lang_WS1S n (FQ a m) unfolding lang_WS1S-def using
m I(1,3)
    by (auto simp del: enc.simps stream-enc.simps intro!: exI[of - enc (dec-word
?x, stream-dec n {m} ?x)],
      fastforce simp del: enc.simps stream-enc.simps,
      auto simp del: stream-enc.simps simp: stream-enc[symmetric] I(2))
 qed
next
 case (FLess m m')
 show ?case
 proof (cases m = m')
  case False
  thus ?thesis
  proof (intro equalityI subsetI)
   fix x assume x ∈ lang_WS1S n (FLess m m')
   then obtain w I where
    *: x ∈ enc (w, I) wf-interp-for-formula (w, I) (FLess m m') length I = n
(w, I) ⊨ FLess m m'
    unfolding lang_WS1S-def by blast
   hence x-alt: x = map (case-prod (enc-atom I)) (zip [0 ..< length x] (stake
(length x) (w @− sconst any)))
    by (intro encD) auto
   from FLess(1) *(2,4) obtain p q where pq: I ! m = Inl p I ! m' = Inl q p
< q
    by (auto simp: all-set-conv-all-nth split: sum.splits)
   with FLess(1) *(1,2,3) have pq-less: p < length x q < length x
   by (auto simp del: stream-enc.simps intro!: trans-less-add1[OF less-length-cut-same-Inl])
   hence enc-atom: x ! p = enc-atom I p ((w @− sconst any) !! p) (is - =
enc-atom - - ?p)
          x ! q = enc-atom I q ((w @− sconst any) !! q) (is - = enc-atom
- - ?q) by (subst x-alt, simp)+
   with *(1) pq-less(1) have x = take p x @ [enc-atom I p ?p] @ drop (p + 1)
x
    using id-take-nth-drop[of p x] by auto
   also have drop (p + 1) x = take (q − p − 1) (drop (p + 1) x) @
    [enc-atom I q ?q] @ drop (q − p) (drop (p + 1) x) (is - = ?LHS)
   using id-take-nth-drop[of q − p − 1 drop (p + 1) x] pq pq-less(2) enc-atom(2)
by auto
   finally have x = take p x @ [enc-atom I p ?p] @ ?LHS .
   moreover from *(2,3) FLess(1) pq(1)
```

have [*enc-atom I p ?p*] ∈ *lang n* (*Atom* (*Arbitrary-Except m True*))
by (*intro enc-atom-lang-Arbitrary-Except-True*) (*auto simp*: *shift-snth*)
moreover from ∗(*2*,*3*) *FLess*(*1*) *pq*(*2*)
have [*enc-atom I q ?q*] ∈ *lang n* (*Atom* (*Arbitrary-Except m′ True*))
by (*intro enc-atom-lang-Arbitrary-Except-True*) (*auto simp*: *shift-snth*)
moreover from ∗(*2*,*3*) **have** *take p x* ∈ *lang n* (*rexp.Not Zero*)
by (*subst x-alt*) (*auto simp*: *in-set-zip shift-snth intro*!: *enc-atom-σ dest*!: *in-set-takeD*)
moreover from ∗(*2*,*3*) **have** *take* (*q* − *p* − *1*) (*drop* (*Suc p*) *x*) ∈ *lang n* (*rexp.Not Zero*)
by (*subst x-alt*) (*auto simp*: *in-set-zip shift-snth intro*!: *enc-atom-σ dest*!: *in-set-dropD in-set-takeD*)
moreover from ∗(*2*,*3*) **have** *drop* (*q* − *p*) (*drop* (*Suc p*) *x*) ∈ *lang n* (*rexp.Not Zero*)
by (*subst x-alt*) (*auto simp*: *in-set-zip shift-snth intro*!: *enc-atom-σ dest*!: *in-set-dropD*)
ultimately show *x* ∈ *?L n* (*FLess m m′*) **using** ∗(*1*,*2*,*3*)
unfolding *rexp-of.simps lang.simps*(*6*,*9*) *rexp-of-list.simps Int-Diff lang-ENC-formula*[*OF FLess*] *if-not-P*[*OF False*]
by (*auto elim*: *ssubst simp del*: *o-apply append.simps lang.simps enc.simps*)
next
fix *x* **let** *?x* = *x* @− *sconst* (*any*, *replicate n False*)
assume *x*: *x* ∈ *?L n* (*FLess m m′*)
with *FLess* **obtain** *w I* **where**
I: *x* ∈ *enc* (*w*, *I*) *length I* = *n wf-interp-for-formula* (*w*, *I*) (*FLess m m′*)
unfolding *rexp-of.simps lang.simps lang-ENC-formula*[*OF FLess*] *if-not-P*[*OF False*] **by** *fastforce*
hence *stream-enc*: *stream-enc* (*w*, *I*) = *x* @− *sconst* (*any*, *replicate n False*) **using** *stream-enc-enc* **by** *auto*
from *I FLess* **obtain** *p p′* **where** *m*: *I* ! *m* = *Inl p m* < *length I I* ! *m′* = *Inl p′ m′* < *length I*
by (*auto split*: *sum.splits*)
with *I* **have** *wf-interp-for-formula* (*dec-word ?x*, *stream-dec n* {*m*, *m′*} *?x*) (*FLess m m′*) **unfolding** *I*(*1*)
using *enc-wf-interp*[*OF FLess*(*1*)[*folded I*(*2*)]] **by** *auto*
moreover
from *x* **obtain** *u1 u u2 u′ u3* **where** *x* = *u1* @ *u* @ *u2* @ *u′* @ *u3*
u ∈ *lang n* (*Atom* (*Arbitrary-Except m True*)) *u′* ∈ *lang n* (*Atom* (*Arbitrary-Except m′ True*))
unfolding *rexp-of.simps lang.simps rexp-of-list.simps if-not-P*[*OF False*] **using** *concE* **by** *fast*
with *FLess*(*1*) **obtain** *v v′* **where** *v*: *x* = *u1* @ [*v*] @ *u2* @ [*v′*] @ *u3*
snd v ! *m snd v′* ! *m′ fst v* ∈ *set* Σ *fst v′* ∈ *set* Σ
using *Arbitrary-ExceptD*[*of u n m True*] *Arbitrary-ExceptD*[*of u′ n m′ True*]
by *simp* (*auto simp*: *σ-def*)
hence *u*: *length u1* < *length x* **and** *u′*: *Suc* (*length u1* + *length u2*) < *length x* (**is** *?u′* < -) **by** *auto*
{ from *v* **have** *snd* (*x* ! *length u1*) ! *m* **by** *auto*
moreover

```
      from m I have p < length x snd (x ! p) ! m by (auto dest: enc-Inl simp
del: enc.simps)
      moreover
        from m I have ex1: ∃!p. snd (stream-enc (w, I) !! p) ! m by (intro
stream-enc-unique) auto
       ultimately have p = length u1 unfolding stream-enc using u I(3) by
auto
    }
    { from v have snd (x ! ?u') ! m' by (auto simp: nth-append)
      moreover
      from m I have p' < length x snd (x ! p') ! m' by (auto dest: enc-Inl simp
del: enc.simps)
      moreover
      from m I have ex1: ∃!p. snd (stream-enc (w, I) !! p) ! m' unfolding I(1)
by (intro stream-enc-unique) auto
       ultimately have p' = ?u' unfolding stream-enc using u' I(3) by auto
(metis shift-snth-less)
    } note * = this ‹p = length u1›
    with m I have (dec-word ?x, stream-dec n {m, m'} ?x) ⊨ FLess m m'
      using stream-enc-enc[OF - I(1), symmetric]
      by (auto dest: stream-dec-not-Inr stream-dec-enc-Inl split: sum.splits simp
del: stream-enc.simps)
    moreover from m I(2)
      have stream-enc-dec: stream-enc (dec-word (stream-enc (w, I)), stream-dec
n {m, m'} (stream-enc (w, I))) = stream-enc (w, I)
      by (intro stream-enc-dec)
        (auto simp: smap2-alt sdrop-snth shift-snth intro: stream-enc-unique,
         auto simp: smap2-szip stream.set-map)
      moreover from I have wf-word n x unfolding wf-word by (auto elim:
enc-set-σ simp del: enc.simps)
    ultimately show x ∈ lang_WS1S n (FLess m m') unfolding lang_WS1S-def
using m I(1,3)
      by (auto simp del: enc.simps stream-enc.simps intro!: exI[of - enc (dec-word
?x, stream-dec n {m, m'} ?x)],
          fastforce simp del: enc.simps stream-enc.simps,
          auto simp del: stream-enc.simps simp: stream-enc[symmetric] I(2))
  qed
 qed (simp add: lang_WS1S-def del: o-apply)
next
 case (FIn m M)
 show ?case
 proof (intro equalityI subsetI)
  fix x assume x ∈ lang_WS1S n (FIn m M)
  then obtain w I where
    *: x ∈ enc (w, I) wf-interp-for-formula (w, I) (FIn m M) length I = n (w,
I) ⊨ FIn m M
    unfolding lang_WS1S-def by blast
