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

Dmitriy Traytel and Tobias Nipkow

September 19, 2015

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 3
   1.1 Concatenation of Languages ............................... 3
   1.2 Iteration of Languages .................................. 4
   1.3 Left-Quotients of Languages ............................ 6
   1.4 Right-Quotients of Languages ............................ 7
   1.5 Two-Sided-Quotients of Languages ...................... 8
   1.6 Arden’s Lemma ............................................. 10
   1.7 Lists of Fixed Length .................................... 10

2 Π-Extended Regular Expressions 10
   2.1 Syntax of regular expressions ........................... 10
   2.2 ACI normalization ......................................... 11
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: \((A @@ B) @@ C = A @@ (B @@ C)\)
⟨proof⟩

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

lemma conc-UNION-distrib:
shows \((A @@ \bigcup I M) = \bigcup I (\% i. A @@ M i)\)
and \((\bigcup I M) @@ A = \bigcup I (\% i. M i @@ A)\)
⟨proof⟩

lemma hom-image-conc:
\(\bigwedge xs ys. f (xs @ ys) = f xs @ f ys\) \implies f \ (A @@ B)
\(= f \ A @@ f \ B\)
⟨proof⟩

lemma map-image-conc[simp]: \(map f \ \ (A @@ B) = \map f A @@ map f B\)
⟨proof⟩

lemma conc-subset-lists: \(A \subseteq \text{lists } S \implies B \subseteq \text{lists } S \implies A @@ B \subseteq \text{lists } S\)
⟨proof⟩

1.2 Iteration of Languages

lemma lang-pow-add: \(\hat{A} (n + m) = \hat{A} \ n \ @@ A \ m\)
⟨proof⟩

lemma lang-pow-simps: \(\hat{A} \ Suc n) = (A \ n \ @@ A\)
⟨proof⟩

lemma lang-pow-empty: \(\{\} \ n = (if n = 0 then \{\}\ else \{\})\)
⟨proof⟩

lemma lang-pow-empty-Suc[simp]: \(\{\}::\text{a lang} \ Suc n = \{\}
⟨proof⟩

lemma conc-pow-comm:
shows \((A @@ A \ n) = (A \ n @@ A\)
⟨proof⟩

lemma length-lang-pow-ub:
\(\\forall w : A. \ length w \leq k \implies w : \hat{A} \ n \implies length w \leq k \ast n\)
⟨proof⟩

lemma length-lang-pow-lb:
\(\\forall w : A. \ length w \geq k \implies w : \hat{A} \ n \implies length w \geq k \ast n\)
⟨proof⟩
lemma lang-pow-subset-lists: \( A \subseteq \text{lists } S \Rightarrow A \mathbin{^\ast} n \subseteq \text{lists } S \)
\(\langle\text{proof}\rangle\)

lemma star-subset-lists: \( A \subseteq \text{lists } S \Rightarrow \star A \subseteq \text{lists } S \)
\(\langle\text{proof}\rangle\)

lemma star-if-lang-pow[simp]: \( w : A \mathbin{^\ast} n \Rightarrow w : \star A \)
\(\langle\text{proof}\rangle\)

lemma Nil-in-star[iff]: \( \mathbb{L} : \star A \)
\(\langle\text{proof}\rangle\)

lemma star-if-lang[simp]: assumes \( w : A \) shows \( w : \star A \)
\(\langle\text{proof}\rangle\)

lemma append-in-starI[simp]:
assumes \( u : \star A \) and \( v : \star A \) shows \( u \@ v : \star A \)
\(\langle\text{proof}\rangle\)

lemma conc-star-star: \( \star A \@ \star A = \star A \)
\(\langle\text{proof}\rangle\)

lemma conc-star-comm:
shows \( A \@ \star A = \star A \@ A \)
\(\langle\text{proof}\rangle\)

lemma star-induct[consumes 1, case-names Nil append, induct set: star]:
assumes \( w : \star A \)
and \( P \mathbb{L} \)
and step: \(!!u. u : A \Rightarrow v : \star A \Rightarrow P v \Rightarrow P (u \@ v) \)
shows \( P w \)
\(\langle\text{proof}\rangle\)

lemma star-empty[simp]: \( \star \mathbb{L} = \{\mathbb{L}\} \)
\(\langle\text{proof}\rangle\)

