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

Dmitriy Traytel and Tobias Nipkow

February 23, 2021

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)
⟨*proof*⟩

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)*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *concE*[*elim*]:
assumes *w ∈ A @@ B*
obtains *u v* **where** *u ∈ A v ∈ B w = u@v*
⟨*proof*⟩

lemma *conc-mono*: *A ⊆ C ⟹ B ⊆ D ⟹ A @@ B ⊆ C @@ D*
⟨*proof*⟩

lemma *conc-empty*[*simp*]: **shows** *{} @@ A = {}* **and** *A @@ {} = {}*
⟨*proof*⟩

lemma *conc-epsilon*[*simp*]: **shows** *{[]} @@ A = A* **and** *A @@ {[]} = A*
⟨*proof*⟩

lemma *conc-assoc*: $(A \mathbin{@@} B) \mathbin{@@} C = A \mathbin{@@} (B \mathbin{@@} C)$
⟨*proof*⟩

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

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

lemma *hom-image-conc*: $[\![\bigwedge xs\ ys.\ f\ (xs \mathbin{@} ys) = f\ xs \mathbin{@} f\ ys]\!] \Longrightarrow f \mathbin{`} (A \mathbin{@@} B) = f \mathbin{`} A \mathbin{@@} f \mathbin{`} B$
⟨*proof*⟩

lemma *map-image-conc*[*simp*]: $map\ f \mathbin{`} (A \mathbin{@@} B) = map\ f \mathbin{`} A \mathbin{@@} map\ f \mathbin{`} B$
⟨*proof*⟩

lemma *conc-subset-lists*: $A \subseteq lists\ S \Longrightarrow B \subseteq lists\ S \Longrightarrow A \mathbin{@@} B \subseteq lists\ S$
⟨*proof*⟩

1.2 Iteration of Languages

lemma *lang-pow-add*: $A \mathbin{^{\frown\frown}} (n + m) = A \mathbin{^{\frown\frown}} n \mathbin{@@} A \mathbin{^{\frown\frown}} m$
⟨*proof*⟩

lemma *lang-pow-simps*: $(A \mathbin{^{\frown\frown}} Suc\ n) = (A \mathbin{^{\frown\frown}} n \mathbin{@@} A)$
⟨*proof*⟩

lemma *lang-pow-empty*: $\{\} \mathbin{^{\frown\frown}} n = (if\ n = 0\ then\ \{[]\}\ else\ \{\})$
⟨*proof*⟩

lemma *lang-pow-empty-Suc*[*simp*]: $(\{\}::'a\ lang) \mathbin{^{\frown\frown}} Suc\ n = \{\}$
⟨*proof*⟩

lemma *conc-pow-comm*:
shows $A \mathbin{@@} (A \mathbin{^{\frown\frown}} n) = (A \mathbin{^{\frown\frown}} n) \mathbin{@@} A$
⟨*proof*⟩

lemma *length-lang-pow-ub*:
$ALL\ w : A.\ length\ w \leq k \Longrightarrow w : A \mathbin{^{\frown\frown}} n \Longrightarrow length\ w \leq k*n$
⟨*proof*⟩

lemma *length-lang-pow-lb*:
$ALL\ w : A.\ length\ w \geq k \Longrightarrow w : A \mathbin{^{\frown\frown}} n \Longrightarrow length\ w \geq k*n$
⟨*proof*⟩

lemma *lang-pow-subset-lists*: $A \subseteq lists\ S \Longrightarrow A\ ^{\frown\frown}\ n \subseteq lists\ S$
⟨*proof*⟩

lemma *star-subset-lists*: $A \subseteq lists\ S \Longrightarrow star\ A \subseteq lists\ S$
⟨*proof*⟩

lemma *star-if-lang-pow*[*simp*]: $w : A\ ^{\frown\frown}\ n \Longrightarrow w : star\ A$
⟨*proof*⟩

lemma *Nil-in-star*[*iff*]: $[] : star\ A$
⟨*proof*⟩

lemma *star-if-lang*[*simp*]: **assumes** $w : A$ **shows** $w : star\ A$
⟨*proof*⟩

lemma *append-in-starI*[*simp*]:
assumes $u : star\ A$ **and** $v : star\ A$ **shows** $u@v : star\ A$
⟨*proof*⟩

lemma *conc-star-star*: $star\ A\ @@\ star\ A = star\ A$
⟨*proof*⟩

lemma *conc-star-comm*:
shows $A\ @@\ star\ A = star\ A\ @@\ A$
⟨*proof*⟩

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*⟩

lemma *star-empty*[*simp*]: $star\ \{\} = \{[]\}$
⟨*proof*⟩

lemma *star-epsilon*[*simp*]: $star\ \{[]\} = \{[]\}$
⟨*proof*⟩

lemma *star-idemp*[*simp*]: $star\ (star\ A) = star\ A$
⟨*proof*⟩

lemma *star-unfold-left*: $star\ A = A\ @@\ star\ A \cup \{[]\}$ (**is** $?L = ?R$)
⟨*proof*⟩

lemma *concat-in-star*: $set\ ws \subseteq A \Longrightarrow concat\ ws : star\ A$
⟨*proof*⟩

lemma *in-star-iff-concat*:
  $w : star\ A = (EX\ ws.\ set\ ws \subseteq A\ \&\ w = concat\ ws\ \&\ [] \notin set\ ws)$
  (**is** - = (*EX ws. ?R w ws*))
⟨*proof*⟩

lemma *star-conv-concat*: $star\ A = \{concat\ ws | ws.\ set\ ws \subseteq A\ \&\ [] \notin set\ ws\}$
  ⟨*proof*⟩

lemma *star-insert-eps*[*simp*]: $star\ (insert\ []\ A) = star(A)$
⟨*proof*⟩

lemma *star-decom*:
  assumes *a*: $x \in star\ A\ x \neq []$
  shows $\exists\ a\ b.\ x = a\ @\ b \wedge a \neq [] \wedge a \in A \wedge b \in star\ A$
  ⟨*proof*⟩

lemma *Ball-starI*: $\forall\ a \in set\ as.\ [a] \in A \Longrightarrow as \in star\ A$
  ⟨*proof*⟩

lemma *map-image-star*[*simp*]: $map\ f\ `\ star\ A = star\ (map\ f\ `\ A)$
  ⟨*proof*⟩

1.3 Left-Quotients of Languages

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

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

abbreviation
  lQuotss :: *'a list ⇒ 'a lang set ⇒ 'a lang*
where
  $lQuotss\ s\ As \equiv \bigcup\ (lQuots\ s\ `\ As)$

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*]: $lQuot\ a\ \{[b] \mid b.\ P\ b\} = (if\ P\ a\ then\ \{[]\}\ else\ \{\})$
  and *lQuot-union*[*simp*]: $lQuot\ a\ (A \cup B) = lQuot\ a\ A \cup lQuot\ a\ B$
  and *lQuot-inter*[*simp*]: $lQuot\ a\ (A \cap B) = lQuot\ a\ A \cap lQuot\ a\ B$
  and *lQuot-compl*[*simp*]: $lQuot\ a\ (-A) = -\ lQuot\ a\ A$
  ⟨*proof*⟩

lemma *lQuot-conc-subset*: $lQuot\ a\ A\ @@\ B \subseteq lQuot\ a\ (A\ @@\ B)$ (**is** *?L ⊆ ?R*)
⟨*proof*⟩

lemma *lQuot-conc* [*simp*]: $lQuot\ c\ (A\ @@\ B) = (lQuot\ c\ A)\ @@\ B \cup (if\ []\ \in A$

then lQuot c B else {})
⟨*proof*⟩

lemma *lQuot-star* [*simp*]: *lQuot c* (*star A*) = (*lQuot c A*) @@ *star A*
⟨*proof*⟩

lemma *lQuot-diff*[*simp*]: *lQuot c* (*A* − *B*) = *lQuot c A* − *lQuot c B*
⟨*proof*⟩

lemma *lQuot-lists*[*simp*]: *c* : *S* ⟹ *lQuot c* (*lists S*) = *lists S*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lQuots-append*[*iff*]: *v* ∈ *lQuots w A* ⟷ *w* @ *v* ∈ *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rQuots-rev-lQuots*: *rQuots x A* = *rev* ‘ *lQuots x* (*rev* ‘ *A*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lQuot-rQuot*: *lQuot a* (*rQuot b A*) = *rQuot b* (*lQuot a A*)

⟨*proof*⟩

lemma *rQuot-lQuot*: *rQuot a* (*lQuot b A*) = *lQuot b* (*rQuot a A*)
⟨*proof*⟩

lemma *rev-simp-invert*: (*xs* @ [*x*] = *rev zs*) = (*zs* = *x* # *rev xs*)
⟨*proof*⟩

lemma *rev-append-invert*: (*xs* @ *ys* = *rev zs*) = (*zs* = *rev ys* @ *rev xs*)
⟨*proof*⟩

lemma *image-rev-lists*[*simp*]: *rev* ‘ *lists S* = *lists S*
⟨*proof*⟩

lemma *image-rev-conc*[*simp*]: *rev* ‘ (*A* @@ *B*) = *rev* ‘ *B* @@ *rev* ‘ *A*
⟨*proof*⟩

lemma *image-rev-star*[*simp*]: *rev* ‘ *star A* = *star* (*rev* ‘ *A*)
⟨*proof*⟩

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

lemma *rQuot-star* [*simp*]: *rQuot c* (*star A*) = *star A* @@ (*rQuot c A*)
⟨*proof*⟩

lemma *rQuot-diff*[*simp*]: *rQuot c* (*A* − *B*) = *rQuot c A* − *rQuot c B*
⟨*proof*⟩

lemma *rQuot-lists*[*simp*]: *c* : *S* ⟹ *rQuot c* (*lists S*) = *lists S*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *rQuots-append*[*iff*]: *v* ∈ *rQuots w A* ⟷ *v* @ *rev w* ∈ *A*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *biQuot-lQuot-rQuot*: *biQuot x y A* = *lQuot x* (*rQuot y A*)
⟨*proof*⟩

lemma *biQuots-rQuots-lQuots*: *biQuots x y A* = *rQuots y* (*lQuots x A*)
⟨*proof*⟩

lemma *biQuots-lQuots-rQuots*: *biQuots x y A* = *lQuots x* (*rQuots y A*)
⟨*proof*⟩

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* ∩ *B*) = *biQuot a b A* ∩ *biQuot a b B*
and *biQuot-compl*[*simp*]: *biQuot a b* (−*A*) = − *biQuot a b A*
⟨*proof*⟩

lemma *biQuot-conc* [*simp*]: *biQuot a b* (*A* @@ *B*) =
lQuot a A @@ *rQuot b B* ∪
(*if* [] ∈ *A* ∧ [] ∈ *B then biQuot a b A* ∪ *biQuot a b B*
else if [] ∈ *A then biQuot a b B*
else if [] ∈ *B then biQuot a b A*
else {})
⟨*proof*⟩

lemma *biQuot-star* [*simp*]: *biQuot a b* (*star A*) = *biQuot a b A* ∪ *lQuot a A* @@ *star A* @@ *rQuot b A*
⟨*proof*⟩

lemma *biQuot-diff*[*simp*]: *biQuot a b* (*A* − *B*) = *biQuot a b A* − *biQuot a b B*
⟨*proof*⟩

lemma *biQuot-lists*[*simp*]: *a* : *S* ⟹ *b* : *S* ⟹ *biQuot a b* (*lists S*) = *lists S*
⟨*proof*⟩

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*⟧ ⟹
biQuots (*s1* @ *s2*) (*t1* @ *t2*) *A* = *biQuots s2 t2* (*biQuots s1 t1 A*)

⟨*proof*⟩

lemma *biQuots-append*[*iff*]: $v \in biQuots\ u\ w\ A \longleftrightarrow u\ @\ v\ @\ rev\ w \in A$
⟨*proof*⟩

1.6 Arden's Lemma

lemma *arden-helper*:
assumes *eq*: $X = A\ @@\ X \cup B$
shows $X = (A\ ^{\frown\frown}\ Suc\ n)\ @@\ X \cup (\bigcup m{\leq}n.\ (A\ ^{\frown\frown}\ m)\ @@\ B)$
⟨*proof*⟩

lemma *Arden*:
assumes $[] \notin A$
shows $X = A\ @@\ X \cup B \longleftrightarrow X = star\ A\ @@\ B$
⟨*proof*⟩

lemma *reversed-arden-helper*:
assumes *eq*: $X = X\ @@\ A \cup B$
shows $X = X\ @@\ (A\ ^{\frown\frown}\ Suc\ n) \cup (\bigcup m{\leq}n.\ B\ @@\ (A\ ^{\frown\frown}\ m))$
⟨*proof*⟩

theorem *reversed-Arden*:
assumes *nemp*: $[] \notin A$
shows $X = X\ @@\ A \cup B \longleftrightarrow X = B\ @@\ star\ A$
⟨*proof*⟩