   hence x-alt: x = map (case-prod (enc-atom I)) (zip [0 ..< length x] (stake
(length x) (w @− sconst any)))
```

by (*intro encD*) *auto*
from *FIn*(*1*) ∗(*2*,*4*) **obtain** *p P* **where** *p*: *I* ! *m* = *Inl p I* ! *M* = *Inr P p* ∈ *P*
by (*auto simp*: *all-set-conv-all-nth split*: *sum.splits*)
with *FIn*(*1*) ∗(*1*,*2*,*3*) **have** *p-less*: *p* < *length x* ∀ *p* ∈ *P*. *p* < *length x*
by (*auto simp del*: *stream-enc.simps intro*: *trans-less-add1*[*OF less-length-cut-same-Inl*]
trans-less-add1[*OF bspec*[*OF less-length-cut-same-Inr*]])
hence *enc-atom*: *x* ! *p* = *enc-atom I p* ((*w* @− *sconst any*) !! *p*) (**is** - = *enc-atom* - - *?p*)
∀ *p* ∈ *P*. *x* ! *p* = *enc-atom I p* ((*w* @− *sconst any*) !! *p*) (**is** *Ball* - (λ*p*. - = *enc-atom* - - (*?P p*)))
by (*subst x-alt*, *simp*)+
with ∗(*1*) *p-less*(*1*) **have** *x* = *take p x* @ [*enc-atom I p ?p*] @ *drop* (*p* + *1*) *x*
using *id-take-nth-drop*[*of p x*] **by** *auto*
moreover
from ∗(*2*,*3*) *FIn*(*1*) *p* **have** [*enc-atom I p ?p*] ∈ *lang n* (*Atom* (*Arbitrary-Except2 m M*))
by (*intro enc-atom-lang-Arbitrary-Except2*) (*auto simp*: *shift-snth*)
moreover from ∗(*2*,*3*) **have** *take p x* ∈ *lang n* (*rexp.Not Zero*)
by (*subst x-alt*) (*auto simp*: *in-set-zip shift-snth intro*!: *enc-atom-σ dest*!: *in-set-takeD*)
moreover from ∗(*2*,*3*) **have** *drop* (*Suc p*) *x* ∈ *lang n* (*rexp.Not Zero*)
by (*subst x-alt*) (*auto simp*: *in-set-zip shift-snth intro*!: *enc-atom-σ dest*!: *in-set-dropD*)
ultimately show *x* ∈ *?L n* (*FIn m M*) **using** ∗(*1*,*2*,*3*)
unfolding *rexp-of.simps lang.simps*(*6*,*9*) *rexp-of-list.simps Int-Diff lang-ENC-formula*[*OF FIn*]
by (*auto elim*: *ssubst simp del*: *o-apply append.simps lang.simps enc.simps*)
next
fix *x* **let** *?x* = *x* @− *sconst* (*any*, *replicate n False*)
assume *x*: *x* ∈ *?L n* (*FIn m M*)
with *FIn* **obtain** *w I* **where**
I: *x* ∈ *enc* (*w*, *I*) *length I* = *n wf-interp-for-formula* (*w*, *I*) (*FIn m M*)
unfolding *rexp-of.simps lang.simps lang-ENC-formula*[*OF FIn*] **by** *fastforce*
hence *stream-enc*: *stream-enc* (*w*, *I*) = *?x* **using** *stream-enc-enc* **by** *auto*
from *I FIn* **obtain** *p P* **where** *m*: *I* ! *m* = *Inl p m* < *length I I* ! *M* = *Inr P M* < *length I*
by (*auto split*: *sum.splits*)
with *I* **have** *wf-interp-for-formula* (*dec-word ?x*, *stream-dec n* {*m*} *?x*) (*FIn m M*) **unfolding** *I*(*1*)
using *enc-wf-interp*[*OF FIn*(*1*)[*folded I*(*2*)]] **by** *auto*
moreover
from *x* **obtain** *u1 u u2* **where** *x* = *u1* @ *u* @ *u2*
u ∈ *lang n* (*Atom* (*Arbitrary-Except2 m M*))
unfolding *rexp-of.simps lang.simps rexp-of-list.simps* **using** *concE* **by** *fast*
with *FIn*(*1*) **obtain** *v* **where** *v*: *x* = *u1* @ [*v*] @ *u2 snd v* ! *m snd v* ! *M* **and** *fst v* ∈ *set* Σ
using *Arbitrary-Except2D*[*of u n m M*] **by** *simp* (*auto simp*: *σ-def*)
from *v* **have** *u*: *length u1* < *length x* **by** *auto*
{ from *v* **have** *snd* (*x* ! *length u1*) ! *m* **by** *auto*

```
    moreover
    from m I have p < length x snd (x ! p) ! m by (auto dest: enc-Inl simp del:
enc.simps)
    moreover
      from m I have ex1: ∃!p. snd (stream-enc (w, I) !! p) ! m by (intro
stream-enc-unique) auto
    ultimately have p = length u1 unfolding stream-enc using u I(3) by auto
  } note * = this
  from v have v = x ! length u1 by simp
  with v(3) m(3,4) u I(1,3) have length u1 ∈ P by (auto dest!: enc-Inr simp
del: enc.simps)
  with * m I have (dec-word ?x, stream-dec n {m} ?x) ⊨ FIn m M
    using stream-enc-enc[OF - I(1), symmetric]
  by (auto simp del: stream-enc.simps dest: stream-dec-not-Inr stream-dec-not-Inl
      stream-dec-enc-Inl stream-dec-enc-Inr split: sum.splits)
  moreover from m I(2)
    have stream-enc-dec: stream-enc (dec-word (stream-enc (w, I)), stream-dec n
{m} (stream-enc (w, I))) = stream-enc (w, I)
      by (intro stream-enc-dec)
        (auto simp: smap2-alt sdrop-snth shift-snth intro: stream-enc-unique,
         auto simp: smap2-szip stream.set-map)
    moreover from I have wf-word n x unfolding wf-word by (auto elim:
enc-set-σ simp del: enc.simps)
  ultimately show x ∈ lang_WS1S n (FIn m M) unfolding lang_WS1S-def using
m I(1,3)
    by (auto simp del: enc.simps stream-enc.simps intro!: exI[of - enc (dec-word
?x, stream-dec n {m} ?x)],
        fastforce simp del: enc.simps stream-enc.simps,
        auto simp del: stream-enc.simps simp: stream-enc[symmetric] I(2))
 qed
next
 case (FOr φ1 φ2)
 from FOr(3) have IH1: lang_WS1S n φ1 = lang n (rexp-of n φ1)
  by (intro FOr(1)) auto
 from FOr(3) have IH2: lang_WS1S n φ2 = lang n (rexp-of n φ2)
  by (intro FOr(2)) auto
 show ?case
 proof (intro equalityI subsetI)
  fix x assume x ∈ lang_WS1S n (FOr φ1 φ2) thus x ∈ lang n (rexp-of n (FOr
φ1 φ2))
   using lang_WS1S-FOr[OF FOr(3)] unfolding lang-ENC-formula[OF FOr(3)]
rexp-of.simps lang.simps IH1 IH2 by blast
 next
  fix x assume x ∈ lang n (rexp-of n (FOr φ1 φ2))
  then obtain w I where or: x ∈ lang_WS1S n φ1 ∨ x ∈ lang_WS1S n φ2 and
I: x ∈ enc (w, I) length I = n
    wf-interp-for-formula (w, I) (FOr φ1 φ2)
    unfolding lang-ENC-formula[OF FOr(3)] rexp-of.simps lang.simps IH1 IH2
Int-Diff by auto
```

```
    have (w, I) ⊨ φ1 ∨ (w, I) ⊨ φ2
    proof (intro mp[OF disj-mono[OF impI impI] or])
      assume x ∈ lang_WS1S n φ1
      with I FOr(3) show (w, I) ⊨ φ1
        unfolding lang_WS1S-def I(1) wf-interp-for-formula-FOr
        by (auto dest!: enc-welldef[of x w I - - φ1] simp del: enc.simps)
    next
      assume x ∈ lang_WS1S n φ2
      with I FOr(3) show (w, I) ⊨ φ2
        unfolding lang_WS1S-def I(1) wf-interp-for-formula-FOr
        by (auto dest!: enc-welldef[of x w I - - φ2] simp del: enc.simps)
    qed
    with I show x ∈ lang_WS1S n (FOr φ1 φ2) unfolding lang_WS1S-def by auto
  qed
next
  case (FAnd φ1 φ2)
  from FAnd(3) have IH1: lang_WS1S n φ1 = lang n (rexp-of n φ1)
    by (intro FAnd(1)) auto
  from FAnd(3) have IH2: lang_WS1S n φ2 = lang n (rexp-of n φ2)
    by (intro FAnd(2)) auto
  show ?case
  proof (intro equalityI subsetI)
   fix x assume x ∈ lang_WS1S n (FAnd φ1 φ2) thus x ∈ lang n (rexp-of n (FAnd
φ1 φ2))
      using lang_WS1S-FAnd[OF FAnd(3)]
       unfolding lang-ENC-formula[OF FAnd(3)] rexp-of.simps rexp-of-list.simps
lang.simps IH1 IH2 Int-assoc
      by blast
  next
    fix x assume x ∈ lang n (rexp-of n (FAnd φ1 φ2))
    then obtain w I where and: x ∈ lang_WS1S n φ1 ∧ x ∈ lang_WS1S n φ2 and
I: x ∈ enc (w, I) length I = n
      wf-interp-for-formula (w, I) (FAnd φ1 φ2)