lemma star-epsilon[simp]: \( \star \{\mathbb{L}\} = \{\mathbb{L}\} \)
\(\langle\text{proof}\rangle\)

lemma star-idemp[simp]: \( \star (\star A) = \star A \)
\(\langle\text{proof}\rangle\)

lemma star-unfold-left: \( \star A = A \@ \star A \cup \{\mathbb{L}\} \) (is \(?L = ?R\))
\(\langle\text{proof}\rangle\)

lemma concat-in-star: set \( ws \subseteq A \Rightarrow \text{concat } ws : \star A \)
\(\langle\text{proof}\rangle\)
**Lemma**: \texttt{in-star-iff-concat}:
\begin{align*}
  w : \star A = (\exists ws. \text{set } ws \subseteq A \& w = \text{concat } ws \& \not\in \text{set } ws) \\
  (\text{is } - = (\exists ws. \not\in \text{set } ws))
\end{align*}
\begin{proof}
\end{proof}

**Lemma**: \texttt{star-conv-concat}:
\[\star A = \{ \text{concat } ws \mid \text{set } ws \subseteq A \& \not\in \text{set } ws \} \]
\begin{proof}
\end{proof}

**Lemma**: \texttt{star-insert-eps} [simp]: \( \star (\text{insert } \not\in A) = \star A \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{star-decom}:
\begin{align*}
  \text{assumes } a & : x \in \star A x \neq \not\in & \\
  \text{shows } \exists a b. x = a @ b \& a \neq \not\in \& a \in A \& b \in \star A
\end{align*}
\begin{proof}
\end{proof}

**Lemma**: \texttt{Ball-starI}:
\[\forall a \in \text{set } \text{as}. [a] \in A \Rightarrow \text{as} \in \star A \]
\begin{proof}
\end{proof}

**Lemma**: \texttt{map-image-star} [simp]: \( \text{map } f \cdot \star A = \star (\text{map } f \cdot A) \)
\begin{proof}
\end{proof}

### 1.3 Left-Quotients of Languages

**Definition**: \texttt{lQuot} :: \( 'a \Rightarrow 'a \text{ lang} \Rightarrow 'a \text{ lang} \)
\begin{align*}
  \text{where } \ lQuot x A = \{ x \# x s \in A \}
\end{align*}

**Definition**: \texttt{lQuots} :: \( 'a \text{ list} \Rightarrow 'a \text{ lang} \Rightarrow 'a \text{ lang} \)
\begin{align*}
  \text{where } \ lQuots xs A = \{ x s \# y s \in A \}
\end{align*}

**Abbreviation**
\texttt{lQuotss} :: \( 'a \text{ list} \Rightarrow 'a \text{ lang set} \Rightarrow 'a \text{ lang} \)
\begin{align*}
  \text{where } \ lQuotss s As \equiv \bigcup (lQuots s \cdot As)
\end{align*}

**Lemma**: \texttt{lQuot-empty} [simp]: \( \text{lQuot } a \{ \} = \{ \} \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-epsilon} [simp]: \( \text{lQuot } a \{ \not\in \} = \{ \} \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-char} [simp]: \( \text{lQuot } a \{ [b] \} = (\text{if } a = b \text{ then } \{ \not\in \} \text{ else } \{ \}) \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-chars} [simp]: \( \text{lQuot } a \{ [b] \mid b. P b \} = (\text{if } P a \text{ then } \{ \not\in \} \text{ else } \{ \}) \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-union} [simp]: \( \text{lQuot } a (A \cup B) = \text{lQuot } a A \cup \text{lQuot } a B \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-inter} [simp]: \( \text{lQuot } a (A \cap B) = \text{lQuot } a A \cap \text{lQuot } a B \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-compl} [simp]: \( \text{lQuot } a (-A) = - \text{lQuot } a A \)
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-conc-subset}: \( \text{lQuot } a A @ @ B \subseteq \text{lQuot } a (A @ @ B) \) (is \( ?L \subseteq ?R \))
\begin{proof}
\end{proof}

**Lemma**: \texttt{lQuot-conc} [simp]: \( \text{lQuot } c (A @ @ B) = (\text{lQuot } c A) @ @ B \cup (\text{if } \not\in \in A \)
then \( \text{lQuot } c \ B \) else \{\}