1.7 Lists of Fixed Length

abbreviation *listsN* **where** $listsN\ n\ S \equiv \{xs.\ xs \in lists\ S \wedge length\ xs = n\}$

lemma *tl-listsN*: $A \subseteq listsN\ (n + 1)\ S \Longrightarrow tl\ `\ A \subseteq listsN\ n\ S$
⟨*proof*⟩

lemma *map-tl-listsN*: $A \subseteq lists\ (listsN\ (n + 1)\ S) \Longrightarrow map\ tl\ `\ A \subseteq lists\ (listsN\ n\ S)$
⟨*proof*⟩

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*
⟨*proof*⟩

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 ≠ {}
⟨*proof*⟩

lemma *toplevel-summands-finite*[*simp*]:
finite (*toplevel-summands r*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *toplevel-summands-not-Plus*[*simp*]:
(∀ *r s*. *x* ≠ *Plus r s*) ⟹ *toplevel-summands x* = {*x*}
⟨*proof*⟩

lemma *toplevel-summands-PLUS-strong*:
⟦*xs* ≠ []; *list-all* (λ*x*. ¬(∃ *r s*. *x* = *Plus r s*)) *xs*⟧ ⟹ *toplevel-summands* (*PLUS xs*) = *set xs*
⟨*proof*⟩

lemma *toplevel-summands-flatten*:
⟦*X* ≠ {}; *finite X*; ∀ *x* ∈ *X*. ¬(∃ *r s*. *x* = *Plus r s*)⟧ ⟹ *toplevel-summands* (*flatten PLUS X*) = *X*
⟨*proof*⟩

lemma *ACI-norm-Plus*:
«*r*» = *Plus s t* ⟹ ∃ *s t*. *r* = *Plus s t*
⟨*proof*⟩

lemma *toplevel-summands-flatten-ACI-norm-image*:
toplevel-summands (*flatten PLUS* (*ACI-norm* ‘ *toplevel-summands r*)) = *ACI-norm* ‘ *toplevel-summands r*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *toplevel-summands-ACI-norm*:
toplevel-summands «*r*» = *ACI-norm* ‘ *toplevel-summands r*
⟨*proof*⟩

lemma *ACI-norm-flatten*:
«*r*» = *flatten PLUS* (*ACI-norm* ‘ *toplevel-summands r*)
⟨*proof*⟩

theorem *ACI-norm-idem*[*simp*]:
««*r*»» = «*r*»
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *toplevel-summands-ACI-norm-alt*:
toplevel-summands (*ACI-norm-alt r*) = *ACI-norm-alt* ‘ *toplevel-summands r*
⟨*proof*⟩

lemma *ACI-norm-alt-Plus*:
ACI-norm-alt r = *Plus s t* ⟹ ∃ *s t*. *r* = *Plus s t*
⟨*proof*⟩

lemma *toplevel-summands-flatten-ACI-norm-alt-image*:
toplevel-summands (*flatten PLUS* (*ACI-norm-alt* ‘ *toplevel-summands r*)) =
ACI-norm-alt ‘ *toplevel-summands r*
⟨*proof*⟩

lemma *ACI-norm-ACI-norm-alt*: «*ACI-norm-alt r*» = «*r*»
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *ACI-nPlus-ACI-norm*[*simp*]: *ACI-nPlus* «*r*» «*s*» = «*Plus r s*»
⟨*proof*⟩

lemma *ACI-norm-alt*:
ACI-norm-alt r = «*r*»
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *final-PLUS*:
final (*PLUS xs*) = (∃ *r* ∈ *set xs*. *final r*)
⟨*proof*⟩

theorem *ACI-norm-final*[*simp*]:
final «*r*» = *final r*
⟨*proof*⟩

2.4 Wellformedness w.r.t. an alphabet

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

primrec *wf* :: *nat* $\Rightarrow$ *'b rexp* $\Rightarrow$ *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* $\wedge$ *wf n s*) |
wf n (*Times r s*) = (*wf n r* $\wedge$ *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* $\wedge$ *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* $\in$ Σ *n*) $\wedge$ *wf-word n ws*)

lemma *wf-word-snoc*[*simp*]: *wf-word n* (*ws* @ [*w*]) = ((*w* $\in$ Σ *n*) $\wedge$ *wf-word n ws*)
⟨*proof*⟩

lemma *wf-word-append*[*simp*]: *wf-word n* (*ws* @ *vs*) = (*wf-word n ws* $\wedge$ *wf-word n vs*)
⟨*proof*⟩

lemma *wf-word*: *wf-word n w* = (*w* $\in$ *lists* (Σ *n*))
⟨*proof*⟩

lemma *toplevel-summands-wf*:
wf n s = ($\forall$ *r*$\in$*toplevel-summands s*. *wf n r*)
⟨*proof*⟩

lemma *wf-PLUS*[*simp*]:
wf n (*PLUS xs*) = ($\forall$ *r* $\in$ *set xs*. *wf n r*)
⟨*proof*⟩

lemma *wf-TIMES*[*simp*]:
wf n (*TIMES xs*) = ($\forall$ *r* $\in$ *set xs*. *wf n r*)
⟨*proof*⟩

lemma *wf-flatten-PLUS*[*simp*]:
finite X $\Longrightarrow$ *wf n* (*flatten PLUS X*) = ($\forall$ *r* $\in$ *X*. *wf n r*)
⟨*proof*⟩

theorem *ACI-norm-wf*[*simp*]:
wf n «r» = wf n r
⟨*proof*⟩

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

lemma *wf-flatten-INTERSECT*[*simp*]:
finite X ⟹ wf n (flatten INTERSECT X) = (∀ r ∈ X. wf n r)
⟨*proof*⟩

end

2.5 Language

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

primrec *lang* :: *nat* ⇒ *'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* ∧ *x* ∈ Σ *n*} |
lang n (*Plus r s*) = (*lang n r*) ∪ (*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* ∩ *lang n s*) |
lang n (*Pr r*) = *map project* ‘ *lang* (*n* + *1*) *r*

lemma *wf-word-map-project*[*simp*]: *wf-word* (*Suc n*) *ws* ⟹ *wf-word n* (*map project ws*)
⟨*proof*⟩

lemma *wf-lang-wf-word*: *wf n r* ⟹ ∀ *w* ∈ *lang n r. wf-word n w*
⟨*proof*⟩

lemma *lang-subset-lists*: *wf n r* ⟹ *lang n r* ⊆ *lists* (Σ *n*)
⟨*proof*⟩

lemma *toplevel-summands-lang*:
r ∈ *toplevel-summands s* ⟹ *lang n r* ⊆ *lang n s*
⟨*proof*⟩

lemma *toplevel-summands-lang-UN*:
lang n s = (⋃ *r*∈*toplevel-summands s*. *lang n r*)
⟨*proof*⟩

lemma *toplevel-summands-in-lang*:
w ∈ *lang n s* = (∃ *r*∈*toplevel-summands s*. *w* ∈ *lang n r*)
⟨*proof*⟩

lemma *lang-PLUS*[*simp*]:
lang n (*PLUS xs*) = (⋃ *r* ∈ *set xs*. *lang n r*)
⟨*proof*⟩

lemma *lang-TIMES*[*simp*]:
lang n (*TIMES xs*) = *foldr* (@@) (*map* (*lang n*) *xs*) {[]}
⟨*proof*⟩

lemma *lang-flatten-PLUS*:
finite X ⟹ *lang n* (*flatten PLUS X*) = (⋃ *r* ∈ *X*. *lang n r*)
⟨*proof*⟩

theorem *ACI-norm-lang*[*simp*]:
lang n «*r*» = *lang n r*
⟨*proof*⟩

lemma *lang-final*: *final r* = ([] ∈ *lang n r*)
⟨*proof*⟩

lemma *in-lang-INTERSECT*:
wf-word n w ⟹ *w* ∈ *lang n* (*INTERSECT xs*) = (∀ *r* ∈ *set xs*. *w* ∈ *lang n r*)
⟨*proof*⟩

lemma *lang-INTERSECT*:
lang n (*INTERSECT xs*) = (*if xs* = [] *then lists* (Σ *n*) *else* ⋂ *r* ∈ *set xs*. *lang n r*)
⟨*proof*⟩

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*⟩

end

3 Derivatives of Π-Extended Regular Expressions

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

3.1 Syntactic Derivatives

primrec *lderiv* :: *'a* $\Rightarrow$ *'b rexp* $\Rightarrow$ *'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 (λa'. 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))
  ⟨*proof*⟩

lemma *lderivs-Zero*[*simp*]: *lderivs xs Zero = Zero*
  ⟨*proof*⟩

lemma *lderivs-Full*[*simp*]: *lderivs xs Full = Full*
  ⟨*proof*⟩

lemma *lderivs-One*: *lderivs xs One* $\in$ {*Zero*, *One*}
  ⟨*proof*⟩

lemma *lderivs-Atom*: *lderivs xs (Atom as)* $\in$ {*Zero*, *One*, *Atom as*}
⟨*proof*⟩

lemma *lderivs-Plus*: *lderivs xs (Plus r s) = Plus (lderivs xs r) (lderivs xs s)*

⟨*proof*⟩

lemma *lderivs-PLUS*: *lderivs xs* (*PLUS ys*) = *PLUS* (*map* (*lderivs xs*) *ys*)
⟨*proof*⟩

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*⟩

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*⟩

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*}
⟨*proof*⟩

lemma *ex-lderivs-Pr*: ∃ *s*. *lderivs ass* (*Pr r*) = *Pr s*
⟨*proof*⟩

lemma *toplevel-summands-PLUS*:
xs ≠ [] ⟹ *toplevel-summands* (*PLUS* (*map f xs*)) = (⋃ *r* ∈ *set xs*. *toplevel-summands* (*f r*))
⟨*proof*⟩

lemma *lderiv-toplevel-summands-Zero*:
⟦*lderivs xs* (*Pr r*) = *Pr s*; *toplevel-summands r* = {*Zero*}⟧ ⟹ *toplevel-summands s* = {*Zero*}
⟨*proof*⟩

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*⟩

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*⟩

lemma *ACI-norm-toplevel-summands-Zero*: *toplevel-summands r* ⊆ {*Zero*} ⟹ «*r*» = *Zero*

⟨*proof*⟩

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*⟩

lemma *finite-ACI-norm-toplevel-summands*: *finite B ⟹ finite {f «s» |s. toplevel-summands s ⊆ B}*
⟨*proof*⟩

lemma *lderivs-Not*: *lderivs xs (Not r) = Not (lderivs xs r)*
⟨*proof*⟩

lemma *lderivs-Inter*: *lderivs xs (Inter r s) = Inter (lderivs xs r) (lderivs xs s)*
⟨*proof*⟩

theorem *finite-lderivs*: *finite {«lderivs xs r» | xs . True}*
⟨*proof*⟩

3.3 Wellformedness and language of derivatives

lemma *wf-lderiv*[*simp*]: *wf n r ⟹ wf n (lderiv w r)*
⟨*proof*⟩

lemma *wf-lderivs*[*simp*]: *wf n r ⟹ wf n (lderivs ws r)*
⟨*proof*⟩

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*⟩

lemma *lang-lderiv*: *⟦wf n r; w ∈ Σ n⟧ ⟹ lang n (lderiv w r) = lQuot w (lang n r)*
⟨*proof*⟩

lemma *lang-lderivs*: *⟦wf n r; wf-word n ws⟧ ⟹ lang n (lderivs ws r) = lQuots ws (lang n r)*
⟨*proof*⟩

corollary *lderivs-final*:
assumes *wf n r wf-word n ws*
shows *final (lderivs ws r) ⟷ ws ∈ lang n r*
⟨*proof*⟩

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 ($\lambda r\ s.\ «r» = «s»$) *xs ys* $\Longrightarrow$ $«PLUS\ xs» = «PLUS\ ys»$
⟨*proof*⟩

lemma *toplevel-summands-ACI-norm-lderiv*:
$(\bigcup a \in toplevel\text{-}summands\ r.\ toplevel\text{-}summands\ «lderiv\ as\ «a»») = toplevel\text{-}summands\ «lderiv\ as\ «r»»$
⟨*proof*⟩

theorem *ACI-norm-lderiv*:
$«lderiv\ as\ «r»» = «lderiv\ as\ r»$
⟨*proof*⟩

corollary *lderiv-preserves*: $«r» = «s» \Longrightarrow «lderiv\ as\ r» = «lderiv\ as\ s»$
⟨*proof*⟩

lemma *lderivs-snoc*[*simp*]: $lderivs\ (ws\ @\ [w])\ r = (lderiv\ w\ (lderivs\ ws\ r))$
⟨*proof*⟩

theorem *ACI-norm-lderivs*:
$«lderivs\ ws\ «r»» = «lderivs\ ws\ r»$
⟨*proof*⟩

lemma *lderivs-alt*: $«lderivs\ w\ r» = fold\ (\lambda a\ r.\ «lderiv\ a\ r»)\ w\ «r»$
⟨*proof*⟩

lemma *finite-fold-lderiv*: $finite\ \{fold\ (\lambda a\ r.\ «lderiv\ a\ r»)\ w\ «s»\ |w.\ True\}$
⟨*proof*⟩

end

4 Some Useful Regular Operators

primrec $REV :: 'a\ rexp \Rightarrow 'a\ rexp$ **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*
⟨*proof*⟩

lemma *final-REV*[*simp*]: *final* (*REV r*) = *final r*
⟨*proof*⟩

lemma *REV-PLUS*: *REV* (*PLUS xs*) = *PLUS* (*map REV xs*)
⟨*proof*⟩

lemma (**in** *alphabet*) *wf-REV*[*simp*]: *wf n r* ⟹ *wf n* (*REV r*)
⟨*proof*⟩

lemma (**in** *project*) *lang-REV*[*simp*]: *lang n* (*REV r*) = *rev* ‘ *lang n r*
⟨*proof*⟩