       unfolding lang-ENC-formula[OF FAnd(3)] rexp-of.simps rexp-of-list.simps
lang.simps IH1 IH2 Int-Diff by auto
    have (w, I) ⊨ φ1 ∧ (w, I) ⊨ φ2
    proof (intro mp[OF conj-mono[OF impI impI] and])
      assume x ∈ lang_WS1S n φ1
      with I FAnd(3) show (w, I) ⊨ φ1
        unfolding lang_WS1S-def I(1) wf-interp-for-formula-FAnd
        by (auto dest!: enc-welldef[of x w I - - φ1] simp del: enc.simps)
    next
      assume x ∈ lang_WS1S n φ2
      with I FAnd(3) show (w, I) ⊨ φ2
        unfolding lang_WS1S-def I(1) wf-interp-for-formula-FAnd
        by (auto dest!: enc-welldef[of x w I - - φ2] simp del: enc.simps)
    qed
     with I show x ∈ lang_WS1S n (FAnd φ1 φ2) unfolding lang_WS1S-def by
auto
```

```
  qed
next
  case (FNot φ)
  hence IH: ?L n φ =  lang_WS1S n φ by simp
  show ?case
  proof (intro equalityI subsetI)
    fix x assume x ∈ lang_WS1S n (FNot φ)
    then obtain w I where
      ∗: x ∈ enc (w, I) wf-interp-for-formula (w, I) φ length I = n and unsat: ¬
(w, I) ⊨ φ
       unfolding lang_WS1S-def by auto
    { assume x ∈ ?L n φ
       hence (w, I) ⊨ φ using enc-welldef[of x w I - - φ, OF ∗(1) - - - ∗(2)]
FNot(2)
        unfolding ∗(3) lang_WS1S-def IH by auto
    }
    with unsat have x ∉ ?L n φ by blast
    with ∗ show x ∈ ?L n (FNot φ) unfolding rexp-of.simps lang.simps using
lang-ENC-formula[OF FNot(2)]
      by (auto simp del: enc.simps simp: comp-def intro: enc-set-σ)
  next
    fix x assume x ∈ ?L n (FNot φ)
    with IH have x ∈ lang n (ENC n (FOV (FNot φ))) and x: x ∉ lang_WS1S n
φ by (auto simp del: o-apply)
    then obtain w I where ∗: x ∈ enc (w, I) wf-interp-for-formula (w, I) (FNot
φ) length I = n
      unfolding lang-ENC-formula[OF FNot(2)] by blast
    { assume ¬ (w, I) ⊨ FNot φ
      with ∗ have x ∈ lang_WS1S n φ unfolding lang_WS1S-def by auto
    }
    with x ∗ show x ∈ lang_WS1S n (FNot φ)  unfolding lang_WS1S-def by blast
  qed
next
  case (FExists φ)
  have σ: (any, replicate n False) ∈ (set o σ Σ) n by (auto simp: σ-def set-n-lists
image-iff)
  from FExists(2) have wf: wf n (Pr (rexp-of (Suc n) φ)) by (fastforce intro:
wf-rexp-of)
  note lang-quot = lang-samequot-exec[OF wf σ]
  show ?case
  proof (intro equalityI subsetI)
    fix x assume x ∈ lang_WS1S n (FExists φ)
    then obtain w I p where
      ∗: x ∈ enc (w, I) wf-interp-for-formula (w, I) (FExists φ) length I = n (w,
Inl p # I) ⊨ φ
       unfolding lang_WS1S-def by auto
    with FExists(2) have enc (w, Inl p # I) ⊆ ?L (Suc n) φ
      by (subst FExists(1)[of Suc n, symmetric])
        (fastforce simp del: enc.simps simp: lang_WS1S-def nth-Cons' intro!: exI[of
```

- *enc* (*w*, *Inl p # I*)])+
  thus *x* ∈ *?L n* (*FExists φ*) **using** ∗(*1*,*2*,*3*)
    by (*auto simp*: *lang-quot simp del*: *o-apply enc.simps elim*: *subsetD*[*OF SAMEQUOT-mono*[*OF image-mono*]])
 next
  fix *x* **assume** *x* ∈ *?L n* (*FExists φ*)
  then obtain *x′ m* **where** *x′* ∈ *?L* (*Suc n*) *φ* **and**
   x: *x* = *fin-cut-same* (*any*, *replicate n False*) (*map π x′*) @ *replicate m* (*any*, *replicate n False*)
    by (*auto simp*: *lang-quot SAMEQUOT-def simp del*: *o-apply enc.simps*)
  with *FExists*(*2*) **have** *x′* ∈ $lang_{WS1S}$ (*Suc n*) *φ*
   by (*intro subsetD*[*OF equalityD2*[*OF FExists*(*1*)], *of Suc n x′*])
    (*auto split*: *if-split-asm sum.splits*)
  then obtain *w I′* **where**
   ∗: *x′* ∈ *enc* (*w*, *I′*) *wf-interp-for-formula* (*w*, *I′*) *φ length I′* = *Suc n* (*w*, *I′*) ⊨ *φ*
    unfolding $lang_{WS1S}$*-def* **by** *blast*
  moreover then obtain I_0 *I* **where** *I′* = I_0 # *I* **by** (*cases I′*) *auto*
  moreover with *FExists*(*2*) ∗(*2*) **obtain** *p* **where** I_0 = *Inl p*
   by (*auto simp*: *nth-Cons′ split*: *sum.splits if-split-asm*)
  ultimately have *x* ∈ *enc* (*w*, *I*) *wf-interp-for-formula* (*w*, *I*) (*FExists φ*) *length I* = *n*
   (*w*, *I*) ⊨ *FExists φ* **using** *FExists*(*2*) *fin-cut-same-tl*[*OF ex-Loop-stream-enc*, *of Inl p # I w*]
    unfolding *x* **by** (*auto simp add*: *π-def nth-Cons′ split*: *if-split-asm*)
  thus *x* ∈ $lang_{WS1S}$ *n* (*FExists φ*) **unfolding** $lang_{WS1S}$*-def* **by** (*auto intro*!: *exI*[*of - I*])
 qed
next
 case (*FEXISTS φ*)
 have *σ*: (*any*, *replicate n False*) ∈ (*set o σ* Σ) *n* **by** (*auto simp*: *σ-def set-n-lists image-iff*)
 from *FEXISTS*(*2*) **have** *wf*: *wf n* (*Pr* (*rexp-of* (*Suc n*) *φ*)) **by** (*fastforce intro*: *wf-rexp-of*)
 note *lang-quot* = *lang-samequot-exec*[*OF wf σ*]
 show *?case*
 proof (*intro equalityI subsetI*)
  fix *x* **assume** *x* ∈ $lang_{WS1S}$ *n* (*FEXISTS φ*)
  then obtain *w I P* **where**
   ∗: *x* ∈ *enc* (*w*, *I*) *wf-interp-for-formula* (*w*, *I*) (*FEXISTS φ*) *length I* = *n* *finite P* (*w*, *Inr P # I*) ⊨ *φ*
    unfolding $lang_{WS1S}$*-def* **by** *auto*
  with *FEXISTS*(*2*) **have** *enc* (*w*, *Inr P # I*) ⊆ *?L* (*Suc n*) *φ*
   by (*subst FEXISTS*(*1*)[*of Suc n*, *symmetric*])
    (*fastforce simp del*: *enc.simps simp*: $lang_{WS1S}$*-def nth-Cons′ intro*!: *exI*[*of - enc* (*w*, *Inr P # I*)])+
  thus *x* ∈ *?L n* (*FEXISTS φ*) **using** ∗(*1*,*2*,*3*,*4*)
    by (*auto simp*: *lang-quot simp del*: *o-apply enc.simps elim*: *subsetD*[*OF SAMEQUOT-mono*[*OF image-mono*]])

next
fix x **assume** $x \in ?L\ n\ (FEXISTS\ \varphi)$
then obtain $x'\ m$ **where** $x' \in ?L\ (Suc\ n)\ \varphi$ **and**
x: $x = fin\text{-}cut\text{-}same\ (any,\ replicate\ n\ False)\ (map\ \pi\ x')\ @\ replicate\ m\ (any,\ replicate\ n\ False)$
by (*auto simp*: *lang-quot SAMEQUOT-def simp del*: *o-apply enc.simps*)
with $FEXISTS(2)$ **have** $x' \in lang_{WS1S}\ (Suc\ n)\ \varphi$
by (*intro subsetD*[*OF equalityD2*[*OF FEXISTS*(*1*)], *of Suc n x'*])
(*auto split*: *if-split-asm sum.splits*)
then obtain $w\ I'$ **where**
$*$: $x' \in enc\ (w,\ I')\ wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I')\ \varphi\ length\ I' = Suc\ n\ (w,\ I') \models \varphi$
unfolding $lang_{WS1S}\text{-}def$ **by** *blast*
moreover then obtain $I_0\ I$ **where** $I' = I_0\ \#\ I$ **by** (*cases I'*) *auto*
moreover with $FEXISTS(2)\ *(2)$ **obtain** P **where** $I_0 = Inr\ P\ finite\ P$
by (*auto simp*: *nth-Cons' split*: *sum.splits if-split-asm*)
ultimately have $x \in enc\ (w,\ I)\ wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I)\ (FEXISTS\ \varphi)\ length\ I = n$
$(w, I) \models FEXISTS\ \varphi$ **using** $FEXISTS(2)$ *fin-cut-same-tl*[*OF ex-Loop-stream-enc, of Inr P # I*]
unfolding x **by** (*auto simp*: *nth-Cons' π-def split*: *if-split-asm*)
thus $x \in lang_{WS1S}\ n\ (FEXISTS\ \varphi)$ **unfolding** $lang_{WS1S}\text{-}def$ **by** (*auto intro*!: *exI*[*of - I*])