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{lQuot-star [simp]}\): \(\text{lQuot } c \ (\text{star } A) = (\text{lQuot } c \ A) @@ \text{star } A\)

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{lQuot-diff [simp]}\): \(\text{lQuot } c \ (A - B) = \text{lQuot } c \ A - \text{lQuot } c \ B\)

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{lQuot-lists [simp]}\): \(c : S \implies \text{lQuot } c \ (\text{lists } S) = \text{lists } S\)

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{lQuots-simps [simp]}\):
- shows \(\text{lQuots } [] \ A = A\)
- and \(\text{lQuots } (c \# s) \ A = \text{lQuots } s \ (\text{lQuot } c \ A)\)
- and \(\text{lQuots } (s1 @ s2) \ A = \text{lQuots } s2 \ (\text{lQuots } s1 \ A)\)

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{lQuots-append [iff]}\): \(v \in \text{lQuots } w \ A \iff w @ v \in A\)

\(\langle \text{proof}\rangle\)

### 1.4 Right-Quotients of Languages

**Definition** \(\text{rQuot} : \:\text{'a lang} \Rightarrow \text{'a lang} :\)

where \(\text{rQuot } x \ A = \{ \text{xs} . \text{xs} @ [x] \in A \}\)

**Definition** \(\text{rQuots} : \:\text{'a list} \Rightarrow \text{'a lang} :\)

where \(\text{rQuots } x s \ A = \{ \text{ys} . \text{ys} @ \text{rev } x s \in A \}\)

**Abbreviation**

\(\text{rQuotss} : \:\text{'a list} \Rightarrow \text{'a lang set} \Rightarrow \text{'a lang} :\)

where \(\text{rQuotss } s \ As \equiv \bigcup \ (\text{rQuots } s \ ' \ As)\)

**Lemma** \(\text{rQuot-rev-lQuot} :\)
\(\text{rQuot } x \ A = \text{rev } \text{lQuot } x \ (\text{rev } \ ' \ A)\)

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{rQuots-rev-lQuots} :\)
\(\text{rQuots } x \ A = \text{rev } \text{lQuots } x \ (\text{rev } \ ' \ A)\)

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{rQuot-empty [simp]}\): \(\text{rQuot } a \ \{\} = \{\}\)

and \(\text{rQuot-epsilon [simp]}\): \(\text{rQuot } a \ \{[]\} = \{\}\)

and \(\text{rQuot-char [simp]}\): \(\text{rQuot } a \ \{[b]\} = (\text{if } a = b \ \text{then } \{[]\} \ \text{else } \{\}\}\)

and \(\text{rQuot-union [simp]}\): \(\text{rQuot } a \ (A \cup B) = \text{rQuot } a \ A \cup \text{rQuot } a \ B\)

and \(\text{rQuot-inter [simp]}\): \(\text{rQuot } a \ (A \cap B) = \text{rQuot } a \ A \cap \text{rQuot } a \ B\)

and \(\text{rQuot-compl [simp]}\): \(\text{rQuot } a \ (-A) = - \text{rQuot } a \ A\)

\(\langle \text{proof}\rangle\)

**Lemma** \(\text{lQuot-rQuot} :\)
\(\text{lQuot } a \ (\text{rQuot } b \ A) = \text{rQuot } b \ (\text{lQuot } a \ A)\)

7
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 {[e]} = {}

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 sl = length tl; length s2 = length t2] ⇒ biQuots (s1 @ s2) (t1 & t2) A = biQuots s2 t2 (biQuots s1 t1 A)
\langle proof \rangle

\textbf{lemma biQuots-append[iff]}: \( v \in \text{biQuots} \ u \ w \ A \iff u \ @ \ v \ @ \ \text{rev} \ w \in A \)
\langle proof \rangle

\subsection{Arden’s Lemma}

\textbf{lemma arden-helper}:
\begin{itemize}
  \item \textbf{assumes} \( eq: X = A \ @ @ \ X \cup B \)
  \item \textbf{shows} \( X = (A \ "\ Suc \ n) \ @ @ X \cup (\bigcup m \leq n. (A \ "\ m) \ @ @ B) \)
\end{itemize}
\langle proof \rangle