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* (λ*a′*. *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*)
⟨*proof*⟩

lemma *rderivs-append*: *rderivs* (*ws* @ *ws′*) *r* = *rderivs ws′* (*rderivs ws r*)
⟨*proof*⟩

lemma *rderiv-lderiv*: *rderiv as r* = *REV* (*lderiv as* (*REV r*))
⟨*proof*⟩

lemma *rderivs-lderivs*: *rderivs w r* = *REV* (*lderivs w* (*REV r*))
⟨*proof*⟩

lemma *wf-rderiv*[*simp*]: *wf n r* $\Longrightarrow$ *wf n* (*rderiv w r*)
⟨*proof*⟩

lemma *wf-rderivs*[*simp*]: *wf n r* $\Longrightarrow$ *wf n* (*rderivs ws r*)
⟨*proof*⟩

lemma *lang-rderiv*: ⟦*wf n r*; *as* $\in \Sigma$ *n*⟧ $\Longrightarrow$ *lang n* (*rderiv as r*) = *rQuot as* (*lang n r*)
⟨*proof*⟩

lemma *lang-rderivs*: ⟦*wf n r*; *wf-word n w*⟧ $\Longrightarrow$ *lang n* (*rderivs w r*) = *rQuots w* (*lang n r*)
⟨*proof*⟩

corollary *rderivs-final*:
assumes *wf n r wf-word n w*
shows *final* (*rderivs w r*) $\longleftrightarrow$ *rev w* $\in$ *lang n r*
⟨*proof*⟩

lemma *toplevel-summands-REV*[*simp*]: *toplevel-summands* (*REV r*) = *REV* ‘ *toplevel-summands r*
⟨*proof*⟩

lemma *ACI-norm-REV*: «*REV* «*r*»» = «*REV r*»
⟨*proof*⟩

lemma *ACI-norm-rderiv*: «*rderiv as* «*r*»» = «*rderiv as r*»
⟨*proof*⟩

lemma *ACI-norm-rderivs*: «*rderivs w* «*r*»» = «*rderivs w r*»
⟨*proof*⟩

theorem *finite-rderivs*: *finite* {«*rderivs xs r*» | *xs* . *True*}
⟨*proof*⟩

lemma *lderiv-PLUS*[*simp*]: *lderiv a* (*PLUS xs*) = *PLUS* (*map* (*lderiv a*) *xs*)
⟨*proof*⟩

lemma *rderiv-PLUS*[*simp*]: *rderiv a* (*PLUS xs*) = *PLUS* (*map* (*rderiv a*) *xs*)
⟨*proof*⟩

lemma *lang-rderiv-lderiv*: *lang n* (*rderiv a* (*lderiv b r*)) = *lang n* (*lderiv b* (*rderiv a r*))
⟨*proof*⟩

lemma *lang-lderiv-rderiv*: *lang n* (*lderiv a* (*rderiv b r*)) = *lang n* (*rderiv b* (*lderiv a r*))
⟨*proof*⟩

lemma *lang-rderiv-lderivs*[*simp*]: ⟦*wf n r*; *wf-word n w*; *a* ∈ Σ *n*⟧ ⟹
lang n (*rderiv a* (*lderivs w r*)) = *lang n* (*lderivs w* (*rderiv a r*))
⟨*proof*⟩

lemma *lang-lderiv-rderivs*[*simp*]: ⟦*wf n r*; *wf-word n w*; *a* ∈ Σ *n*⟧ ⟹
lang n (*lderiv a* (*rderivs w r*)) = *lang n* (*rderivs w* (*lderiv a r*))
⟨*proof*⟩

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

lemma *lang-biderivs*: ⟦*wf n r*; *wf-word n w1*; *wf-word n w2*⟧ ⟹
lang n (*biderivs w1 w2 r*) = *biQuots w1 w2* (*lang n r*)
⟨*proof*⟩

lemma *wf-biderivs*[*simp*]: *wf n r* ⟹ *wf n* (*biderivs w1 w2 r*)
⟨*proof*⟩

corollary *biderivs-final*:
assumes *wf n r wf-word n w1 wf-word n w2*
shows *final* (*biderivs w1 w2 r*) ⟷ *w1* @ *rev w2* ∈ *lang n r*
⟨*proof*⟩

lemma *ACI-norm-biderivs*: «*biderivs w1 w2* «*r*»» = «*biderivs w1 w2 r*»
⟨*proof*⟩

lemma *finite* {«*biderivs w1 w2 r*» | *w1 w2* . *True*}
⟨*proof*⟩

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* [] = []
⟨*proof*⟩

lemma *Least-fin-cut-same*: (*LEAST n*. *drop n xs* = *replicate* (*length xs* − *n*) *y*) =
length xs − *length* (*takeWhile* (λ*x*. *x* = *y*) (*rev xs*))
(**is** *Least ?P* = *?min*)
⟨*proof*⟩

lemma *takeWhile-takes-all*: *length xs* = *m* ⟹ *m* ≤ *length* (*takeWhile P xs*) ⟷
Ball (*set xs*) *P*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fin-cut-same-singleton*[*simp*]: *fin-cut-same x* (*xs* @ [*x*]) = *fin-cut-same x xs*
⟨*proof*⟩

lemma *fin-cut-same-replicate*[*simp*]: *fin-cut-same x* (*xs* @ *replicate n x*) = *fin-cut-same x xs*
⟨*proof*⟩

lemma *fin-cut-sameE*: *fin-cut-same x xs* = *ys* ⟹ ∃ *m*. *xs* = *ys* @ *replicate m x*
⟨*proof*⟩

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*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *wf-word-replicate*[*simp*]: *a* ∈ Σ *n* ⟹ *wf-word n* (*replicate m a*)
⟨*proof*⟩

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

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*)
⟨*proof*⟩

lemma *lang-samequot*: ⟦*wf n r*; *a* ∈ Σ *n*⟧ ⟹
lang n (*samequot a r*) = *SAMEQUOT a* (*lang n r*)
⟨*proof*⟩

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 ≠ [] ∧ *distinct* (*snd brs*) ∧
(∀ *i* < *length* (*snd brs*). *snd brs* ! *i* = «*rderivs* (*replicate* (*length* (*snd brs*) − *1* − *i*) *as*) *r*»)

lemma *invar-rderiv-and-add-init*: *invar-rderiv-and-add as r* (*True*, [«*r*»])
⟨*proof*⟩

lemma *invar-rderiv-and-add-step*: *invar-rderiv-and-add as r brs* ⟹ *fst brs* ⟹
invar-rderiv-and-add as r (*rderiv-and-add as brs*)
⟨*proof*⟩

lemma *rderivs-replicate-mult*: ⟦«*rderivs* (*replicate i as*) *r*» = «*r*»; *i* > *0*⟧ ⟹
«*rderivs* (*replicate* (*m* ∗ *i*) *as*) *r*» = «*r*»
⟨*proof*⟩

lemma *rderivs-replicate-mult-rest*:
assumes «*rderivs* (*replicate i as*) *r*» = «*r*» *k* < *i*
shows «*rderivs* (*replicate* (*m* ∗ *i* + *k*) *as*) *r*» = «*rderivs* (*replicate k as*) *r*» (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *rderivs-replicate-mod*:
assumes «*rderivs* (*replicate i as*) *r*» = «*r*» *i* > *0*
shows «*rderivs* (*replicate m as*) *r*» = «*rderivs* (*replicate* (*m mod i*) *as*) *r*» (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *rderivs-replicate-diff*: ⟦«*rderivs* (*replicate i as*) *r*» = «*rderivs* (*replicate j as*) *r*»; *i* > *j*⟧ ⟹
«*rderivs* (*replicate* (*i* − *j*) *as*) (*rderivs* (*replicate j as*) *r*)» = «*rderivs* (*replicate j as*) *r*»
⟨*proof*⟩

lemma *samequot-wf*:
assumes *wf n r while-option fst* (*rderiv-and-add as*) (*True*, [«*r*»]) = *Some* (*b*, *rs*)
shows *wf n* (*PLUS rs*)
⟨*proof*⟩

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*⟩

lemma *length-subset-card*: ⟦*finite X*; *distinct* (*x* # *xs*); *set* (*x* # *xs*) ⊆ *X*⟧ ⟹ *length xs* < *card X*
⟨*proof*⟩

lemma *samequot-termination*:
assumes *while-option fst* (*rderiv-and-add as*) (*True*, [«*r*»]) = *None* (**is** *?cl* = *None*)
shows *False*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *samequot-exec-samequot*: *lang n* (*samequot-exec as r*) = *lang n* (*samequot as r*)
⟨*proof*⟩

lemma *lang-samequot-exec*:
⟦*wf n r*; *as* ∈ Σ *n*⟧ ⟹ *lang n* (*samequot-exec as r*) = *SAMEQUOT as* (*lang n r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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**
$SUFFIX\ r = flatten\ PLUS\ \{«lderivs\ w\ r»|\ w.\ wf\text{-}word\ n\ w\}$

lemma *finite-lderivs-wf*: $finite\ \{«lderivs\ w\ r»|\ w.\ wf\text{-}word\ n\ w\}$
⟨*proof*⟩

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

lemma *wf-SUFFIX*[*simp*]: $wf\ n\ r \Longrightarrow wf\ n\ (SUFFIX\ r)$
⟨*proof*⟩

lemma *lang-SUFFIX*[*simp*]: $wf\ n\ r \Longrightarrow lang\ n\ (SUFFIX\ r) = Suffix\ (lang\ n\ r)$
⟨*proof*⟩

lemma *wf-PREFIX*[*simp*]: $wf\ n\ r \Longrightarrow wf\ n\ (PREFIX\ r)$
⟨*proof*⟩

lemma *lang-PREFIX*[*simp*]: $wf\ n\ r \Longrightarrow lang\ n\ (PREFIX\ r) = Prefix\ (lang\ n\ r)$
⟨*proof*⟩

end

lemma *take-drop-CycleI*[*intro*!]: $x \in L \Longrightarrow drop\ i\ x\ @\ take\ i\ x \in Cycle\ L$
⟨*proof*⟩

lemma *take-drop-CycleI'*[*intro*!]: $drop\ i\ x\ @\ take\ i\ x \in L \Longrightarrow x \in Cycle\ L$
⟨*proof*⟩

end

5 Π-Extended Dual Regular Expressions

5.1 Syntax of regular expressions

datatype *'a rexp-dual* =
CoZero (*co*: *bool*) |
CoOne (*co*: *bool*) |
CoAtom (*co*: *bool*) *'a* |
CoPlus (*co*: *bool*) *'a rexp-dual 'a rexp-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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lang-dual-subset-lists*: *wf-dual n r ⟹ lang-dual n r ⊆ lists (Σ n)*
⟨*proof*⟩

lemma *lang-dual-final-dual*: *final-dual r = ([] ∈ lang-dual n r)*
⟨*proof*⟩

lemma *lang-dual-PLUS-dual*[*simp*]:
lang-dual n (CoPLUS-dual True xs) = (⋃ r ∈ set xs. lang-dual n r)
⟨*proof*⟩

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)
⟨*proof*⟩

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 ⟹ wf-dual n (lderiv-dual w r)*
⟨*proof*⟩

lemma *wf-dual-lderivs-dual*[*simp*]: *wf-dual n r ⟹ wf-dual n (lderivs-dual ws r)*
⟨*proof*⟩

lemma *lang-dual-lderiv-dual*: ⟦*wf-dual n r*; *w* ∈ Σ *n*⟧ ⟹
lang-dual n (*lderiv-dual w r*) = *lQuot w* (*lang-dual n r*)
⟨*proof*⟩

lemma *lang-dual-lderivs-dual*: ⟦*wf-dual n r*; *wf-word n ws*⟧ ⟹
lang-dual n (*lderivs-dual ws r*) = *lQuots ws* (*lang-dual n r*)
⟨*proof*⟩

corollary *lderivs-dual-final-dual*:
assumes *wf-dual n r wf-word n ws*
shows *final-dual* (*lderivs-dual ws r*) ⟷ *ws* ∈ *lang-dual n r*
⟨*proof*⟩

end

fun *pnCoPlus* :: *bool* ⇒ *'a*::*linorder rexp-dual* ⇒ *'a rexp-dual* ⇒ *'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 ≤ *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 ≤ *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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-dual-pnCoPr*[*simp*]: *wf-dual* (*Suc n*) *r* ⟹ *lang-dual n* (*pnCoPr b r*) = *lang-dual n* (*CoPr b r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-dual-pnorm-dual*[*simp*]: *wf-dual n r* ⟹ *lang-dual n* (*pnorm-dual r*) = *lang-dual n r*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-dual-CoNot*[*simp*]: *wf-dual n r* ⟹ *lang-dual n* (*CoNot r*) = *lists* (Σ *n*) − *lang-dual n r*
⟨*proof*⟩

lemma (**in** *alphabet*) *wf-dual-rexp-dual-of*[*simp*]: *wf n r* ⟹ *wf-dual n* (*rexp-dual-of r*)
⟨*proof*⟩