qed
qed

lemma *wf-rexp-of-alt*: *wf-formula n φ ⟹ wf n* (*rexp-of-alt n φ*)
by (*induct φ arbitrary*: *n*)
(*auto intro*!: *wf-samequot-exec wf-rexp-ENC*,
auto simp: *max-idx-vars finite-FOV*)

lemma *wf-rexp-of'*: *wf-formula n φ ⟹ wf n* (*rexp-of' n φ*)
unfolding *rexp-of'-def* **by** (*auto simp*: *max-idx-vars intro*: *wf-rexp-of-alt wf-rexp-ENC*[*OF finite-FOV*])

lemma *wf-rexp-of-alt'*: *wf-formula n φ ⟹ wf n* (*rexp-of-alt' n φ*)
by (*induct φ arbitrary*: *n*)
(*auto intro*!: *wf-samequot-exec wf-rexp-ENC*,
auto simp: *max-idx-vars finite-FOV*)

lemma *wf-rexp-of''*: *wf-formula n φ ⟹ wf n* (*rexp-of'' n φ*)
unfolding *rexp-of''-def* **by** (*auto simp*: *wf-rexp-ENC wf-rexp-of-alt' finite-FOV max-idx-vars*)

lemma *ENC-FNot*: *ENC n* (*FOV* (*FNot φ*)) = *ENC n* (*FOV φ*)
unfolding *ENC-def* **by** *auto*

lemma *ENC-FAnd*:

wf-formula n (FAnd φ ψ) ⟹ lang n (ENC n (FOV (FAnd φ ψ))) ⊆ lang n (ENC n (FOV φ)) ∩ lang n (ENC n (FOV ψ))
proof
fix *x* **assume** *wf*: *wf-formula n (FAnd φ ψ)* **and** *x*: *x ∈ lang n (ENC n (FOV (FAnd φ ψ)))*
hence *wf1*: *wf-formula n φ* **and** *wf2*: *wf-formula n ψ* **by** *auto*
from *x* **obtain** *w I* **where** *I*: *x ∈ enc (w, I) wf-interp-for-formula (w, I) (FAnd φ ψ) length I = n*
using *lang-ENC-formula*[*OF wf*] **by** *auto*
hence *wf-interp-for-formula (w, I) φ wf-interp-for-formula (w, I) ψ*
using *wf-interp-for-formula-FAnd* **by** *auto*
thus *x ∈ lang n (ENC n (FOV φ)) ∩ lang n (ENC n (FOV ψ))*
unfolding *lang-ENC-formula*[*OF wf1*] *lang-ENC-formula*[*OF wf2*] **using** *I* **by** *blast*
qed

lemma *ENC-FOr*:
wf-formula n (FOr φ ψ) ⟹ lang n (ENC n (FOV (FOr φ ψ))) ⊆ lang n (ENC n (FOV φ)) ∩ lang n (ENC n (FOV ψ))
proof
fix *x* **assume** *wf*: *wf-formula n (FOr φ ψ)* **and** *x*: *x ∈ lang n (ENC n (FOV (FOr φ ψ)))*
hence *wf1*: *wf-formula n φ* **and** *wf2*: *wf-formula n ψ* **by** *auto*
from *x* **obtain** *w I* **where** *I*: *x ∈ enc (w, I) wf-interp-for-formula (w, I) (FOr φ ψ) length I = n*
using *lang-ENC-formula*[*OF wf*] **by** *auto*
hence *wf-interp-for-formula (w, I) φ wf-interp-for-formula (w, I) ψ*
using *wf-interp-for-formula-FOr* **by** *auto*
thus *x ∈ lang n (ENC n (FOV φ)) ∩ lang n (ENC n (FOV ψ))*
unfolding *lang-ENC-formula*[*OF wf1*] *lang-ENC-formula*[*OF wf2*] **using** *I* **by** *blast*
qed

lemma *ENC-FExists*:
wf-formula n (FExists φ) ⟹ lang n (ENC n (FOV (FExists φ))) =
SAMEQUOT (any, replicate n False) (map π ‘ lang (Suc n) (ENC (Suc n) (FOV φ))) (**is** *- ⟹ ?L = ?R*)
proof (*intro equalityI subsetI*)
fix *x* **assume** *wf*: *wf-formula n (FExists φ)* **and** *x*: *x ∈ ?L*
hence *wf1*: *wf-formula (Suc n) φ* **by** *auto*
from *x* **obtain** *w I* **where** *I*: *x ∈ enc (w, I) wf-interp-for-formula (w, I) (FExists φ) length I = n*
using *lang-ENC-formula*[*OF wf*] **by** *auto*
from *I*(*2*) **obtain** *p* **where** *wf-interp-for-formula (w, Inl p # I) φ*
using *wf-interp-for-formula-FExists*[*OF wf*[*folded I*(*3*)]] **by** *blast*
with *I*(*3*) **show** *x ∈ ?R*
unfolding *lang-ENC-formula*[*OF wf1*] **using** *I*(*1*) *tl-enc*[*of Inl p I*, *symmetric*]
by (*simp del*: *enc.simps*)
(*fastforce simp del*: *enc.simps elim*!: *rev-subsetD*[*OF - SAMEQUOT-mono*[*OF*

image-mono]]
intro: *exI*[*of* - *enc* (*w*, *Inl p* # *I*)])
next
fix *x* **assume** *wf*: *wf-formula n* (*FExists* φ) **and** *x*: $x \in ?R$
hence *wf1*: *wf-formula* (*Suc n*) φ **and** $0 \in FOV\ \varphi$ **by** *auto*
from *x* **obtain** *w I* **where** *I*: $x \in SAMEQUOT$ (*any*, *replicate n False*) (*map* π ‘ *enc* (*w*, *I*))
wf-interp-for-formula (*w*, *I*) φ *length I* = *Suc n*
using *lang-ENC-formula*[*OF wf1*] **unfolding** *SAMEQUOT-def* **by** *fast*
with ‹$0 \in FOV\ \varphi$› **obtain** *p I′* **where** *I′*: $I = Inl\ p\ \#\ I'$ **by** (*cases I*) (*fastforce split*: *sum.splits*)+
with *I* **have** *wtlI*: $x \in enc\ (w, I')$ *length I′* = *n* **using** *tl-enc*[*of Inl p I′ w*] **by** *auto*
have *wf-interp-for-formula* (*w*, *I′*) (*FExists* φ)
using *wf-interp-for-formula-FExists*[*OF wf*[*folded wtlI*(*2*)]]
wf-interp-for-formula-any-Inl[*OF I*(*2*)[*unfolded I′*]] **..**
with *wtlI* **show** $x \in ?L$ **unfolding** *lang-ENC-formula*[*OF wf*] **by** *blast*
qed

lemma *ENC-FEXISTS*:
wf-formula n (*FEXISTS* φ) $\Longrightarrow$ *lang n* (*ENC n* (*FOV* (*FEXISTS* φ))) =
SAMEQUOT (*any*, *replicate n False*) (*map* π ‘ *lang* (*Suc n*) (*ENC* (*Suc n*) (*FOV* φ))) (**is** - $\Longrightarrow$ *?L* = *?R*)
proof (*intro equalityI subsetI*)
fix *x* **assume** *wf*: *wf-formula n* (*FEXISTS* φ) **and** *x*: $x \in ?L$
hence *wf1*: *wf-formula* (*Suc n*) φ **by** *auto*
from *x* **obtain** *w I* **where** *I*: $x \in enc\ (w, I)$ *wf-interp-for-formula* (*w*, *I*) (*FEXISTS* φ) *length I* = *n*
using *lang-ENC-formula*[*OF wf*] **by** *auto*
from *I*(*2*) **obtain** *P* **where** *wf-interp-for-formula* (*w*, *Inr P* # *I*) φ
using *wf-interp-for-formula-FEXISTS*[*OF wf*[*folded I*(*3*)]] **by** *blast*
with *I*(*3*) **show** $x \in ?R$
unfolding *lang-ENC-formula*[*OF wf1*] **using** *I*(*1*) *tl-enc*[*of Inr P I*, *symmetric*]
by (*simp del*: *enc.simps*)
(*fastforce simp del*: *enc.simps elim*!: *rev-subsetD*[*OF* - *SAMEQUOT-mono*[*OF image-mono*]]
intro: *exI*[*of* - *enc* (*w*, *Inr P* # *I*)])
next
fix *x* **assume** *wf*: *wf-formula n* (*FEXISTS* φ) **and** *x*: $x \in ?R$
hence *wf1*: *wf-formula* (*Suc n*) φ **and** $0 \in SOV\ \varphi$ **by** *auto*
from *x* **obtain** *w I* **where** *I*: $x \in SAMEQUOT$ (*any*, *replicate n False*) (*map* π ‘ *enc* (*w*, *I*))
wf-interp-for-formula (*w*, *I*) φ *length I* = *Suc n*
using *lang-ENC-formula*[*OF wf1*] **unfolding** *SAMEQUOT-def* **by** *fast*
with ‹$0 \in SOV\ \varphi$› **obtain** *P I′* **where** *I′*: $I = Inr\ P\ \#\ I'$ **by** (*cases I*) (*fastforce split*: *sum.splits*)+
with *I* **have** *wtlI*: $x \in enc\ (w, I')$ *length I′* = *n* **using** *tl-enc*[*of Inr P I′ w*] **by** *auto*
have *wf-interp-for-formula* (*w*, *I′*) (*FEXISTS* φ)

using *wf-interp-for-formula-FEXISTS*[*OF wf*[*folded wtlI*(*2*)]]
wf-interp-for-formula-any-Inr[*OF I*(*2*)[*unfolded I'*]] **..**
with *wtlI* **show** $x \in ?L$ **unfolding** *lang-ENC-formula*[*OF wf*] **by** *blast*
qed

lemma $lang_{WS1S}$*-rexp-of-rexp-of'*:
wf-formula n $\varphi \Longrightarrow$ *lang n* (*rexp-of n* φ) = *lang n* (*rexp-of' n* φ)
unfolding *rexp-of'-def* **proof** (*induction* φ *arbitrary*: *n*)
case (*FNot* φ)
hence *wf-formula n* φ **by** *simp*