\textbf{lemma Arden}:
\begin{itemize}
  \item \textbf{assumes} \[ \[] \notin A
  \item \textbf{shows} \( X = A \ @ @ X \cup B \iff X = \text{star} A \ @ @ B \)
\end{itemize}
\langle proof \rangle

\textbf{lemma reversed-arden-helper}:
\begin{itemize}
  \item \textbf{assumes} \( eq: X = X \ @ @ A \cup B \)
  \item \textbf{shows} \( X = X \ @ @ (A \ "\ Suc \ n) \cup (\bigcup m \leq n. B \ @ @ (A \ "\ m)) \)
\end{itemize}
\langle proof \rangle

\textbf{theorem reversed-Arden}:
\begin{itemize}
  \item \textbf{assumes} \( \text{nemp: } \[ \notin A \)
  \item \textbf{shows} \( X = X \ @ @ A \cup B \iff X = B \ @ @ \text{star} A \)
\end{itemize}
\langle proof \rangle

\subsection{Lists of Fixed Length}

\textbf{abbreviation listsN} where \( \text{listsN} \ n \ S \equiv \{ x s. \ x s \in \text{lists} \ S \land \text{length} \ x s = n \} \)

\textbf{lemma tl-listsN}:
\( A \subseteq \text{listsN} \ (n+1) \ S \implies \text{tl} \ A \subseteq \text{listsN} \ n \ S \)
\langle proof \rangle

\textbf{lemma map-tl-listsN}:
\( A \subseteq \text{lists} \ (\text{listsN} \ (n+1) \ S) \implies \text{map} \ \text{tl} \ A \subseteq \text{lists} \ (\text{listsN} \ n \ S) \)
\langle proof \rangle

\section{Π-Extended Regular Expressions}

\subsection{Syntax of regular expressions}

\textbf{datatype} \('a\ rexp =
  \begin{align*}
  & \text{Zero} | \\
  & \text{Full} | \\
  & \text{One} | \\
  & \text{Atom} \ 'a | \\
  & \text{Plus} \ ('a \ rexp) \ ('a \ rexp) |
\end{align*}
\( Times \ (\texttt{a rexp}) \ (\texttt{a rexp}) \ |
\)  
\( Star \ (\texttt{a rexp}) \ |
\)  
\( Not \ (\texttt{a rexp}) \ |
\)  
\( Inter \ (\texttt{a rexp}) \ (\texttt{a rexp}) \ |
\)  
\( Pr \ (\texttt{a rexp}) \)

**derive linorder rexp**

Lifting constructors to lists

\[
\text{fun } \text{rexp-of-list } \text{where} \\
\text{rexp-of-list OPERATION N } [] = N \\
\text{| rexp-of-list OPERATION N } [x] = x \\
\text{| rexp-of-list OPERATION N } (x \# xs) = \text{OPERATION } x \ (\text{rexp-of-list OPERATION } N \ xs)
\]

**abbreviation** PLUS \( \equiv \) \text{rexp-of-list Plus Zero}

**abbreviation** TIMES \( \equiv \) \text{rexp-of-list Times One}

**abbreviation** INTERSECT \( \equiv \) \text{rexp-of-list Inter Full}

**lemma** list-singleton-induct [case-names nil single cons]:
\[\text{assumes } \text{nil}: P [] \]
\[\text{assumes } \text{single}: \bigwedge x. P [x] \]
\[\text{assumes } \text{cons}: \bigwedge x y xs. P (y \# xs) \implies P (x \# (y \# xs)) \]
\[\text{shows } P xs \]

**proof\]

\[
\text{2.2 ACI normalization}
\]

\[
\text{fun } \text{toplevel-summands } \text{where} \\
\text{toplevel-summands } (\text{Plus } r s) = \text{toplevel-summands } r \cup \text{toplevel-summands } s \\
\text{| toplevel-summands } r = \{ r \}
\]