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

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*)
⟨*proof*⟩

lemma *image2p-apply*: *BNF-Greatest-Fixpoint.image2p f g R x y* = (∃ *x′ y′*. *R x′ y′* ∧ *f x′* = *x* ∧ *g y′* = *y*)
⟨*proof*⟩

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*⟩

lemma *in-fold-lQuot*: $v \in fold\ lQuot\ w\ L \longleftrightarrow w\ @\ v \in L$
⟨*proof*⟩

lemma (**in** *project*) *lang-eq-ext*: $[\![wf\ n\ r;\ wf\ n\ s]\!] \Longrightarrow (lang\ n\ r = lang\ n\ s) =$
$(\forall\, w \in lists(\Sigma\ n).\ w \in lang\ n\ r \longleftrightarrow w \in lang\ n\ s)$
⟨*proof*⟩

lemma (**in** *project*) *lang-eq-ext-Nil-fold-Deriv*:
fixes *r s n*
assumes *WF*: *wf n r wf n s*
defines $\mathfrak{B} \equiv \{(fold\ lQuot\ w\ (lang\ n\ r),\ fold\ lQuot\ w\ (lang\ n\ s))|\ w.\ w \in lists\ (\Sigma\ n)\}$
shows $lang\ n\ r = lang\ n\ s \longleftrightarrow (\forall\, (K,\ L) \in \mathfrak{B}.\ [] \in K \longleftrightarrow [] \in L)$
⟨*proof*⟩

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*]: $[\![a \in set\ (\sigma\ n);\ wf\text{-}state\ s]\!] \Longrightarrow L\ (delta\ a\ s) = lQuot\ a\ (L\ s)$
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*]: $[\![a \in set\ (\sigma\ n);\ wf\text{-}state\ s]\!] \Longrightarrow wf\text{-}state\ (delta\ a\ s)$
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*]: $[\![wf\text{-}word\ n\ w;\ wf\text{-}state\ s]\!] \Longrightarrow L\ (fold\ delta\ w\ s) = fold\ lQuot\ w\ (L\ s)$
⟨*proof*⟩

definition *progression* (**infix** → *60*) **where**
$R \to S = (\forall\, s1\ s2.\ R\ s1\ s2 \longrightarrow wf\text{-}state\ s1 \wedge wf\text{-}state\ s2 \wedge final\ s1 = final\ s2 \wedge$
$(\forall\, x \in set\ (\sigma\ n).\ BNF\text{-}Greatest\text{-}Fixpoint.image2p\ post\ post\ S\ (post\ (delta\ x\ s1))\ (post\ (delta\ x\ s2))))$

lemma *SUPR-progression*[*intro*!]: $\forall\, n.\ \exists\, m.\ X\ n \to Y\ m \Longrightarrow (SUP\ n.\ X\ n) \to$

(*SUP n. Y n*)
⟨*proof*⟩

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*⟩

lemma *bisimulation-eqL*: *bisimulation* (λ*s1 s2*. *wf-state s1* ∧ *wf-state s2* ∧ *L s1* = *L s2*)
⟨*proof*⟩

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*⟩

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*⟩

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)})
  ⟨proof⟩

lemma step-invariant-mono:
  assumes step-invariant (ws, ps, N) = (ws', ps', N')
  shows snd ‘ set ws ∪ set ps ⊆ snd ‘ set ws' ∪ set ps'
⟨proof⟩