with *FNot.IH* **show** *?case* **unfolding** *rexp-of.simps rexp-of-alt.simps lang.simps ENC-FNot* **by** *blast*
next
case (*FAnd* φ_1 φ_2)
hence *wf1*: *wf-formula n* φ_1 **and** *wf2*: *wf-formula n* φ_2 **by** *force+*
from *FAnd.IH*(*1*)[*OF wf1*] *FAnd.IH*(*2*)[*OF wf2*] **show** *?case* **using** *ENC-FAnd*[*OF FAnd.prems*]
unfolding *rexp-of.simps rexp-of-alt.simps lang.simps rexp-of-list.simps* **by** *blast*
next
case (*FOr* φ_1 φ_2)
hence *wf1*: *wf-formula n* φ_1 **and** *wf2*: *wf-formula n* φ_2 **by** *force+*
from *FOr.IH*(*1*)[*OF wf1*] *FOr.IH*(*2*)[*OF wf2*] **show** *?case* **using** *ENC-FOr*[*OF FOr.prems*]
unfolding *rexp-of.simps rexp-of-alt.simps lang.simps* **by** *blast*
next
case (*FExists* φ)
from *FExists*(*2*) **have** *IH*: *lang* (*n* + *1*) (*rexp-of* (*n* + *1*) φ) =
lang (*n* + *1*) (*Inter* (*rexp-of-alt* (*n* + *1*) φ) (*ENC* (*n* + *1*) (*FOV* φ))) **by** (*intro FExists.IH*) *auto*
have σ: (*any*, *replicate n False*) $\in$ (*set o* σ Σ) *n* **by** (*auto simp*: σ*-def set-n-lists image-iff*)
from *FExists*(*2*) **have** *wf*: *wf n* (*Pr* (*rexp.Inter* (*rexp-of-alt* (*n* + *1*) φ) (*ENC* (*n* + *1*) (*FOV* φ))))
wf n (*Pr* (*rexp-of* (*n* + *1*) φ)) **by** (*fastforce simp*: *max-idx-vars intro*!: *wf-rexp-of wf-rexp-of-alt wf-rexp-ENC*[*OF finite-FOV*])+
note *lang-quot* = *lang-samequot-exec*[*OF wf*(*1*) σ] *lang-samequot-exec*[*OF wf*(*2*) σ]
show *?case* **unfolding** *rexp-of.simps rexp-of-alt.simps lang.simps IH lang-quot Suc-eq-plus1*
ENC-FExists[*OF FExists.prems*, *unfolded Suc-eq-plus1*] **by** (*auto simp add*: *SAMEQUOT-def*)
next
case (*FEXISTS* φ)
from *FEXISTS*(*2*) **have** *IH*: *lang* (*n* + *1*) (*rexp-of* (*n* + *1*) φ) =
lang (*n* + *1*) (*Inter* (*rexp-of-alt* (*n* + *1*) φ) (*ENC* (*n* + *1*) (*FOV* φ))) **by** (*intro FEXISTS.IH*) *auto*
have σ: (*any*, *replicate n False*) $\in$ (*set o* σ Σ) *n* **by** (*auto simp*: σ*-def set-n-lists image-iff*)
from *FEXISTS*(*2*) **have** *wf*: *wf n* (*Pr* (*rexp.Inter* (*rexp-of-alt* (*n* + *1*) φ) (*ENC*

$(n + 1)$ $(FOV\ \varphi)))$
wf n (Pr (rexp-of (n + 1) φ)) **by** (*fastforce simp*: *max-idx-vars intro*: *wf-rexp-of wf-rexp-of-alt wf-rexp-ENC*[*OF finite-FOV*])+
note *lang-quot* = *lang-samequot-exec*[*OF wf*(*1*) σ] *lang-samequot-exec*[*OF wf*(*2*) σ]
show *?case* **unfolding** *rexp-of.simps rexp-of-alt.simps lang.simps IH lang-quot Suc-eq-plus1*
ENC-FEXISTS[*OF FEXISTS.prems*, *unfolded Suc-eq-plus1*] **by** (*auto simp add*: *SAMEQUOT-def*)
qed *auto*

lemma *SAMEQUTO-UN*[*simp*]: *SAMEQUOT x* $(\bigcup y \in A.\ B\ y)$ = $(\bigcup y \in A.$ *SAMEQUOT x* $(B\ y))$
unfolding *SAMEQUOT-def* **by** *auto*

lemma *finite-positions-in-row*[*simp*]:
$n > 0 \Longrightarrow$ *finite* (*positions-in-row* (*x @− sconst* (*any*, *replicate n False*)) *0*)
unfolding *positions-in-row shift-snth* **by** *auto*

lemma *fin-cut-same-snoc*: *fin-cut-same x* (*xs* @ [*y*]) = (*if x* = *y then fin-cut-same x xs else xs* @ [*y*])
by (*induct xs*) *auto*

lemma *fin-cut-same-idem*: *fin-cut-same x* (*fin-cut-same x xs*) = *fin-cut-same x xs*
by (*induct xs*) *auto*

lemma *cut-same-sconst*: *cut-same x* (*xs @− sconst x*) = *fin-cut-same x xs*
proof (*induct xs rule*: *rev-induct*)
case (*snoc y ys*)
then show *?case* **by** (*auto simp del*: *id-apply simp add*: *fin-cut-same-snoc sconst-collapse*)
qed (*simp del*: *id-apply*)

lemma *length-cut-same*: *length* (*cut-same x s*) = (*LEAST n. sdrop n s* = *sconst x*)
unfolding *cut-same-def* **by** *simp*

lemma *enc-alt*: *wf-interp w I* $\Longrightarrow$
x ∈ *enc* (*w*, *I*) $\longleftrightarrow$ *x @− sconst* ((*any*, *replicate* (*length I*) *False*)) = *stream-enc* (*w*, *I*)
unfolding *enc.simps*
by (*force simp only*: *shift-append shift-replicate-sconst stream-enc-cut-same*[*symmetric*]
length-append length-replicate length-cut-same sdrop-shift drop-all diff-self-eq-0
shift.simps sdrop.simps
dest: *sym*[*of - stream-enc* (*w*, *I*)]
intro: *shift-sconst-inj*[*rotated*, *of -* (*any*, *replicate* (*length I*) *False*)] *Least-le*
exI[*of - length x* − *length* (*cut-same* (*any*, *replicate* (*length I*) *False*) (*stream-enc* (*w*, *I*)))]
le-add-diff-inverse[*symmetric*])

lemma *stream-stream-eqI*: ⟦∀(-, *x*) ∈ *sset xs*. *x* ≠ []; ∀(-, *x*) ∈ *sset ys*. *x* ≠ []; *smap* (λ(-, *x*). *hd x*) *xs* = *smap* (λ(-, *x*). *hd x*) *ys*; *smap π xs* = *smap π ys*⟧ ⟹ *xs* = *ys*
proof (*coinduction arbitrary*: *xs ys*)
case *Eq-stream*
then show *?case*
proof (*cases xs ys rule*: *stream.exhaust*[*case-product stream.exhaust*])
case (*SCons-SCons h1 t1 h2 t2*)
with *Eq-stream* **show** *?thesis*
by (*cases snd h1 snd h2 rule*: *list.exhaust*[*case-product list.exhaust*])
(*auto simp*: *π-def split*: *prod.splits*)
qed
qed

lemma *project-enc-extend*:
fixes *x I*
defines *n* ≡ *length I*
defines *z* ≡ λ*n*. (*any*, *replicate n False*)
defines *I′* ≡ *Inr* (*positions-in-row* (*x* @− *sconst* (*z* (*Suc n*))) *0*) # *I*
assumes *wf*: *wf-interp w I*
assumes *enc*: *fin-cut-same* (*z n*) (*map π x*) @ *replicate m* (*z n*) ∈ *enc* (*w*, *I*)
assumes *nonempty*: ∀(-, *x*) ∈ *set x*. *x* ≠ []
shows *x* ∈ *enc* (*w*, *I′*)
proof −
have [*simp*]: *π* (*z* (*Suc n*)) = *z n*
and *z-def*: ⋀*n*. *z n* = (*any*, *replicate n False*) **unfolding** *π-def z-def* **by** *auto*
have *wf′*: *wf-interp w I′* **by** (*simp add*: *wf I′-def z-def del*: *replicate-Suc*)
note *simps*[*simp del*] = *stream-enc.simps*
show *?thesis* **unfolding** *enc-alt*[*OF wf′*]
proof (*rule stream-stream-eqI*)
from *nonempty stream-smap-nats*[*of map* (λ(-, *y*). *hd y*) *x* @− *sconst False*] *smap-szip-fst*
show *smap* (λ(-, *x*). *hd x*) (*x* @− *sconst* (*any*, *replicate* (*length I′*) *False*)) = *smap* (λ(-, *x*). *hd x*) (*stream-enc* (*w*, *I′*))
by (*auto simp add*: *stream-enc.simps I′-def z-def smap2-szip stream.map-comp o-def split-def positions-in-row shift-snth hd-conv-nth intro*: *smap-szip-fst*[*symmetric*] *cong*: *stream.map-cong*)
next
from *wf* **have** *fin-cut-same* (*z n*) (*map π x*) = *cut-same* (*z n*) (*stream-enc* (*w*, *I*))
using *stream-enc-enc*[*OF - enc*] **by** (*auto simp add*: *cut-same-sconst z-def n-def fin-cut-same-idem*)
then obtain *m′* **where** *πx*: *map π x* = *cut-same* (*z n*) (*stream-enc* (*w*, *I*)) @ *replicate m′* (*z n*)
by (*auto dest*!: *fin-cut-sameE*)
with *wf* **show** *smap π* (*x* @− *sconst* (*any*, *replicate* (*length I′*) *False*)) = *smap π* (*stream-enc* (*w*, *I′*))

```
    by (simp del: replicate-Suc add: n-def[symmetric] z-def[symmetric] I'-def