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

**lemma** toplevel-summands-nonempty[simp]:
\[\text{toplevel-summands } r \neq \{ \} \]

**proof\]

**lemma** toplevel-summands-finite[simp]:
\[\text{finite } \text{(toplevel-summands } r) \]

**proof\]

**primrec** ACI-norm :: (\(\text{a::linorder}\) rexp \( \Rightarrow \) \text{a rexp} (\(\text{\textless \textgreater}\)) where
\[<\text{Zero}> = \text{Zero} \]
\[<\text{Full}> = \text{Full} \]
\[<\text{One}> = \text{One} \]
\[<\text{Atom } a> = \text{Atom } a \]
\[<\text{Plus } r s> = \text{flatten PLUS } (\text{toplevel-summands } (\text{Plus } r s)) \]
\[<\text{Times } r s> = \text{Times } r s \]
\[<\text{Star } r> = \text{Star } r \]
lemma Plus-toplevel-summands:
\[\text{Plus } r \ s \in \text{toplevel-summands } t \implies \text{False}\]
(\text{proof})

lemma toplevel-summands-not-Plus[simp]:
\[(\forall r \ s. \ x \neq \text{Plus } r \ s) \implies \text{toplevel-summands } x = \{ x \}\]
(\text{proof})

lemma toplevel-summands-PLUS-strong:
\[\emptyset \neq \text{list-all } (\lambda x. \neg (\exists r \ s. \ x = \text{Plus } r \ s)) \text{ } \text{xs} \implies \text{toplevel-summands } (\text{PLUS } \text{xs}) = \text{set } \text{xs}\]
(\text{proof})

lemma toplevel-summands-flatten:
\[\{ X \neq \{}; \text{finite } X; \forall x \in X. \neg (\exists r \ s. \ x = \text{Plus } r \ s)\] \implies \text{toplevel-summands } (\text{flatten PLUS } X) = X
(\text{proof})

lemma ACI-norm-Plus:
\[<r> = \text{Plus } s \ t \implies \exists s \ t. \ r = \text{Plus } s \ t\]
(\text{proof})

lemma toplevel-summands-flatten-ACI-norm-image:
\[\text{toplevel-summands } (\text{flatten PLUS } (\text{ACI-norm } ' \text{toplevel-summands } r)) = \text{ACI-norm } ' \text{toplevel-summands } r\]
(\text{proof})

lemma toplevel-summands-flatten-ACI-norm-image-Union:
\[\text{toplevel-summands } (\text{flatten PLUS } (\text{ACI-norm } ' \text{toplevel-summands } r \cup \text{ACI-norm } ' \text{toplevel-summands } s)) = \text{ACI-norm } ' \text{toplevel-summands } r \cup \text{ACI-norm } ' \text{toplevel-summands } s\]
(\text{proof})

lemma toplevel-summands-ACI-norm:
\[\text{toplevel-summands } <r> = \text{ACI-norm } ' \text{toplevel-summands } r\]
(\text{proof})

lemma ACI-norm-flatten:
\[<r> = \text{flatten PLUS } (\text{ACI-norm } ' \text{toplevel-summands } r)\]
(\text{proof})

theorem ACI-norm-idem[simp]:
\[<<r>> = <r>\]
(\text{proof})
fun ACI-nPlus :: 'a::linorder ⇒ exp ⇒ exp ⇒ exp
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:
\[ [xs1 \neq []; xs2 \neq []; \forall x \in set (xs1 @ xs2). \neg(\exists r s. x = Plus r s); \text{sorted } xs2; \text{distinct } xs2] \implies \text{ACI-nPlus (PLUS} \ \text{xs1}) (\text{PLUS} \ \text{xs2}) = \text{flatten PLUS (set} \ \text{xs1} @ \ \text{xs2}) \]

**proof**

**lemma** ACI-nPlus-flatten-PLUS:
\[ [X1 \neq {}; X2 \neq {}; \text{finite } X1; \text{finite } X2; \forall x \in X1 \cup X2. \neg(\exists r s. x = Plus r s)] \implies \text{ACI-nPlus (flatten PLUS} \ \text{X1}) (\text{flatten PLUS} \ \text{X2}) = \text{flatten PLUS (X1} \cup \text{X2)} \]

**proof**

**lemma** ACI-nPlus-ACI-norm[simp]: ACI-nPlus \( \langle r \rangle \langle s \rangle = \langle \text{Plus } r s \rangle \)