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 (λa. (a#xs, delta a r, delta a s)) (σ n))) ∧
  N' = set (map (λa. (post (delta a r), post (delta a s))) (σ n)) ∪ N)
  ⟨proof⟩
```

lemma *invariant*: *invariant r s st* $\Longrightarrow$ *test-invariant st* $\Longrightarrow$ *invariant r s* (*step-invariant st*)
⟨*proof*⟩

lemma *step-commute*: *ws* $\neq$ [] $\Longrightarrow$
  (*case step-invariant* (*ws*, *ps*, *N*) *of* (*ws'*, *ps'*, *N'*) $\Rightarrow$ (*map snd ws'*, *N'*)) = *step* (*map snd ws*, *N*)
⟨*proof*⟩

lemma *closure-invariant-closure*:
  map-option (λ(*ws*, *ps*, *N*). (*map snd ws*, *N*)) (*closure-invariant* (*ws*, *ps*, *N*)) = *closure* (*map snd ws*, *N*)
  ⟨*proof*⟩

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* = [] $\Longrightarrow$ *lang n r* = *lang n s* **and**
    counterexample: *ws* $\neq$ [] $\Longrightarrow$ *rev* (*fst* (*hd ws*)) $\in$ *lang n r* $\longleftrightarrow$ *rev* (*fst* (*hd ws*)) $\notin$ *lang n s*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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 ([], -) $\Rightarrow$ *True* | - $\Rightarrow$ *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*
  ⟨*proof*⟩

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*,-,-) # -, -) $\Rightarrow$ *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*
⟨*proof*⟩

Auxiliary exacutable functions:

definition *reachable* :: *'b rexp* ⇒ *'s set* **where**
reachable s = *snd* (*the* (*rtrancl-while* (λ-. *True*) (λ*s*. *map* (λ*a*. *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* (λ*a*. ((*s*, *a*), *delta a s*)) (σ *n*);
new = *remdups* (*filter* (λ*ss*. *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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *check-eqv-complete*:
assumes *lang n r* = *lang n s wf n r wf n s*
shows *check-eqv r s*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $L :: 's \Rightarrow 'a\ lang$
and $n :: nat$
begin

lemma *step-efficient*[*code*]: $step\ (ws,\ N) =$
$(let$
$(r,\ s) = hd\ ws;$
$new = remdups\ (filter\ (\lambda(r,s).\ (r,s) \notin N)\ (map\ (\lambda a.\ (delta\ a\ r,\ delta\ a\ s))\ (\sigma\ n)))$
$in\ (tl\ ws\ @\ new,\ set\ new \cup N))$
$\langle proof \rangle$

end

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

sublocale *rexp-DA-no-post* $\langle proof \rangle$

end

locale *rexp-DA-sim* = *project set o* σ *wf-atom project lookup*
for $\sigma :: nat \Rightarrow 'a\ list$
and *wf-atom* $:: nat \Rightarrow 'b :: linorder \Rightarrow bool$
and *project* $:: 'a \Rightarrow 'a$
and *lookup* $:: 'b \Rightarrow 'a \Rightarrow bool$ +
fixes *init* $:: 'b\ rexp \Rightarrow 's$
fixes *sim-delta* $:: 's \Rightarrow 's\ list$
fixes *final* $:: 's \Rightarrow bool$
fixes *wf-state* $:: 's \Rightarrow bool$
fixes $L :: 's \Rightarrow 'a\ lang$
fixes *post* $:: 's \Rightarrow 's$
fixes $n :: nat$
assumes *L-init*[*simp*]: $wf\ n\ r \Longrightarrow L\ (init\ r) = lang\ n\ r$
assumes *final-iff-Nil*[*simp*]: $final\ s \longleftrightarrow [] \in L\ s$
assumes *L-wf-state*[*dest*]: $wf\text{-}state\ s \Longrightarrow L\ s \subseteq lists\ (set\ (\sigma\ n))$
assumes *init-wf-state*[*simp*]: $wf\ n\ r \Longrightarrow wf\text{-}state\ (init\ r)$
assumes *L-post*[*simp*]: $wf\text{-}state\ s \Longrightarrow L\ (post\ s) = L\ s$
assumes *wf-state-post*[*simp*]: $wf\text{-}state\ s \Longrightarrow wf\text{-}state\ (post\ s)$

assumes *L-sim-delta*[*simp*]: *wf-state s* $\Longrightarrow$ *map L* (*sim-delta s*) = *map* (λ*a*. *lQuot a* (*L s*)) (σ *n*)
assumes *sim-delta-wf-state*[*simp*]: *wf-state s* $\Longrightarrow$ $\forall$ *s'* $\in$ *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* $\Longrightarrow$ *length* (*sim-delta s*) = *length* (σ *n*)
⟨*proof*⟩

lemma *L-delta*[*simp*]: ⟦*a* $\in$ *set* (σ *n*); *wf-state s*⟧ $\Longrightarrow$ *L* (*delta a s*) = *lQuot a* (*L s*)
⟨*proof*⟩

lemma *delta-wf-state*[*simp*]: ⟦*a* $\in$ *set* (σ *n*); *wf-state s*⟧ $\Longrightarrow$ *wf-state* (*delta a s*)
⟨*proof*⟩

sublocale *rexp-DA* σ *wf-atom project lookup init delta final wf-state post L*
⟨*proof*⟩

sublocale *rexp-DA-sim-no-post*: *rexp-DA-no-post* σ *wf-atom project lookup init delta final wf-state L*
⟨*proof*⟩

end

7 Initial Normalization of the Input

fun *toplevel-inters* **where**
toplevel-inters (*Inter r s*) = *toplevel-inters r* $\cup$ *toplevel-inters s*
| *toplevel-inters r* = {*r*}

lemma *toplevel-inters-nonempty*[*simp*]:
toplevel-inters r $\neq$ {}
⟨*proof*⟩

lemma *toplevel-inters-finite*[*simp*]:
finite (*toplevel-inters r*)
⟨*proof*⟩

context *alphabet*
begin

lemma *toplevel-inters-wf*:
wf n s = ($\forall$ *r*$\in$*toplevel-inters s*. *wf n r*)
⟨*proof*⟩

end

context *project*
begin

lemma *toplevel-inters-lang*:
r ∈ *toplevel-inters s* ⟹ *lang n s* ⊆ *lang n r*
⟨*proof*⟩

lemma *toplevel-inters-lang-INT*:
lang n s = (⋂ *r*∈*toplevel-inters s*. *lang n r*)
⟨*proof*⟩

lemma *toplevel-inters-in-lang*:
w ∈ *lang n s* = (∀ *r*∈*toplevel-inters s*. *w* ∈ *lang n r*)
⟨*proof*⟩

lemma *lang-flatten-INTERSECT-finite*[*simp*]:
finite X ⟹ *w* ∈ *lang n* (*flatten INTERSECT X*) =
(*if X* = {} *then w* ∈ *lists* (Σ *n*) *else* (∀ *r* ∈ *X*. *w* ∈ *lang n r*))
⟨*proof*⟩

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* ∪ *set ys*
⟨*proof*⟩

lemma *sorted-merge-distinct*[*simp*]: ⟦*sorted xs*; *sorted ys*⟧ ⟹ *sorted* (*merge-distinct xs ys*)
⟨*proof*⟩

lemma *distinct-merge-distinct*[*simp*]: ⟦*sorted xs*; *distinct xs*; *sorted ys*; *distinct ys*⟧ ⟹
distinct (*merge-distinct xs ys*)
⟨*proof*⟩

lemma *sorted-list-of-set-merge-distinct*[*simp*]: ⟦*sorted xs*; *distinct xs*; *sorted ys*; *distinct ys*⟧ ⟹
merge-distinct xs ys = *sorted-list-of-set* (*set xs* ∪ *set ys*)
⟨*proof*⟩

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* ⟷ (∃ *a b*. *z* = *f a b* ∧ *x* = *Some a* ∧ *y* = *Some b*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *Pluss-Some*: *Pluss r* = *Some xs* ⟷
  (*Full* ∉ *set xs* ∧ *xs* = *sorted-list-of-set* (*toplevel-summands r* − {*Zero*}))
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *Inters-Some*: *Inters r* = *Some xs* ⟷
  (*Zero* ∉ *set xs* ∧ *xs* = *sorted-list-of-set* (*toplevel-inters r* − {*Full*}))
⟨*proof*⟩

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*⟩

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*⟩

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*)
  ⟨*proof*⟩

lemma *wf-inTimes*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *wf n* (*inTimes r s*)
⟨*proof*⟩

lemma *wf-inStar*[*simp*]: *wf n r* ⟹ *wf n* (*inStar r*)
⟨*proof*⟩

lemma *wf-inInter*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *wf n* (*inInter r s*)
⟨*proof*⟩

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

lemma *wf-inPr*[*simp*]: *wf* (*Suc n*) *r* ⟹ *wf n* (*inPr r*)
⟨*proof*⟩

lemma *wf-inorm*[*simp*]: *wf n r* ⟹ *wf n* (*inorm r*)
⟨*proof*⟩

end

context *project* **begin**

lemma *lang-inPlus*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*inPlus r s*) = *lang n* (*Plus r s*)
⟨*proof*⟩

lemma *lang-inTimes*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*inTimes r s*) = *lang n* (*Times r s*)
⟨*proof*⟩

lemma *lang-inStar*[*simp*]: *wf n r* ⟹ *lang n* (*inStar r*) = *lang n* (*Star r*)
⟨*proof*⟩

lemma *Zero-toplevel-inters*[*dest*]: *Zero* ∈ *toplevel-inters r* ⟹ *lang n r* = {}
⟨*proof*⟩

lemma *toplevel-inters-Full*: ⟦*toplevel-inters r* = {*Full*}; *wf n r*⟧ ⟹ *lang n r* = *lists* (Σ *n*)
⟨*proof*⟩

lemma *toplevel-inters-subset-singleton*[*simp*]: *toplevel-inters r* ⊆ {*s*} ⟷ *toplevel-inters r* = {*s*}
⟨*proof*⟩

lemma *lang-inInter*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*inInter r s*) = *lang n* (*Inter r s*)
⟨*proof*⟩

lemma *lang-inNot*[*simp*]: *wf n r* ⟹ *lang n* (*inNot r*) = *lang n* (*Not r*)

⟨*proof*⟩

lemma *lang-inPr*[*simp*]: *wf* (*Suc n*) *r* ⟹ *lang n* (*inPr r*) = *lang n* (*Pr r*)
⟨*proof*⟩

lemma *lang-inorm*[*simp*]: *wf n r* ⟹ *lang n* (*inorm r*) = *lang n r*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-pnPlus*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*pnPlus r s*) = *lang n* (*Plus r s*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-pnTimes*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*pnTimes r s*) = *lang n* (*Times r s*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-pnInter*[*simp*]: ⟦*wf n r*; *wf n s*⟧ ⟹ *lang n* (*pnInter r s*) = *lang n* (*Inter r s*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-pnNot*[*simp*]: *wf n r* ⟹ *lang n* (*pnNot r*) = *lang n* (*Not r*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *project*) *lang-pnPr*[*simp*]: *wf* (*Suc n*) *r* ⟹ *lang n* (*pnPr r*) = *lang n* (*Pr r*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma (**in** *project*) *lang-pnorm*[*simp*]: *wf n r* ⟹ *lang n* (*pnorm r*) = *lang n r*
  ⟨*proof*⟩

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) = ($\lambda x.\ x - 1$) ‘ *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) = ($\lambda x.\ x - 1$) ‘ *SOV* φ
| *SOV* (*FEXISTS* φ) = ($\lambda x.\ x - 1$) ‘ (*SOV* φ − {*0*})

definition σ = ($\lambda\Sigma$ *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 = ($\lambda\Sigma$ (*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 o* σ Σ) (*n* + *1*) = (*set o* σ Σ) *n*
  ⟨*proof*⟩

locale *formula* = *embed2 set o* (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* ≡
   List.maps (λ*bools*. *map* (λ*a*. (*a*, *bools*)) Σ) (*bool-product-lists n*)

primrec *pre-wf-formula* :: *nat* ⇒ *'a formula* ⇒ *bool* **where**
  pre-wf-formula n (*FQ a m*) = (*a* ∈ *set* Σ ∧ *m* < *n*)
| *pre-wf-formula n* (*FLess m1 m2*) = (*m1* < *n* ∧ *m2* < *n*)
| *pre-wf-formula n* (*FIn m M*) = (*m* < *n* ∧ *M* < *n*)
| *pre-wf-formula n* (*FNot* φ) = *pre-wf-formula n* φ
| *pre-wf-formula n* (*FOr* φ$_1$ φ$_2$) = (*pre-wf-formula n* φ$_1$ ∧ *pre-wf-formula n* φ$_2$)
| *pre-wf-formula n* (*FAnd* φ$_1$ φ$_2$) = (*pre-wf-formula n* φ$_1$ ∧ *pre-wf-formula n* φ$_2$)
| *pre-wf-formula n* (*FExists* φ) = (*pre-wf-formula* (*n* + *1*) φ ∧ *0* ∈ *FOV* φ ∧ *0* ∉ *SOV* φ)
| *pre-wf-formula n* (*FEXISTS* φ) = (*pre-wf-formula* (*n* + *1*) φ ∧ *0* ∉ *FOV* φ ∧ *0* ∈ *SOV* φ)

abbreviation *closed* ≡ *pre-wf-formula 0*

definition [*simp*]: *wf-formula n* φ ≡ *pre-wf-formula n* φ ∧ *FOV* φ ∩ *SOV* φ = {}

lemma *max-idx-vars*: *pre-wf-formula n* φ ⟹ ∀ *p* ∈ *FOV* φ ∪ *SOV* φ. *p* < *n*
  ⟨*proof*⟩

lemma *finite-FOV*: *finite* (*FOV* φ)
  ⟨*proof*⟩

9.3 ENC

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

lemma *wf-rexp-valid-ENC*: *n* = *0* ∨ *p* < *n* ⟹ *wf n* (*valid-ENC n p*)
  ⟨*proof*⟩

definition *ENC* :: *nat* ⇒ *nat set* ⇒ (*'a atom*) *rexp* **where**
  ENC n V = *flatten INTERSECT* (*valid-ENC n* ‘ *V*)

lemma *wf-rexp-ENC*: ⟦*finite V*; *n* = *0* ∨ (∀ *v* ∈ *V*. *v* < *n*)⟧ ⟹ *wf n* (*ENC n V*)
  ⟨*proof*⟩

lemma *enc-atom-σ-eq*: *i* < *length w* ⟹
 (*length I* = *n* ∧ *p* ∈ *set* Σ) ⟷ *enc-atom I i p* ∈ *set* (σ Σ *n*)