      stream-enc-cut-same[of I, symmetric, folded n-def z-def])
 qed (insert nonempty, simp-all add: stream-enc.simps I'-def split-beta smap2-szip
stream.set-map)
qed

lemma pred-case-conv: x - Suc 0 = (case x of 0 ⇒ 0 | Suc m ⇒ m)
  by (cases x) auto

lemma in-pred-image-iff: 0 ∉ X ⟹ (x ∈ (λx. x - Suc 0) ` X) = (Suc x ∈ X)
  by (auto simp: pred-case-conv split: nat.splits)

lemma map-project-Int-ENC:
  fixes X Z n
  defines z ≡ (any, replicate n False)
  assumes 0 ∉ X X ⊆ {0 ..< n + 1} Z ⊆ lists ((set o σ Σ) (n + 1))
  shows SAMEQUOT z (map π ` (Z ∩ lang (n + 1) (ENC (n + 1) X))) =
    SAMEQUOT z (map π ` Z) ∩ lang n (ENC n ((λx. x - 1) ` X))
proof -
  let ?Y = {0 ..< n + 1} - X
  let ?fX = (λx. x - 1) ` X
  let ?fY = {0 ..< n} - (λx. x - 1) ` X
  from assms have *: (λx. x - 1) ` X ⊆ {0 ..< n} by (cases n) auto
  show ?thesis
 proof (safe elim!: subsetD[OF SAMEQUOT-mono[OF subset-trans[OF image-Int-subset
Int-lower1]]])
    fix w assume w ∈ SAMEQUOT z (map π ` (Z ∩ lang (n + 1) (ENC (n +
1) X)))
    then have w ∈ SAMEQUOT z (map π ` lang (n + 1) (ENC (n + 1) X))
      by (rule rev-subsetD[OF - SAMEQUOT-mono]) auto
    with assms(2) show w ∈ lang n (ENC n ((λx. x - 1) ` X))
      unfolding lang-ENC[OF assms(3) subset-refl] lang-ENC[OF * subset-refl]
      by (auto simp: image-Union z-def length-Suc-conv simp del: enc.simps
        intro!: exI[of - enc (w, I) for w I, OF conjI[of - x ∈ A for x A]])
        (fastforce simp: nth-Cons image-iff split: nat.splits sum.splits)
  next
    fix w assume w ∈ SAMEQUOT z (map π ` Z) w ∈ lang n (ENC n ((λx. x
- 1) ` X))
    then show w ∈ SAMEQUOT z (map π ` (Z ∩ lang (n + 1) (ENC (n + 1)
X)))
    unfolding  z-def SAMEQUOT-def proof (safe, intro exI conjI)
      fix m x
      assume πx: fin-cut-same (any, replicate n False) (map π x) @
        replicate m (any, replicate n False) ∈ lang n (ENC n ((λx. x - 1) ` X))
and x ∈ Z
      show map π x ∈ map π ` (Z ∩ lang (n + 1) (ENC (n + 1) X))
      proof (intro imageI IntI)
        from ‹x ∈ Z› assms(4) have ∀(-, x)∈set x. x ≠ [] by (auto simp: σ-def)
        with πx assms(2) show x ∈ lang (n + 1) (ENC (n + 1) X)
```

```
      unfolding lang-ENC[OF assms(3) subset-refl] lang-ENC[OF * subset-refl]
        proof (safe, intro UnionI[OF - project-enc-extend[rotated]] CollectI exI
conjI)
        fix w and I :: (nat + nat set) list
        assume Ball (set I) (case-sum (λa. True) finite)
        then show Ball (set
            (Inr (positions-in-row (x @− sconst (any, replicate (Suc (length I))
False)) 0) #I))
         (case-sum (λa. True) finite) by (auto simp del: replicate-Suc)
       qed (auto simp add: nth-Cons' Ball-def in-pred-image-iff)
     qed (rule ‹x ∈ Z›)
   qed (rule refl)
  qed
qed

lemma lang-ENC-split:
  assumes finite X X = Y1 ∪ Y2 n = 0 ∨ (∀ p ∈ X. p < n)
  shows lang n (ENC n X) = lang n (ENC n Y1) ∩ lang n (ENC n Y2)
 unfolding ENC-def lang-INTERSECT using assms lang-subset-lists[OF wf-rexp-valid-ENC,
of n] by auto

lemma map-project-ENC:
  fixes n
  assumes X ⊆ {0 ..< n + 1} Z ⊆ lists ((set o σ Σ) (n + 1))
  defines z ≡ (any, replicate n False)
  shows SAMEQUOT z (map π ‘ (Z ∩ lang (n + 1) (ENC (n + 1) X))) =
    (if 0 ∈ X
    then SAMEQUOT z (map π ‘ (Z ∩ lang (n + 1) (ENC (n + 1) {0}))) ∩ lang
n (ENC n ((λx. x − 1) ‘ (X − {0})))
    else SAMEQUOT z (map π ‘ Z) ∩ lang n (ENC n ((λx. x − 1) ‘ (X − {0}))))
    (is ?L = (if - then ?R1 else ?R2))
proof (split if-splits, intro conjI impI)
  assume 0: 0 ∉ X
  from assms have fin: finite X finite ((λx. x − 1) ‘ X)
    by (auto elim: finite-subset intro!: finite-imageI[of X])
  from 0 show ?L = ?R2 using map-project-Int-ENC[OF 0 assms(1,2)]
    unfolding lists-image[symmetric] π-σ
      Int-absorb1[OF lang-subset-lists[OF wf-rexp-ENC[OF fin(1)]], of n + 1]
     Int-absorb1[OF lang-subset-lists[OF wf-rexp-ENC[OF fin(2)]], of n] unfolding
z-def
    by auto
next
  assume 0 ∈ X
  hence 0: 0 ∉ X − {0} and X: X = {0} ∪ (X − {0}) by auto
  from assms have fin: finite X
    by (auto elim: finite-subset intro!: finite-imageI[of X])
  have ?L = SAMEQUOT z (map π ‘ ((Z ∩ lang (n + 1) (ENC (n + 1) {0}))
∩ lang (n + 1) (ENC (n + 1) (X − {0}))))
    unfolding Int-assoc z-def using assms by (subst lang-ENC-split[OF fin X, of
```

```
n + 1]) auto
  also have ... = ?R1 unfolding z-def
    using assms(1,2) by (intro map-project-Int-ENC) auto
  finally show ?L = ?R1 .
qed

lemma lang_M2L-rexp-of'-rexp-of'':
  wf-formula n φ ⟹ lang n (rexp-of' n φ) = lang n (rexp-of'' n φ)
unfolding rexp-of'-def rexp-of''-def
proof (induction φ arbitrary: n)
  case (FNot φ)
  hence wf-formula n φ by simp
 with FNot.IH show ?case unfolding rexp-of-alt.simps rexp-of-alt'.simps lang.simps
ENC-FNot by blast
next
  case (FAnd φ₁ φ₂)
  hence wf1: wf-formula n φ₁ and wf2: wf-formula n φ₂ by force+
 from FAnd.IH(1)[OF wf1] FAnd.IH(2)[OF wf2] show ?case using ENC-FAnd[OF
FAnd.prems]
    unfolding rexp-of-alt.simps rexp-of-alt'.simps lang.simps rexp-of-list.simps by
blast
next
  case (FOr φ₁ φ₂)
  hence wf1: wf-formula n φ₁ and wf2: wf-formula n φ₂ by force+
 from FOr.IH(1)[OF wf1] FOr.IH(2)[OF wf2] show ?case using ENC-FOr[OF
FOr.prems]
    unfolding rexp-of-alt.simps rexp-of-alt'.simps lang.simps rexp-of-list.simps by
blast
next
  case (FExists φ)
  hence wf: wf-formula (n + 1) φ and 0: 0 ∈ FOV φ by auto
  then show ?case
    using max-idx-vars[of n + 1 φ] wf-rexp-of-alt'[OF wf]
    unfolding rexp-of-alt.simps rexp-of-alt'.simps lang.simps Suc-eq-plus1
    proof (subst (1 2) lang-samequot-exec)
      show SAMEQUOT (any, replicate n False)
          (lang n (Pr (Inter (rexp-of-alt (n + 1) φ) (ENC (n + 1) (FOV φ))))) ∩
          lang n (ENC n (FOV (FExists φ))) =
        SAMEQUOT (any, replicate n False)
          (lang n (Pr (Inter (rexp-of-alt' (n + 1) φ) (ENC (n + 1) {0})))) ∩
          lang n (ENC n (FOV (FExists φ)))
        using wf 0 max-idx-vars[of n + 1 φ] wf-rexp-of-alt'[OF wf]
        unfolding lang.simps FExists.IH[OF wf, unfolded lang.simps]
        by (subst (1) map-project-ENC) (auto dest: subsetD[OF lang-subset-lists])
    qed (auto simp add: wf-rexp-of-alt finite-FOV wf-rexp-ENC)
next
  case (FEXISTS φ)
  hence wf: wf-formula (n + 1) φ and 0: 0 ∉ FOV φ by auto
  then show ?case
```

using *max-idx-vars*[*of* $n + 1\ \varphi$] *wf-rexp-of-alt'*[*OF wf*]
unfolding *rexp-of-alt.simps rexp-of-alt'.simps lang.simps Suc-eq-plus1*
proof (*subst* (*1 2*) *lang-samequot-exec*)
show *SAMEQUOT* (*any*, *replicate n False*)
(*lang n* (*Pr* (*Inter* (*rexp-of-alt* (*n* + *1*) φ) (*ENC* (*n* + *1*) (*FOV* φ))))) ∩
lang n (*ENC n* (*FOV* (*FEXISTS* φ))) =
SAMEQUOT (*any*, *replicate n False*)
(*lang n* (*Pr* (*rexp-of-alt'* (*n* + *1*) φ))) ∩
lang n (*ENC n* (*FOV* (*FEXISTS* φ)))
using *wf 0 max-idx-vars*[*of* $n + 1\ \varphi$] *wf-rexp-of-alt'*[*OF wf*]
unfolding *lang.simps FEXISTS.IH*[*OF wf*, *unfolded lang.simps*]
by (*subst* (*1*) *map-project-ENC*) (*auto dest*: *subsetD*[*OF lang-subset-lists*])
qed (*auto simp add*: *wf-rexp-of-alt finite-FOV wf-rexp-ENC*)
qed *simp-all*

theorem $lang_{WS1S}$*-rexp-of'*: *wf-formula n* $\varphi \Longrightarrow lang_{WS1S}\ n\ \varphi = lang\ n$ (*rexp-of'* *n* φ)
unfolding $lang_{WS1S}$*-rexp-of-rexp-of'*[*symmetric*] **by** (*rule* $lang_{WS1S}$*-rexp-of*)

theorem $lang_{WS1S}$*-rexp-of''*: *wf-formula n* $\varphi \Longrightarrow lang_{WS1S}\ n\ \varphi = lang\ n$ (*rexp-of''* *n* φ)
unfolding $lang_{M2L}$*-rexp-of'-rexp-of''*[*symmetric*] **by** (*rule* $lang_{WS1S}$*-rexp-of'*)

end

14 Normalization of WS1S Formulas

fun *nNot* **where**
nNot (*FNot* φ) = φ
| *nNot* (*FAnd* $\varphi 1$ $\varphi 2$) = *FOr* (*nNot* $\varphi 1$) (*nNot* $\varphi 2$)
| *nNot* (*FOr* $\varphi 1$ $\varphi 2$) = *FAnd* (*nNot* $\varphi 1$) (*nNot* $\varphi 2$)
| *nNot* φ = *FNot* φ

primrec *norm* **where**
norm (*FQ a m*) = *FQ a m*
| *norm* (*FLess m n*) = *FLess m n*
| *norm* (*FIn m M*) = *FIn m M*
| *norm* (*FOr* φ ψ) = *FOr* (*norm* φ) (*norm* ψ)
| *norm* (*FAnd* φ ψ) = *FAnd* (*norm* φ) (*norm* ψ)
| *norm* (*FNot* φ) = *nNot* (*norm* φ)
| *norm* (*FExists* φ) = *FExists* (*norm* φ)
| *norm* (*FEXISTS* φ) = *FEXISTS* (*norm* φ)

context *formula*
begin

lemma *satisfies-nNot*[*simp*]: (*w*, *I*) ⊨ *nNot* $\varphi \longleftrightarrow$ (*w*, *I*) ⊨ *FNot* φ

by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *FOV-nNot*[*simp*]: *FOV* (*nNot* φ) = *FOV* (*FNot* φ)
by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *SOV-nNot*[*simp*]: *SOV* (*nNot* φ) = *SOV* (*FNot* φ)
by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *pre-wf-formula-nNot*[*simp*]: *pre-wf-formula* n (*nNot* φ) = *pre-wf-formula* n (*FNot* φ)
by (*induct* φ *rule*: *nNot.induct*) *auto*

lemma *FOV-norm*[*simp*]: *FOV* (*norm* φ) = *FOV* φ
by (*induct* φ) *auto*

lemma *SOV-norm*[*simp*]: *SOV* (*norm* φ) = *SOV* φ
by (*induct* φ) *auto*

lemma *pre-wf-formula-norm*[*simp*]: *pre-wf-formula* n (*norm* φ) = *pre-wf-formula* n φ
by (*induct* φ *arbitrary*: n) *auto*

lemma *satisfies-norm*[*simp*]: $wI \models$ *norm* $\varphi \longleftrightarrow wI \models \varphi$
by (*induct* φ *arbitrary*: wI) *auto*

lemma $lang_{WS1S}$-*norm*[*simp*]: $lang_{WS1S}$ n (*norm* φ) = $lang_{WS1S}$ n φ
unfolding $lang_{WS1S}$-*def* **by** *auto*

end

15 Deciding Equivalence of WS1S Formulas

global-interpretation *embed2 set o* σ Σ *wf-atom* Σ π *lookup* ε Σ *case-prod Singleton*
for Σ :: *'a* :: *linorder list*
defines
$\mathfrak{D}$ = *embed.lderiv lookup* (ε Σ)
and *Co*$\mathfrak{D}$ = *embed.lderiv-dual lookup* (ε Σ)
and *r*$\mathfrak{D}$ = *embed.rderiv lookup* (ε Σ)
and *r*$\mathfrak{D}$-*add* = *embed2.rderiv-and-add lookup* (ε Σ)
and $\mathfrak{Q}$ = *embed2.samequot-exec lookup* (ε Σ) (*case-prod Singleton*)
by *unfold-locales* (*auto simp*: σ-*def* π-*def* ε-*def set-n-lists*)

lemma *enum-not-empty*[*simp*]: *Enum.enum* $\neq$ [] (**is** *?enum* $\neq$ [])
proof (*rule notI*)
assume *?enum* = []
hence *set ?enum* = {} **by** *simp*

thus *False* **unfolding** *UNIV-enum*[*symmetric*] **by** *simp*
qed

global-interpretation Φ: *formula Enum.enum* :: *'a* :: {*enum*, *linorder*} *list*
rewrites *embed2.samequot-exec lookup* (*ε* (*Enum.enum* :: *'a* :: {*enum*, *linorder*} *list*)) (*case-prod Singleton*) = 𝔔 *Enum.enum*
defines
pre-wf-formula = Φ.*pre-wf-formula*
and *wf-formula* = Φ.*wf-formula*
and *rexp-of* = Φ.*rexp-of*
and *rexp-of-alt* = Φ.*rexp-of-alt*
and *rexp-of-alt'* = Φ.*rexp-of-alt'*
and *rexp-of'* = Φ.*rexp-of'*
and *rexp-of''* = Φ.*rexp-of''*
and *valid-ENC* = Φ.*valid-ENC*
and *ENC* = Φ.*ENC*
and *dec-interp* = Φ.*stream-dec*
and *any* = Φ.*any*
by *unfold-locales* (*auto simp*: *σ-def π-def* 𝔔*-def*)

lemmas $lang_{WS1S}$*-rexp-of-norm* = *trans*[*OF sym*[*OF* Φ.$lang_{WS1S}$*-norm*] Φ.$lang_{WS1S}$*-rexp-of*]
lemmas $lang_{WS1S}$*-rexp-of'-norm* = *trans*[*OF sym*[*OF* Φ.$lang_{WS1S}$*-norm*] Φ.$lang_{WS1S}$*-rexp-of'*]
lemmas $lang_{WS1S}$*-rexp-of''-norm* = *trans*[*OF sym*[*OF* Φ.$lang_{WS1S}$*-norm*] Φ.$lang_{WS1S}$*-rexp-of''*]

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path slow*)›

global-interpretation *D*: *rexp-DFA σ Σ wf-atom Σ π lookup λx*. ≪*pnorm* (*inorm x*)≫
λa r. ≪𝔇 *Σ a r*≫ *final alphabet.wf* (*wf-atom Σ*) *n pnorm lang Σ n n*
for *Σ* :: *'a* :: *linorder list* **and** *n* :: *nat*
defines
test = *rexp-DA.test* (*final* :: *'a atom rexp* ⇒ *bool*)
and *step* = *rexp-DA.step* (*σ Σ*) (*λa r*. ≪𝔇 *Σ a r*≫) *pnorm n*
and *closure* = *rexp-DA.closure* (*σ Σ*) (*λa r*. ≪𝔇 *Σ a r*≫) *final pnorm n*
and *check-eqvRE* = *rexp-DA.check-eqv* (*σ Σ*) (*λx*. ≪*pnorm* (*inorm x*)≫) (*λa r*. ≪𝔇 *Σ a r*≫) *final pnorm n*
and *test-invariant* = *rexp-DA.test-invariant* (*final* :: *'a atom rexp* ⇒ *bool*) ::
((*'a* × *bool list*) *list* × -) *list* × - ⇒ *bool*
and *step-invariant* = *rexp-DA.step-invariant* (*σ Σ*) (*λa r*. ≪𝔇 *Σ a r*≫) *pnorm n*
and *closure-invariant* = *rexp-DA.closure-invariant* (*σ Σ*) (*λa r*. ≪𝔇 *Σ a r*≫) *final pnorm n*
and *counterexampleRE* = *rexp-DA.counterexample* (*σ Σ*) (*λx*. ≪*pnorm* (*inorm x*)≫) (*λa r*. ≪𝔇 *Σ a r*≫) *final pnorm n*
and *reachable* = *rexp-DA.reachable* (*σ Σ*) (*λx*. ≪*pnorm* (*inorm x*)≫) (*λa r*. ≪𝔇 *Σ a r*≫) *pnorm n*
and *automaton* = *rexp-DA.automaton* (*σ Σ*) (*λx*. ≪*pnorm* (*inorm x*)≫) (*λa r*. ≪𝔇 *Σ a r*≫) *pnorm n*
by *unfold-locales* (*auto simp only*: *comp-apply*
ACI-norm-wf ACI-norm-lang wf-inorm lang-inorm wf-pnorm lang-pnorm wf-lderiv

lang-lderiv
lang-final finite-fold-lderiv dest!: *lang-subset-lists*)

definition *check-eqv* **where**
check-eqv n φ $\psi \longleftrightarrow$ *wf-formula n* (*FOr* φ ψ) $\wedge$
slow.check-eqvRE Enum.enum n (*rexp-of''n* (*norm* φ)) (*rexp-of''n* (*norm* ψ))

definition *counterexample* **where**
counterexample n φ ψ =
map-option (λ*w*. *dec-interp n* (*FOV* (*FOr* φ ψ)) (*w* @− *sconst* (*any*, *replicate n False*)))
(*slow.counterexampleRE Enum.enum n* (*rexp-of''n* (*norm* φ)) (*rexp-of''n* (*norm* ψ)))

lemma *soundness*: *slow.check-eqv n* φ $\psi \Longrightarrow \Phi.lang_{WS1S}$ *n* $\varphi = \Phi.lang_{WS1S}$ *n* ψ
by (*rule box-equals*[*OF slow.D.check-eqv-sound*
sym[*OF trans*[*OF lang*$_{WS1S}$*-rexp-of''-norm*]] *sym*[*OF trans*[*OF lang*$_{WS1S}$*-rexp-of''-norm*]]])