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *enc-atom-lang-Arbitrary-Except-False*: ⟦*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 False*))
⟨*proof*⟩

lemma *AQ-D*:
assumes *v* ∈ *lang n* (*Atom* (*AQ m a*)) *m* < *n a* ∈ *set* Σ
shows ∃ *x*. *v* = [*x*] ∧ *fst x* = *a* ∧ *snd x* ! *m*
⟨*proof*⟩

lemma *Arbitrary-ExceptD*:
assumes *v* ∈ *lang n* (*Atom* (*Arbitrary-Except r b*)) *r* < *n*
shows ∃ *x*. *v* = [*x*] ∧ *snd x* ! *r* = *b*
⟨*proof*⟩

lemma *Arbitrary-Except2D*:

assumes $v \in lang\ n\ (Atom\ (Arbitrary\text{-}Except2\ r\ s))\ r < n\ s < n$
shows $\exists x.\ v = [x] \wedge snd\ x\ !\ r \wedge snd\ x\ !\ s$
⟨*proof*⟩

lemma *star-Arbitrary-ExceptD*:
$[\![v \in star\ (lang\ n\ (Atom\ (Arbitrary\text{-}Except\ r\ b)));\ r < n;\ i < length\ v]\!] \Longrightarrow$
$snd\ (v\ !\ i)\ !\ r = b$
⟨*proof*⟩

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* ∈ *set w*. *a* ∈ *set* Σ) ∧
(∀ *x* ∈ *set I*. *case x of Inl p* ⇒ *p* < *length w* | *Inr P* ⇒ ∀ *p* ∈ *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* φ1 φ2) = ((*w*, *I*) ⊨ φ1 ∨ (*w*, *I*) ⊨ φ2)
| (*w*, *I*) ⊨ (*FAnd* φ1 φ2) = ((*w*, *I*) ⊨ φ1 ∧ (*w*, *I*) ⊨ φ2)
| (*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 (λp 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 (λi.*
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}*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *positions-in-row-length*: ∃!*p. p < length w ∧ snd (w ! p) ! i ⟹*
the-elem (positions-in-row w i) < length w
⟨*proof*⟩

lemma *dec-interp-Inl*: ⟦*i ∈ FO*; *i < n*⟧ ⟹ ∃ *p. dec-interp n FO x ! i = Inl p*
⟨*proof*⟩

lemma *dec-interp-not-Inr*: ⟦*dec-interp n FO x ! i = Inr P*; *i ∈ FO*; *i < n*⟧ ⟹ *False*
⟨*proof*⟩

lemma *dec-interp-Inr*: ⟦*i ∉ FO*; *i < n*⟧ ⟹ ∃ *P. dec-interp n FO x ! i = Inr P*
⟨*proof*⟩

lemma *dec-interp-not-Inl*: ⟦*dec-interp n FO x ! i = Inl p*; *i ∉ FO*; *i < n*⟧ ⟹ *False*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Inr-dec-interp-length*: ⟦*Inr P ∈ set (dec-interp n FO w)*; *p ∈ P*⟧ ⟹ *p < length w*
⟨*proof*⟩

lemma *the-elem-Collect*[*simp*]:
assumes ∃!*x. P x*
shows *the-elem (Collect P) = (The P)*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dec-word-enc*: *dec-word* (*enc* (*w*, *I*)) = *w*
⟨*proof*⟩

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*⟩

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*′
⟨*proof*⟩

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*′
⟨*proof*⟩

lemma *length-dec-interp*[*simp*]: *length* (*dec-interp n FO x*) = *n*
⟨*proof*⟩

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

lemma *set-σD*[*simp*]: (*a*, *bs*) ∈ *set* (*σ* Σ *n*) ⟹ *a* ∈ *set* Σ
⟨*proof*⟩

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*) ∧

$(\forall\, i \in SO.\ case\ I\ !\ i\ of\ Inl\ \text{-} \Rightarrow False \mid Inr\ \text{-} \Rightarrow True)\}$
(**is** $?L = ?R$)
⟨*proof*⟩

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*⟩

10.2 Welldefinedness of enc wrt. Models

lemma *enc-alt-def*:
enc (w, x # I) = map-index (λn (a, bs). (a, (case x of Inl p ⇒ n = p | Inr P ⇒ n ∈ P) # bs)) (enc (w, I))
⟨*proof*⟩

lemma *enc-extend-interp*: *enc (w, I) = enc (w′, I′) ⟹ enc (w, x # I) = enc (w′, x # I′)*
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *enc-word-length*: *enc (w, I) = enc (w′, I′) ⟹ length w = length w′*
⟨*proof*⟩

lemma *enc-length*:
assumes *w ≠ [] enc (w, I) = enc (w′, I′)*
shows *length I = length I′*
⟨*proof*⟩

lemma *wf-interp-for-formula-FOr*:
wf-interp-for-formula (*w*, *I*) (*FOr φ1 φ2*) =
(*wf-interp-for-formula* (*w*, *I*) *φ1* ∧ *wf-interp-for-formula* (*w*, *I*) *φ2*)
⟨*proof*⟩

lemma *wf-interp-for-formula-FAnd*:
wf-interp-for-formula (*w*, *I*) (*FAnd φ1 φ2*) =
(*wf-interp-for-formula* (*w*, *I*) *φ1* ∧ *wf-interp-for-formula* (*w*, *I*) *φ2*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma $lang_{M2L}$*-FOr*:
assumes *wf-formula n* (*FOr* φ_1 φ_2)
shows $lang_{M2L}$ *n* (*FOr* φ_1 φ_2) ⊆
($lang_{M2L}$ *n* φ_1 ∪ $lang_{M2L}$ *n* φ_2) ∩ {*enc* (*w*, *I*) | *w I*. *length I* = *n* ∧ *wf-interp-for-formula* (*w*, *I*) (*FOr* φ_1 φ_2)}
(**is** - ⊆ (*?L1* ∪ *?L2*) ∩ *?ENC*)
⟨*proof*⟩

lemma $lang_{M2L}$*-FAnd*:
assumes *wf-formula n* (*FAnd* φ_1 φ_2)
shows $lang_{M2L}$ *n* (*FAnd* φ_1 φ_2) ⊆
$lang_{M2L}$ *n* φ_1 ∩ $lang_{M2L}$ *n* φ_2 ∩ {*enc* (*w*, *I*) | *w I*. *length I* = *n* ∧ *wf-interp-for-formula* (*w*, *I*) (*FAnd* φ_1 φ_2)}
(**is** - ⊆ *?L1* ∩ *?L2* ∩ *?ENC*)
⟨*proof*⟩

10.3 From M2L to Regular expressions

fun *rexp-of* :: *nat* ⇒ *'a formula* ⇒ (*'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* 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) = *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* 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) = *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* 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) = *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*⟩

lemma *wf-rexp-of*: *wf-formula n* φ ⟹ *wf n* (*rexp-of n* φ)
⟨*proof*⟩

lemma *wf-rexp-of-alt*: *wf-formula n* φ ⟹ *wf n* (*rexp-of-alt n* φ)
⟨*proof*⟩

lemma *wf-rexp-of'*: *wf-formula n φ ⟹ wf n (rexp-of' n φ)*
⟨*proof*⟩

lemma *wf-rexp-of-alt'*: *wf-formula n φ ⟹ wf n (rexp-of-alt' n φ)*
⟨*proof*⟩

lemma *wf-rexp-of''*: *wf-formula n φ ⟹ wf n (rexp-of'' n φ)*
⟨*proof*⟩

lemma *ENC-Not*: *ENC n (FOV (FNot φ)) = ENC n (FOV φ)*
⟨*proof*⟩

lemma *ENC-And*:
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*⟩

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*⟩

lemma *project-enc*: *map π (enc (w, x # I)) = enc (w, I)*
⟨*proof*⟩

lemma *list-list-eqI*:
assumes *∀(-, x) ∈ set xs. x ≠ [] ∀(-, y) ∈ set ys. y ≠ []*
map (λ(-, x). hd x) xs = map (λ(-, x). hd x) ys map π xs = map π ys
shows *xs = ys*
⟨*proof*⟩

lemma *project-enc-extend*:
assumes *map π x = enc (w, I) ∀(-, x) ∈ set x. x ≠ []*
shows *x = enc (w, Inr (positions-in-row x 0) # I)*
⟨*proof*⟩

lemma *ENC-Exists*:
wf-formula n (FExists φ) ⟹ lang n (ENC n (FOV (FExists φ))) − {[]} = map π ‘ lang (Suc n) (ENC (Suc n) (FOV φ)) − {[]}
⟨*proof*⟩

lemma *ENC-EXISTS*:
wf-formula n (FEXISTS φ) ⟹ lang n (ENC n (FOV (FEXISTS φ))) − {[]} = map π ‘ lang (Suc n) (ENC (Suc n) (FOV φ)) − {[]}
⟨*proof*⟩

lemma *map-project-empty*: *map π ‘ A − {[]} = map π ‘ (A − {[]})*
⟨*proof*⟩

lemma *lang$_{M2L}$-rexp-of-rexp-of'*:
wf-formula n φ ⟹ lang n (rexp-of n φ) − {[]} = lang n (rexp-of' n φ) − {[]}
⟨*proof*⟩

lemma *Int-Diff-both*: *A ∩ B − C = (A − C) ∩ (B − C)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *map-project-Int-ENC*:
assumes *0 ∉ X X ⊆ {0 ..< n + 1} Z ⊆ lists ((set o σ Σ) (n + 1))*
shows *map π ‘ (Z ∩ lang (n + 1) (ENC (n + 1) X) − {[]}) =*
map π ‘ Z ∩ lang n (ENC n ((λx. x − 1) ‘ X)) − {[]}
⟨*proof*⟩

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*⟩

abbreviation $\mathfrak{L} \equiv$ *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) − {[]}
⟨*proof*⟩

theorem *lang$_{M2L}$-rexp-of'*: *wf-formula n φ ⟹ lang$_{M2L}$ n φ = lang n (rexp-of' n φ) − {[]}*
⟨*proof*⟩

theorem *lang$_{M2L}$-rexp-of''*: *wf-formula n φ ⟹ lang$_{M2L}$ n φ = lang n (rexp-of'' n φ) − {[]}*
⟨*proof*⟩

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)
  ⟨*proof*⟩

lemma *FOV-nNot*[*simp*]: *FOV* (*nNot* φ) = *FOV* (*FNot* φ)
  ⟨*proof*⟩

lemma *SOV-nNot*[*simp*]: *SOV* (*nNot* φ) = *SOV* (*FNot* φ)
  ⟨*proof*⟩

lemma *pre-wf-formula-nNot*[*simp*]: *pre-wf-formula n* (*nNot* φ) = *pre-wf-formula n* (*FNot* φ)
  ⟨*proof*⟩

lemma *FOV-norm*[*simp*]: *FOV* (*norm* φ) = *FOV* φ
  ⟨*proof*⟩

lemma *SOV-norm*[*simp*]: *SOV* (*norm* φ) = *SOV* φ
  ⟨*proof*⟩

lemma *pre-wf-formula-norm*[*simp*]: *pre-wf-formula n* (*norm* φ) = *pre-wf-formula n* φ
  ⟨*proof*⟩

lemma *satisfies-norm*[*simp*]: *satisfies* (*w*, *I*) (*norm* φ) = *satisfies* (*w*, *I*) φ
  ⟨*proof*⟩

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

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)
⟨*proof*⟩

lemma *enum-not-empty*[*simp*]: *Enum.enum* ≠ [] (**is** *?enum* ≠ [])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lang-Plus-Zero*: *lang* Σ *n* (*Plus r One*) = *lang* Σ *n* (*Plus s One*) ⟷ *lang* Σ *n r* − {[]} = *lang* Σ *n s* − {[]}
⟨*proof*⟩

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''*]

⟨*ML*⟩

global-interpretation *D*: *rexp-DFA* σ Σ *wf-atom* Σ π *lookup* λ*x*. «*pnorm* (*inorm x*)»
λ*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) (λ*a r*. «$\mathfrak{D}$ Σ *a r*») *pnorm n*
and *closure* = *rexp-DA.closure* (σ Σ) (λ*a r*. «$\mathfrak{D}$ Σ *a r*») *final pnorm n*
and *check-eqvRE* = *rexp-DA.check-eqv* (σ Σ) (λ*x*. «*pnorm* (*inorm x*)») (λ*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) (*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*
⟨*proof*⟩

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 (*λw. 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*
⟨*proof*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

global-interpretation *D*: *rexp-DA-no-post σ* Σ *wf-atom* Σ *π lookup λx. pnorm* (*inorm x*)
λa r. pnorm (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) (*λa r. pnorm* (𝔇 Σ *a r*)) *id n*
and *closure* = *rexp-DA.closure* (*σ* Σ) (*λa r. pnorm* (𝔇 Σ *a r*)) *final id n*
and *check-eqvRE* = *rexp-DA.check-eqv* (*σ* Σ) (*λx. pnorm* (*inorm x*)) (*λa r. pnorm* (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) (*λa r. pnorm* (𝔇 Σ *a r*)) *id n*
and *closure-invariant* = *rexp-DA.closure-invariant* (*σ* Σ) (*λa r. pnorm* (𝔇 Σ *a*

r)) *final id n*
and *counterexampleRE* = *rexp-DA.counterexample* (σ Σ) (λx. *pnorm* (*inorm* x)) (λa r. *pnorm* ($\mathfrak{D}$ Σ a r)) *final id n*
and *reachable* = *rexp-DA.reachable* (σ Σ) (λx. *pnorm* (*inorm* x)) (λa r. *pnorm* ($\mathfrak{D}$ Σ a r)) *id n*
and *automaton* = *rexp-DA.automaton* (σ Σ) (λx. *pnorm* (*inorm* x)) (λa r. *pnorm* ($\mathfrak{D}$ Σ a r)) *id n*
⟨*proof*⟩

definition *check-eqv* **where**
check-eqv n φ ψ $\longleftrightarrow$ *wf-formula n* (*FOr* φ ψ) $\wedge$
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 (λw. *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 $\Longrightarrow$ $\Phi.lang_{M2L}$ *n* φ = $\Phi.lang_{M2L}$ *n* ψ
⟨*proof*⟩

⟨*ML*⟩

global-interpretation *D*: *rexp-DA-no-post* σ Σ *wf-atom* Σ π *lookup*
λx. *pnorm-dual* (*rexp-dual-of* (*inorm* x)) λa 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) (λa r. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ a r)) *id n*
and *closure* = *rexp-DA.closure* (σ Σ) (λa 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))) (λa 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) (λa r. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ a r)) *id n*
and *closure-invariant* = *rexp-DA.closure-invariant* (σ Σ) (λa 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))) (λa 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))) (λa 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))) (λa r. *pnorm-dual* (*Co*$\mathfrak{D}$ Σ a r)) *id n*