(*auto simp*: *slow.check-eqv-def intro*!: Φ.*wf-rexp-of''*)

lemma *completeness*:
assumes $\Phi.lang_{WS1S}$ *n* $\varphi = \Phi.lang_{WS1S}$ *n* ψ *wf-formula n* (*FOr* φ ψ)
shows *slow.check-eqv n* φ ψ
using *assms*(*2*) **unfolding** *slow.check-eqv-def*
by (*intro conjI*[*OF assms*(*2*) *slow.D.check-eqv-complete*,
OF box-equals[*OF assms*(*1*) *lang*$_{WS1S}$*-rexp-of''-norm lang*$_{WS1S}$*-rexp-of''-norm*]])
(*auto intro*!: Φ.*wf-rexp-of''*)

setup ‹*Sign.map-naming Name-Space.parent-path*›

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path fast*)›

global-interpretation *D*: *rexp-DA-no-post* σ Σ *wf-atom* Σ π *lookup* λ*x*. *pnorm* (*inorm x*)
λ*a r*. *pnorm* ($\mathfrak{D}$ Σ *a r*) *final alphabet.wf* (*wf-atom* Σ) *n lang* Σ *n n*
for Σ :: *'a* :: *linorder list* **and** *n* :: *nat*
defines
test = *rexp-DA.test* (*final* :: *'a atom rexp* $\Rightarrow$ *bool*)
and *step* = *rexp-DA.step* (σ Σ) (λ*a r*. *pnorm* ($\mathfrak{D}$ Σ *a r*)) *id n*
and *closure* = *rexp-DA.closure* (σ Σ) (λ*a r*. *pnorm* ($\mathfrak{D}$ Σ *a r*)) *final id n*
and *check-eqvRE* = *rexp-DA.check-eqv* (σ Σ) (λ*x*. *pnorm* (*inorm x*)) (λ*a r*. *pnorm* ($\mathfrak{D}$ Σ *a r*)) *final id n*
and *test-invariant* = *rexp-DA.test-invariant* (*final* :: *'a atom rexp* $\Rightarrow$ *bool*) ::
(('*a* $\times$ *bool list*) *list* $\times$ -) *list* $\times$ - $\Rightarrow$ *bool*
and *step-invariant* = *rexp-DA.step-invariant* (σ Σ) (λ*a r*. *pnorm* ($\mathfrak{D}$ Σ *a r*)) *id n*
and *closure-invariant* = *rexp-DA.closure-invariant* (σ Σ) (λ*a r*. *pnorm* ($\mathfrak{D}$ Σ *a r*)) *final id n*
and *counterexampleRE* = *rexp-DA.counterexample* (σ Σ) (λ*x*. *pnorm* (*inorm x*))

$(\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ final\ id\ n$
and *reachable* = *rexp-DA.reachable* $(\sigma\ \Sigma)$ $(\lambda x.\ pnorm\ (inorm\ x))$ $(\lambda a\ r.\ pnorm$
$(\mathfrak{D}\ \Sigma\ a\ r))\ id\ n$
and *automaton* = *rexp-DA.automaton* $(\sigma\ \Sigma)$ $(\lambda x.\ pnorm\ (inorm\ x))$ $(\lambda a\ r.\ pnorm$
$(\mathfrak{D}\ \Sigma\ a\ r))\ id\ n$
by *unfold-locales* (*auto simp only*: *comp-apply*
ACI-norm-wf ACI-norm-lang wf-inorm lang-inorm wf-pnorm lang-pnorm wf-lderiv
lang-lderiv id-apply
lang-final dest!: *lang-subset-lists*)

definition *check-eqv* **where**
check-eqv n $\varphi\ \psi \longleftrightarrow$ *wf-formula n* (*FOr* $\varphi\ \psi$) $\wedge$
fast.check-eqvRE Enum.enum n (*rexp-of''n* (*norm* φ)) (*rexp-of''n* (*norm* ψ))

definition *counterexample* **where**
counterexample n $\varphi\ \psi$ =
map-option $(\lambda w.$ *dec-interp n* (*FOV* (*FOr* $\varphi\ \psi$)) (w @− *sconst* (*any*, *replicate*
n False)))
(*fast.counterexampleRE Enum.enum n* (*rexp-of''n* (*norm* φ)) (*rexp-of''n* (*norm*
ψ)))

lemma *soundness*: *fast.check-eqv n* $\varphi\ \psi \Longrightarrow \Phi.lang_{WS1S}\ n\ \varphi = \Phi.lang_{WS1S}\ n\ \psi$
by (*rule box-equals*[*OF fast.D.check-eqv-sound*
sym[*OF trans*[*OF* $lang_{WS1S}$*-rexp-of''-norm*]] *sym*[*OF trans*[*OF* $lang_{WS1S}$*-rexp-of''-norm*]]])
(*auto simp*: *fast.check-eqv-def intro*!: Φ.*wf-rexp-of''*)

setup ‹*Sign.map-naming Name-Space.parent-path*›

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path dual*)›

global-interpretation *D*: *rexp-DA-no-post* $\sigma\ \Sigma$ *wf-atom* $\Sigma\ \pi$ *lookup*
$\lambda x.$ *pnorm-dual* (*rexp-dual-of* (*inorm x*)) $\lambda a\ r.$ *pnorm-dual* (*Co*$\mathfrak{D}\ \Sigma\ a\ r$) *final-dual*
alphabet.wf-dual (*wf-atom* Σ) *n lang-dual* Σ *n n*
for Σ :: *'a* :: *linorder list* **and** *n* :: *nat*
defines
test = *rexp-DA.test* (*final-dual* :: *'a atom rexp-dual* $\Rightarrow$ *bool*)
and *step* = *rexp-DA.step* $(\sigma\ \Sigma)$ $(\lambda a\ r.$ *pnorm-dual* (*Co*$\mathfrak{D}\ \Sigma\ a\ r$)) *id n*
and *closure* = *rexp-DA.closure* $(\sigma\ \Sigma)$ $(\lambda a\ r.$ *pnorm-dual* (*Co*$\mathfrak{D}\ \Sigma\ a\ r$)) *final-dual*
id n
and *check-eqvRE* = *rexp-DA.check-eqv* $(\sigma\ \Sigma)$ $(\lambda x.$ *pnorm-dual* (*rexp-dual-of*
(*inorm x*))) $(\lambda a\ r.$ *pnorm-dual* (*Co*$\mathfrak{D}\ \Sigma\ a\ r$)) *final-dual id n*
and *test-invariant* = *rexp-DA.test-invariant* (*final-dual* :: *'a atom rexp-dual* $\Rightarrow$
bool) ::
(($'a \times$ *bool list*) *list* $\times$ -) *list* $\times$ - $\Rightarrow$ *bool*
and *step-invariant* = *rexp-DA.step-invariant* $(\sigma\ \Sigma)$ $(\lambda a\ r.$ *pnorm-dual* (*Co*$\mathfrak{D}\ \Sigma$
$a\ r$)) *id n*
and *closure-invariant* = *rexp-DA.closure-invariant* $(\sigma\ \Sigma)$ $(\lambda a\ r.$ *pnorm-dual*
(*Co*$\mathfrak{D}\ \Sigma\ a\ r$)) *final-dual id n*
and *counterexampleRE* = *rexp-DA.counterexample* $(\sigma\ \Sigma)$ $(\lambda x.$ *pnorm-dual* (*rexp-dual-of*

(*inorm x*))) (λ*a r*. *pnorm-dual* (*Co*𝔇 Σ *a r*)) *final-dual id n*
and *reachable* = *rexp-DA.reachable* (σ Σ) (λ*x*. *pnorm-dual* (*rexp-dual-of* (*inorm x*))) (λ*a r*. *pnorm-dual* (*Co*𝔇 Σ *a r*)) *id n*
and *automaton* = *rexp-DA.automaton* (σ Σ) (λ*x*. *pnorm-dual* (*rexp-dual-of* (*inorm x*))) (λ*a r*. *pnorm-dual* (*Co*𝔇 Σ *a r*)) *id n*
by *unfold-locales* (*auto simp only*: *comp-apply id-apply*
wf-inorm lang-inorm
wf-dual-pnorm-dual lang-dual-pnorm-dual
wf-dual-rexp-dual-of lang-dual-rexp-dual-of
wf-dual-lderiv-dual lang-dual-lderiv-dual
lang-dual-final-dual dest!: *lang-dual-subset-lists*)

definition *check-eqv* **where**
check-eqv n φ ψ ⟷ *wf-formula n* (*FOr φ ψ*) ∧
dual.check-eqvRE Enum.enum n (*rexp-of'' n* (*norm φ*)) (*rexp-of'' n* (*norm ψ*))

definition *counterexample* **where**
counterexample n φ ψ =
map-option (λ*w*. *dec-interp n* (*FOV* (*FOr φ ψ*)) (*w* @− *sconst* (*any*, *replicate n False*)))
(*dual.counterexampleRE Enum.enum n* (*rexp-of'' n* (*norm φ*)) (*rexp-of'' n* (*norm ψ*)))

lemma *soundness*: *dual.check-eqv n φ ψ* ⟹ Φ.*lang*$_{WS1S}$ *n φ* = Φ.*lang*$_{WS1S}$ *n ψ*
by (*rule box-equals*[*OF dual.D.check-eqv-sound*
sym[*OF trans*[*OF lang*$_{WS1S}$*-rexp-of''-norm*]] *sym*[*OF trans*[*OF lang*$_{WS1S}$*-rexp-of''-norm*]]])
(*auto simp*: *dual.check-eqv-def intro*!: Φ.*wf-rexp-of''*)

setup ‹*Sign.map-naming Name-Space.parent-path*›

References

[1] A. Krauss and T. Nipkow. Regular sets and expressions. *Archive of Formal Proofs*, 2010. http://isa-afp.org/entries/Regular-Sets.shtml, Formal proof development.

[2] D. Traytel and T. Nipkow. Verified decision procedures for MSO on words based on derivatives of regular expressions. In G. Morrisett and T. Uustalu, editors, *Proc. Int. Conf. Functional Programming, ICFP 2013*, pages 3–12. ACM, 2013.