⟨*proof*⟩

definition *check-eqv* **where**
check-eqv n φ ψ ⟷ *wf-formula n* (*FOr φ ψ*) ∧
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 (λ*w. 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* ⟹ Φ.*lang*$_{M2L}$ *n φ* = Φ.*lang*$_{M2L}$ *n ψ*
⟨*proof*⟩

⟨*ML*⟩

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*⟩

lemma *sconst-collapse*: *y ## sconst y* = *sconst y*
⟨*proof*⟩

lemma *shift-sconst-inj*: ⟦*length x* = *length y*; *x* @− *sconst z* = *y* @− *sconst z*⟧ ⟹ *x* = *y*
⟨*proof*⟩

context *formula*
begin

definition *any* ≡ *hd* Σ

lemma *any*-Σ[*simp*]: *any* ∈ *set* Σ
⟨*proof*⟩

lemma *any*-σ[*simp*]: *length bs* = *n* ⟹ (*any*, *bs*) ∈ *set* (σ Σ *n*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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

lemma *sset-singleton*[*simp*]: *sset s* ⊆ {*x*} ⟷ *sset s* = {*x*}
⟨*proof*⟩

lemma *drop-sconstE*: ⟦*drop n w* @− *sconst y* = *sconst y*; *p* < *length w*; ¬ *p* < *n*⟧
⟹ *w* ! *p* = *y*
⟨*proof*⟩

lemma *less-length-cut-same*:
⟦(*w* @− *sconst y*) !! *p* = *a*⟧ ⟹ *a* = *y* ∨ (*p* < *length* (*cut-same y* (*w* @− *sconst y*)) ∧ *w* ! *p* = *a*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*) = []
⟨*proof*⟩

lemma *cut-same-stop*[*simp*]:
assumes $x \neq y$
shows *cut-same x* (*xs* @− *y* ## *sconst x*) = *xs* @ [*y*] (**is** *cut-same x ?s* = -)
⟨*proof*⟩

lemma *cut-same-shift-sconst*: ∃ *n*. *w* = *cut-same x* (*w* @− *sconst x*) @ *replicate n x*
⟨*proof*⟩

lemma *set-cut-same*: *set* (*cut-same x* (*w* @− *sconst x*)) ⊆ *set w*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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

lemma *sdrop-sconst*: *sdrop n s* = *sconst x* ⟹ *n* ≤ *m* ⟹ *s* !! *m* = *x*
⟨*proof*⟩

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*⟩

lemma *tl-enc*[*simp*]:

assumes $\forall x \in set\ (x\ \#\ I).\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True$
shows $SAMEQUOT\ (any,\ replicate\ (length\ I)\ False)\ (map\ \pi\ `\ enc\ (w,\ x\ \#\ I)) = enc\ (w,\ I)$
⟨*proof*⟩

lemma *encD*:
$[\![v \in enc\ (w,\ I);\ (\forall x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True)]\!] \Longrightarrow$
$v = map\ (case\text{-}prod\ (enc\text{-}atom\ I))\ (zip\ [0\ ..<\ length\ v]\ (stake\ (length\ v)\ (w\ @\text{-}sconst\ any)))$
⟨*proof*⟩

lemma *enc-Inl*: $[\![x \in enc\ (w,\ I);\ (\forall x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True);$
$m < length\ I;\ I\ !\ m = Inl\ p]\!] \Longrightarrow p < length\ x \wedge snd\ (x\ !\ p)\ !\ m$
⟨*proof*⟩

lemma *enc-Inr*: **assumes** $x \in enc\ (w,\ I)\ \forall x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True$
$M < length\ I\ I\ !\ M = Inr\ P$
shows $p \in P \longleftrightarrow p < length\ x \wedge snd\ (x\ !\ p)\ !\ M$
⟨*proof*⟩

lemma *enc-length*:
assumes $enc\ (w,\ I) = enc\ (w',\ I')$
shows $length\ I = length\ I'$
⟨*proof*⟩

lemma *enc-stream-enc*:
$[\![(\forall x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True);$
$(\forall x \in set\ I'.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True);$
$enc\ (w,\ I) = enc\ (w',\ I')]\!] \Longrightarrow stream\text{-}enc\ (w,\ I) = stream\text{-}enc\ (w',\ I')$
⟨*proof*⟩

abbreviation $wf\text{-}interp\ w\ I \equiv$
$((\forall a \in set\ w.\ a \in set\ \Sigma) \wedge (\forall x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True))$

fun $wf\text{-}interp\text{-}for\text{-}formula :: 'a\ interp \Rightarrow 'a\ formula \Rightarrow bool$ **where**
$wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I)\ \varphi =$
$(wf\text{-}interp\ w\ I \wedge$
$(\forall n \in FOV\ \varphi.\ case\ I\ !\ n\ of\ Inl\ \text{-} \Rightarrow True \mid \text{-} \Rightarrow False) \wedge$
$(\forall n \in SOV\ \varphi.\ case\ I\ !\ n\ of\ Inl\ \text{-} \Rightarrow False \mid Inr\ \text{-} \Rightarrow True))$

fun $satisfies :: 'a\ interp \Rightarrow 'a\ formula \Rightarrow bool$ (**infix** $\models$ *50*) **where**
$(w,\ I) \models FQ\ a\ m = ((case\ I\ !\ m\ of\ Inl\ p \Rightarrow if\ p < length\ w\ then\ w\ !\ p\ else\ any) = a)$
$\mid (w,\ I) \models FLess\ m1\ m2 = ((case\ I\ !\ m1\ of\ Inl\ p \Rightarrow p) < (case\ I\ !\ m2\ of\ Inl\ p \Rightarrow p))$
$\mid (w,\ I) \models FIn\ m\ M = ((case\ I\ !\ m\ of\ Inl\ p \Rightarrow p) \in (case\ I\ !\ M\ of\ Inr\ P \Rightarrow P))$
$\mid (w,\ I) \models FNot\ \varphi = (\neg\ (w,\ I) \models \varphi)$

| $(w,\ I) \models (FOr\ \varphi_1\ \varphi_2) = ((w,\ I) \models \varphi_1 \vee (w,\ I) \models \varphi_2)$
| $(w,\ I) \models (FAnd\ \varphi_1\ \varphi_2) = ((w,\ I) \models \varphi_1 \wedge (w,\ I) \models \varphi_2)$
| $(w,\ I) \models (FExists\ \varphi) = (\exists\, p.\ (w,\ Inl\ p\ \#\ I) \models \varphi)$
| $(w,\ I) \models (FEXISTS\ \varphi) = (\exists\, P.\ finite\ P \wedge (w,\ Inr\ P\ \#\ I) \models \varphi)$

definition $lang_{WS1S} :: nat \Rightarrow 'a\ formula \Rightarrow ('a \times bool\ list)\ list\ set$ **where**
$lang_{WS1S}\ n\ \varphi = \bigcup \{enc\ (w,\ I) \mid w\ I\ .\ length\ I = n \wedge wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I)\ \varphi \wedge (w,\ I) \models \varphi\}$

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

lemma *enc-set-σ*: $[\![x \in enc\ (w,\ I);\ (\forall\, x \in set\ I.\ case\ x\ of\ Inr\ P \Rightarrow finite\ P \mid \text{-} \Rightarrow True);$
$length\ I = n;\ a \in set\ x;\ set\ w \subseteq set\ \Sigma]\!] \Longrightarrow a \in set\ (\sigma\ \Sigma\ n)$
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *positions-in-row-unique*: ∃!*p*. *snd* (*s* !! *p*) ! *i* ⟹
the-elem (*positions-in-row s i*) = (*THE p*. *snd* (*s* !! *p*) ! *i*)
⟨*proof*⟩

lemma *positions-in-row-nth*: ∃!*p*. *snd* (*s* !! *p*) ! *i* ⟹
snd (*s* !! *the-elem* (*positions-in-row s i*)) ! *i*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ∈ *FO*; *i* < *n*] ⟹ ∃ *p*. *stream-dec n FO s* ! *i* = *Inl p*
⟨*proof*⟩

lemma *stream-dec-not-Inr*: [*stream-dec n FO s* ! *i* = *Inr P*; *i* ∈ *FO*; *i* < *n*] ⟹

False
⟨*proof*⟩

lemma *stream-dec-Inr*: ⟦*i* ∉ *FO*; *i* < *n*⟧ ⟹ ∃ *P*. *stream-dec n FO s* ! *i* = *Inr P*
⟨*proof*⟩

lemma *stream-dec-not-Inl*: ⟦*stream-dec n FO s* ! *i* = *Inl p*; *i* ∉ *FO*; *i* < *n*⟧ ⟹ *False*
⟨*proof*⟩

lemma *Inr-dec-finite*: ⟦∀ *i*<*n*. *finite* {*p*. *snd* (*s* !! *p*) ! *i*}; *Inr P* ∈ *set* (*stream-dec n FO s*)⟧ ⟹
finite P
⟨*proof*⟩

lemma *enc-atom-dec*:
⟦∀ *p*. *length* (*snd* (*s* !! *p*)) = *n*; ∀ *i* ∈ *FO*. *i* < *n* ⟶ (∃!*p*. *snd* (*s* !! *p*) ! *i*); *a* = *fst* (*s* !! *p*)⟧ ⟹
enc-atom (*stream-dec n FO s*) *p a* = *s* !! *p*
⟨*proof*⟩

lemma *length-stream-dec*[*simp*]: *length* (*stream-dec n FO x*) = *n*
⟨*proof*⟩

lemma *stream-enc-dec*:
⟦∃ *n*. *sdrop n* (*smap fst s*) = *sconst any*;
stream-all (λ*x*. *length* (*snd x*) = *n*) *s*; ∀ *i* ∈ *FO*. (∃!*p*. *snd* (*s* !! *p*) ! *i*)⟧ ⟹
stream-enc (*dec-word s*, *stream-dec n FO s*) = *s*
⟨*proof*⟩

lemma *stream-enc-unique*:
i < *length I* ⟹ ∃ *p*. *I* ! *i* = *Inl p* ⟹ ∃!*p*. *snd* (*stream-enc* (*w*, *I*) !! *p*) ! *i*
⟨*proof*⟩

lemma *stream-dec-enc-Inl*:
⟦*stream-dec n FO* (*stream-enc* (*w*, *I*)) ! *i* = *Inl p*′; *I* ! *i* = *Inl p*; *i* ∈ *FO*; *i* < *n*; *length I* = *n*⟧ ⟹
p = *p*′
⟨*proof*⟩

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*′
⟨*proof*⟩

lemma *Collect-snth*: {*p*. *P* ((*x* ## *s*) !! *p*)} ⊆ {*0*} ∪ *Suc* ‘ {*p*. *P* (*s* !! *p*)}
⟨*proof*⟩

lemma *finite-True-in-row*: $\forall\, i < n.$ *finite* $\{p.$ *snd* $((w$ @− *sconst* $(any,$ *replicate* n *False*)) !! $p)$! $i\}$
⟨*proof*⟩

lemma *lang-ENC*:
assumes $FO \subseteq \{0 ..< n\}$ $SO \subseteq \{0 ..< n\} - FO$
shows *lang* n $(ENC\ n\ FO) = \bigcup\{enc\ (w,\ I) \mid w\ I .$ *length* $I = n \wedge$ *wf-interp* $w\ I$
$\wedge$
$(\forall\, i \in FO.$ *case* I ! i *of Inl* - ⇒ *True* | *Inr* - ⇒ *False*) $\wedge$
$(\forall\, i \in SO.$ *case* I ! i *of Inl* - ⇒ *False* | *Inr* - ⇒ *True*)$\}$
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *lang-ENC-formula*:
assumes *wf-formula* $n\ \varphi$
shows *lang* n $(ENC\ n\ (FOV\ \varphi)) = \bigcup\{enc\ (w,\ I) \mid w\ I .$ *length* $I = n \wedge$ *wf-interp-for-formula* $(w,\ I)\ \varphi\}$
⟨*proof*⟩

13.2 Welldefinedness of enc wrt. Models

lemma *wf-interp-for-formula-FExists*:
⟦*wf-formula* $(length\ I)$ $(FExists\ \varphi)$⟧⟹
wf-interp-for-formula $(w,\ I)$ $(FExists\ \varphi) \longleftrightarrow (\forall\, p.$ *wf-interp-for-formula* $(w,$ *Inl* $p\ \#\ I)\ \varphi)$
⟨*proof*⟩

lemma *wf-interp-for-formula-any-Inl*: *wf-interp-for-formula* $(w,$ *Inl* $p\ \#\ I)\ \varphi \Longrightarrow$
$\forall\, p.$ *wf-interp-for-formula* $(w,$ *Inl* $p\ \#\ I)\ \varphi$
⟨*proof*⟩

lemma *wf-interp-for-formula-FEXISTS*:
⟦*wf-formula* $(length\ I)$ $(FEXISTS\ \varphi)$⟧⟹
wf-interp-for-formula (w, I) $(FEXISTS\ \varphi) \longleftrightarrow (\forall\, P.$ *finite* $P \longrightarrow$ *wf-interp-for-formula* $(w,$ *Inr* $P\ \#\ I)\ \varphi)$
⟨*proof*⟩

lemma *wf-interp-for-formula-any-Inr*: *wf-interp-for-formula* $(w,$ *Inr* $P\ \#\ I)\ \varphi \Longrightarrow$
$\forall\, P.$ *finite* $P \longrightarrow$ *wf-interp-for-formula* $(w,$ *Inr* $P\ \#\ I)\ \varphi$
⟨*proof*⟩

lemma *wf-interp-for-formula-FOr*:
wf-interp-for-formula $(w,\ I)$ $(FOr\ \varphi 1\ \varphi 2) =$
$(wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I)\ \varphi 1 \wedge wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I)\ \varphi 2)$
⟨*proof*⟩

lemma *wf-interp-for-formula-FAnd*:
wf-interp-for-formula $(w,\ I)$ $(FAnd\ \varphi 1\ \varphi 2) =$
$(wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I)\ \varphi 1 \wedge wf\text{-}interp\text{-}for\text{-}formula\ (w,\ I)\ \varphi 2)$

⟨*proof*⟩

lemma *enc-wf-interp*:
⟦*wf-formula* (*length I*) φ; *wf-interp-for-formula* (*w*, *I*) φ; *x* ∈ *enc* (*w*, *I*)⟧ ⟹
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
⟨*proof*⟩

lemma *enc-atom-welldef*: ∀ *x a*. *enc-atom I x a* = *enc-atom I′ x a* ⟹ *m* < *length I* ⟹
(*case* (*I* ! *m*, *I′* ! *m*) *of* (*Inl p*, *Inl q*) ⇒ *p* = *q* | (*Inr P*, *Inr Q*) ⇒ *P* = *Q* | - ⇒ *True*)
⟨*proof*⟩

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*) ⊨ φ ⟷ (*w′*, *I′*) ⊨ φ
⟨*proof*⟩

lemma $lang_{WS1S}$-*FOr*:
assumes *wf-formula n* (*FOr* φ_1 φ_2)
shows $lang_{WS1S}$ *n* (*FOr* φ_1 φ_2) ⊆
($lang_{WS1S}$ *n* φ_1 ∪ $lang_{WS1S}$ *n* φ_2) ∩ ⋃{*enc* (*w*, *I*) | *w I*. *length I* = *n* ∧ *wf-interp-for-formula* (*w*, *I*) (*FOr* φ_1 φ_2)}
(**is** - ⊆ (*?L1* ∪ *?L2*) ∩ *?ENC*)
⟨*proof*⟩

lemma $lang_{WS1S}$-*FAnd*:
assumes *wf-formula n* (*FAnd* φ_1 φ_2)
shows $lang_{WS1S}$ *n* (*FAnd* φ_1 φ_2) ⊆
$lang_{WS1S}$ *n* φ_1 ∩ $lang_{WS1S}$ *n* φ_2 ∩ ⋃{*enc* (*w*, *I*) | *w I*. *length I* = *n* ∧ *wf-interp-for-formula* (*w*, *I*) (*FAnd* φ_1 φ_2)}
⟨*proof*⟩

13.3 From WS1S to Regular expressions

fun *rexp-of* :: *nat* ⇒ *'a formula* ⇒ (*'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 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) = 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 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) = 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 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) = 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*⟩

lemma *enc-eq-welldef*:
$[\![enc\ (w,\ I) = enc\ (w',\ I')$; *wf-formula* $(length\ I)\ \varphi$; *wf-interp-for-formula* $(w,\ I)$ φ ;*wf-interp-for-formula* $(w',\ I')\ \varphi]\!] \Longrightarrow$
$(w,\ I) \models \varphi \longleftrightarrow (w',\ I') \models \varphi$
⟨*proof*⟩

lemma *enc-welldef*:
$[\![x \in enc\ (w,\ I)$; $x \in enc\ (w',\ I')$; $length\ I = length\ I'$; *wf-formula* $(length\ I)\ \varphi$;
wf-interp-for-formula $(w,\ I)\ \varphi$;*wf-interp-for-formula* $(w',\ I')\ \varphi]\!] \Longrightarrow$
$(w,\ I) \models \varphi \longleftrightarrow (w',\ I') \models \varphi$
⟨*proof*⟩

lemma *wf-rexp-of*: *wf-formula* $n\ \varphi \Longrightarrow wf\ n\ (rexp\text{-}of\ n\ \varphi)$
⟨*proof*⟩

theorem $lang_{WS1S}$*-rexp-of*: *wf-formula* $n\ \varphi \Longrightarrow lang_{WS1S}\ n\ \varphi = lang\ n\ (rexp\text{-}of\ n\ \varphi)$
(**is** - ⟹ - = *?L* $n\ \varphi$)
⟨*proof*⟩

lemma *wf-rexp-of-alt*: *wf-formula* $n\ \varphi \Longrightarrow wf\ n\ (rexp\text{-}of\text{-}alt\ n\ \varphi)$
⟨*proof*⟩

lemma *wf-rexp-of'*: *wf-formula* $n\ \varphi \Longrightarrow wf\ n\ (rexp\text{-}of'\ n\ \varphi)$
⟨*proof*⟩

lemma *wf-rexp-of-alt'*: *wf-formula* $n\ \varphi \Longrightarrow wf\ n\ (rexp\text{-}of\text{-}alt'\ n\ \varphi)$
⟨*proof*⟩

lemma *wf-rexp-of''*: *wf-formula* $n\ \varphi \Longrightarrow wf\ n\ (rexp\text{-}of''\ n\ \varphi)$
⟨*proof*⟩

lemma *ENC-FNot*: $ENC\ n\ (FOV\ (FNot\ \varphi)) = ENC\ n\ (FOV\ \varphi)$
⟨*proof*⟩

lemma *ENC-FAnd*:
wf-formula $n\ (FAnd\ \varphi\ \psi) \Longrightarrow lang\ n\ (ENC\ n\ (FOV\ (FAnd\ \varphi\ \psi))) \subseteq lang\ n\ (ENC\ n\ (FOV\ \varphi)) \cap lang\ n\ (ENC\ n\ (FOV\ \psi))$
⟨*proof*⟩

lemma *ENC-FOr*:
wf-formula $n\ (FOr\ \varphi\ \psi) \Longrightarrow lang\ n\ (ENC\ n\ (FOV\ (FOr\ \varphi\ \psi))) \subseteq lang\ n\ (ENC\ n\ (FOV\ \varphi)) \cap lang\ n\ (ENC\ n\ (FOV\ \psi))$
⟨*proof*⟩

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*⟩

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*⟩

lemma *lang$_{WS1S}$-rexp-of-rexp-of′*:
wf-formula n φ ⟹ lang n (rexp-of n φ) = lang n (rexp-of′ n φ)
⟨*proof*⟩

lemma *SAMEQUTO-UN*[*simp*]: *SAMEQUOT x (⋃ y ∈ A. B y) = (⋃ y ∈ A. SAMEQUOT x (B y))*
⟨*proof*⟩

lemma *finite-positions-in-row*[*simp*]:
n > 0 ⟹ finite (positions-in-row (x @− sconst (any, replicate n False)) 0)
⟨*proof*⟩

lemma *fin-cut-same-snoc*: *fin-cut-same x (xs @ [y]) = (if x = y then fin-cut-same x xs else xs @ [y])*
⟨*proof*⟩

lemma *fin-cut-same-idem*: *fin-cut-same x (fin-cut-same x xs) = fin-cut-same x xs*
⟨*proof*⟩

lemma *cut-same-sconst*: *cut-same x (xs @− sconst x) = fin-cut-same x xs*
⟨*proof*⟩

lemma *length-cut-same*: *length (cut-same x s) = (LEAST n. sdrop n s = sconst x)*
⟨*proof*⟩

lemma *enc-alt*: *wf-interp w I ⟹*
x ∈ enc (w, I) ⟷ x @− sconst ((any, replicate (length I) False)) = stream-enc (w, I)
⟨*proof*⟩

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*⟩

lemma *project-enc-extend*:

fixes x I
defines $n \equiv$ *length* I
defines $z \equiv \lambda n.$ (*any*, *replicate* n *False*)
defines $I' \equiv$ *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. $x \neq []$
shows $x \in$ *enc* (w, I')
⟨*proof*⟩

lemma *pred-case-conv*: x − *Suc 0* = (*case* x *of 0* ⇒ *0* | *Suc* m ⇒ m)
⟨*proof*⟩

lemma *in-pred-image-iff*: *0* ∉ X ⟹ ($x \in (\lambda x.\ x$ − *Suc 0*) ‘ X) = (*Suc* $x \in X$)
⟨*proof*⟩

lemma *map-project-Int-ENC*:
fixes X Z n
defines $z \equiv$ (*any*, *replicate* n *False*)
assumes *0* ∉ X $X \subseteq \{0$ ..< $n + 1\}$ $Z \subseteq$ *lists* ((*set* o σ Σ) ($n + 1$))
shows *SAMEQUOT* z (*map* π ‘ ($Z \cap$ *lang* ($n + 1$) (*ENC* ($n + 1$) X))) =
SAMEQUOT z (*map* π ‘ Z) ∩ *lang* n (*ENC* n ((λx. x − *1*) ‘ X))
⟨*proof*⟩

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*) ∩ *lang* n (*ENC* n *Y2*)
⟨*proof*⟩

lemma *map-project-ENC*:
fixes n
assumes $X \subseteq \{0$ ..< $n + 1\}$ $Z \subseteq$ *lists* ((*set* o σ Σ) ($n + 1$))
defines $z \equiv$ (*any*, *replicate* n *False*)
shows *SAMEQUOT* z (*map* π ‘ ($Z \cap$ *lang* ($n + 1$) (*ENC* ($n + 1$) X))) =
(*if 0* ∈ X
then SAMEQUOT z (*map* π ‘ ($Z \cap$ *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*⟩

lemma $lang_{M2L}$-*rexp-of′-rexp-of″*:
wf-formula n φ ⟹ *lang* n (*rexp-of′* n φ) = *lang* n (*rexp-of″* n φ)
⟨*proof*⟩

theorem $lang_{WS1S}$-*rexp-of′*: *wf-formula* n φ ⟹ $lang_{WS1S}$ n φ = *lang* n (*rexp-of′* n φ)
⟨*proof*⟩

theorem *lang$_{WS1S}$-rexp-of''*: *wf-formula n φ ⟹ lang$_{WS1S}$ n φ = lang n (rexp-of'' n φ)*
⟨*proof*⟩

end

14 Normalization of WS1S Formulas

fun *nNot* **where**
nNot (FNot φ) = φ
| *nNot (FAnd φ1 φ2) = FOr (nNot φ1) (nNot φ2)*
| *nNot (FOr φ1 φ2) = FAnd (nNot φ1) (nNot φ2)*
| *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 φ ⟷ (w, I) ⊨ FNot φ*
⟨*proof*⟩

lemma *FOV-nNot*[*simp*]: *FOV (nNot φ) = FOV (FNot φ)*
⟨*proof*⟩

lemma *SOV-nNot*[*simp*]: *SOV (nNot φ) = SOV (FNot φ)*
⟨*proof*⟩

lemma *pre-wf-formula-nNot*[*simp*]: *pre-wf-formula n (nNot φ) = pre-wf-formula n (FNot φ)*
⟨*proof*⟩

lemma *FOV-norm*[*simp*]: *FOV (norm φ) = FOV φ*
⟨*proof*⟩

lemma *SOV-norm*[*simp*]: *SOV (norm φ) = SOV φ*
⟨*proof*⟩

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

n φ
⟨*proof*⟩

lemma *satisfies-norm*[*simp*]: *wI* ⊨ *norm φ* ⟷ *wI* ⊨ *φ*
⟨*proof*⟩

lemma *lang$_{WS1S}$-norm*[*simp*]: *lang$_{WS1S}$ n* (*norm φ*) = *lang$_{WS1S}$ n φ*
⟨*proof*⟩

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
𝔇 = *embed.lderiv lookup* (*ε* Σ)
and *Co*𝔇 = *embed.lderiv-dual lookup* (*ε* Σ)
and *r*𝔇 = *embed.rderiv lookup* (*ε* Σ)
and *r*𝔇*-add* = *embed2.rderiv-and-add lookup* (*ε* Σ)
and 𝔘 = *embed2.samequot-exec lookup* (*ε* Σ) (*case-prod Singleton*)
⟨*proof*⟩

lemma *enum-not-empty*[*simp*]: *Enum.enum* ≠ [] (**is** *?enum* ≠ [])
⟨*proof*⟩

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*) = 𝔘 *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*
⟨*proof*⟩

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''*]

⟨*ML*⟩

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*
  ⟨*proof*⟩

definition *check-eqv* **where**
check-eqv n φ ψ ⟷ *wf-formula n* (*FOr* φ ψ) ∧
  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 ($\lambda w.$ *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 ψ ⟹ $\Phi.lang_{WS1S}$ *n* φ = $\Phi.lang_{WS1S}$ *n* ψ
  ⟨*proof*⟩

lemma *completeness*:
  assumes $\Phi.lang_{WS1S}$ *n* φ = $\Phi.lang_{WS1S}$ *n* ψ *wf-formula n* (*FOr* φ ψ)
  shows *slow.check-eqv n* φ ψ
  ⟨*proof*⟩

⟨*ML*⟩

global-interpretation *D*: *rexp-DA-no-post* σ Σ *wf-atom* Σ π *lookup* $\lambda x.$ *pnorm* (*inorm x*)

$\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r)$ *final alphabet.wf* $(wf\text{-}atom\ \Sigma)\ n\ lang\ \Sigma\ n\ n$
for $\Sigma :: 'a :: linorder\ list$ **and** $n :: nat$
defines
$test = rexp\text{-}DA.test\ (final :: 'a\ atom\ rexp \Rightarrow bool)$
and $step = rexp\text{-}DA.step\ (\sigma\ \Sigma)\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ id\ n$
and $closure = rexp\text{-}DA.closure\ (\sigma\ \Sigma)\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ final\ id\ n$
and $check\text{-}eqvRE = rexp\text{-}DA.check\text{-}eqv\ (\sigma\ \Sigma)\ (\lambda x.\ pnorm\ (inorm\ x))\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ final\ id\ n$
and $test\text{-}invariant = rexp\text{-}DA.test\text{-}invariant\ (final :: 'a\ atom\ rexp \Rightarrow bool) ::$
$(('a \times bool\ list)\ list \times \text{-})\ list \times \text{-} \Rightarrow bool$
and $step\text{-}invariant = rexp\text{-}DA.step\text{-}invariant\ (\sigma\ \Sigma)\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ id\ n$
and $closure\text{-}invariant = rexp\text{-}DA.closure\text{-}invariant\ (\sigma\ \Sigma)\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ final\ id\ n$
and $counterexampleRE = rexp\text{-}DA.counterexample\ (\sigma\ \Sigma)\ (\lambda x.\ pnorm\ (inorm\ x))\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ final\ id\ n$
and $reachable = rexp\text{-}DA.reachable\ (\sigma\ \Sigma)\ (\lambda x.\ pnorm\ (inorm\ x))\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ id\ n$
and $automaton = rexp\text{-}DA.automaton\ (\sigma\ \Sigma)\ (\lambda x.\ pnorm\ (inorm\ x))\ (\lambda a\ r.\ pnorm\ (\mathfrak{D}\ \Sigma\ a\ r))\ id\ n$
⟨*proof*⟩

definition *check-eqv* **where**
$check\text{-}eqv\ n\ \varphi\ \psi \longleftrightarrow wf\text{-}formula\ n\ (FOr\ \varphi\ \psi) \wedge$
$fast.check\text{-}eqvRE\ Enum.enum\ n\ (rexp\text{-}of''\ n\ (norm\ \varphi))\ (rexp\text{-}of''\ n\ (norm\ \psi))$

definition *counterexample* **where**
$counterexample\ n\ \varphi\ \psi =$
$map\text{-}option\ (\lambda w.\ dec\text{-}interp\ n\ (FOV\ (FOr\ \varphi\ \psi))\ (w\ @-\ sconst\ (any, replicate\ n\ False)))$
$(fast.counterexampleRE\ Enum.enum\ n\ (rexp\text{-}of''\ n\ (norm\ \varphi))\ (rexp\text{-}of''\ n\ (norm\ \psi)))$

lemma *soundness*: $fast.check\text{-}eqv\ n\ \varphi\ \psi \Longrightarrow \Phi.lang_{WS1S}\ n\ \varphi = \Phi.lang_{WS1S}\ n\ \psi$
⟨*proof*⟩

⟨*ML*⟩

global-interpretation *D*: $rexp\text{-}DA\text{-}no\text{-}post\ \sigma\ \Sigma\ wf\text{-}atom\ \Sigma\ \pi\ lookup$
$\lambda x.\ pnorm\text{-}dual\ (rexp\text{-}dual\text{-}of\ (inorm\ x))\ \lambda a\ r.\ pnorm\text{-}dual\ (Co\mathfrak{D}\ \Sigma\ a\ r)\ final\text{-}dual$
$alphabet.wf\text{-}dual\ (wf\text{-}atom\ \Sigma)\ n\ lang\text{-}dual\ \Sigma\ n\ n$
for $\Sigma :: 'a :: linorder\ list$ **and** $n :: nat$
defines
$test = rexp\text{-}DA.test\ (final\text{-}dual :: 'a\ atom\ rexp\text{-}dual \Rightarrow bool)$
and $step = rexp\text{-}DA.step\ (\sigma\ \Sigma)\ (\lambda a\ r.\ pnorm\text{-}dual\ (Co\mathfrak{D}\ \Sigma\ a\ r))\ id\ n$
and $closure = rexp\text{-}DA.closure\ (\sigma\ \Sigma)\ (\lambda a\ r.\ pnorm\text{-}dual\ (Co\mathfrak{D}\ \Sigma\ a\ r))\ final\text{-}dual\ id\ n$
and $check\text{-}eqvRE = rexp\text{-}DA.check\text{-}eqv\ (\sigma\ \Sigma)\ (\lambda x.\ pnorm\text{-}dual\ (rexp\text{-}dual\text{-}of\ (inorm\ x)))\ (\lambda a\ r.\ pnorm\text{-}dual\ (Co\mathfrak{D}\ \Sigma\ a\ r))\ final\text{-}dual\ id\ n$

and *test-invariant* = *rexp-DA.test-invariant* (*final-dual* :: *'a atom rexp-dual* ⇒ *bool*) ::
((*'a* × *bool list*) *list* × -) *list* × - ⇒ *bool*
and *step-invariant* = *rexp-DA.step-invariant* (σ Σ) (λ*a r*. *pnorm-dual* (*Co*𝔇 Σ *a r*)) *id n*
and *closure-invariant* = *rexp-DA.closure-invariant* (σ Σ) (λ*a r*. *pnorm-dual* (*Co*𝔇 Σ *a r*)) *final-dual id n*
and *counterexampleRE* = *rexp-DA.counterexample* (σ Σ) (λ*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*
⟨*proof*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

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.