**proof**

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

### 2.3 Finality

**primrec** final :: 'a repn \( \Rightarrow \) bool

**where**
- final Zero = False
- final Full = True
- final One = True
- final (Atom -) = False
- final (Plus r s) = (final r \( \lor \) final s)
- final (Times r s) = (final r \( \land \) final s)
- final (Star -) = True
- final (Not r) = (\( \neg \) final r)
- final (Inter r1 r2) = (final r1 \( \land \) final r2)
- final (Pr r) = final r

**lemma** toplevel-summands-final:
\[ \text{final } s = (\exists r \in \text{toplevel-summands } s. \text{final } r) \]

**proof**

**lemma** final-PLUS:
\[ \text{final } (\text{PLUS} \ \text{xs}) = (\exists r \in \text{set } \ \text{xs. final } r) \]

**proof**

**theorem** ACI-norm-final[simp]:
\[ \text{final } \langle r \rangle = \text{final } r \]

**proof**
2.4 Wellformedness w.r.t. an alphabet

locale alphabet = 
fixes \Sigma :: 'a set (\Sigma :: 'a set) 
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 \land wf n s) |
  wf n (Times r s) = (wf n r \land 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 \land 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 \Sigma n) \land wf-word n ws)

lemma wf-word-snoc[simp]: wf-word n (ws @ [w]) = ((w \in \Sigma n) \land wf-word n ws)
  ⟨proof⟩

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

lemma wf-word: wf-word n w = (w \in lists (\Sigma 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 \Rightarrow wf n (flatten PLUS X) = (\forall r \in X. wf n r)
  ⟨proof⟩
**theorem** ACI-norm-wf [simp]:
\[ \text{wf} \ n \ \langle r \rangle = \text{wf} \ n \ r \]
(\langle \text{proof} \rangle)

**lemma** wf-INTERSECT [simp]:
\[ \text{wf} \ n \ (\text{INTERSECT} \ \text{xs}) = (\forall r \in \text{set} \ \text{xs}. \ \text{wf} \ n \ r) \]
(\langle \text{proof} \rangle)

**lemma** wf-flatten-INTERSECT [simp]:
\[ \text{finite} \ X \implies \text{wf} \ n \ (\text{flatten INTERSECT} \ X) = (\forall r \in X. \ \text{wf} \ n \ r) \]
(\langle \text{proof} \rangle)

end

### 2.5 Language

locale project =
  alphabet \( \Sigma \) wf-atom for \( \Sigma :: \text{nat} \) \text{ and } wf-atom :: \text{nat} \Rightarrow 'a :: \text{linorder} \Rightarrow \text{bool} +
  fixes \( \text{project} :: 'a \Rightarrow 'a \)
  and \( \text{lookup} :: 'b \Rightarrow 'a \Rightarrow \text{bool} \)
  assumes \( \text{project} : \bigwedge a. a \in \Sigma (\text{Suc} \ n) \Rightarrow \text{project} a \in \Sigma n \)
begin

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

**lemma** wf-word-map-project [simp]: \( \text{wf-word} (\text{Suc} \ n) \ \text{ws} \Rightarrow \text{wf-word} \ n \ (\text{map} \ \text{project} \ \text{ws}) \)
(\langle \text{proof} \rangle)

**lemma** wf-lang-uf-word: \( \text{wf} \ n \ r \Rightarrow \forall w \in \text{lang} \ n \ r. \ \text{wf-word} \ n \ w \)
(\langle \text{proof} \rangle)

**lemma** lang-subset-lists: \( \text{wf} \ n \ r \Rightarrow \text{lang} \ n \ r \subseteq \text{lists} (\Sigma n) \)
(\langle \text{proof} \rangle)

**lemma** toplevel-summands-lang:
\( r \in \text{toplevel-summands} \ \text{s} \Rightarrow \text{lang} \ n \ r \subseteq \text{lang} \ n \ s \)
(\langle \text{proof} \rangle)

16


**lemma** toplevel-summands-lang-UN:
\[
\text{lang } n \text{ } s = (\bigcup r \in \text{toplevel-summands } s. \text{ lang } n \text{ } r)
\]
⟨proof⟩

**lemma** toplevel-summands-in-lang:
\[
w \in \text{lang } n \text{ } s = (\exists r \in \text{toplevel-summands } s. \text{ w } \in \text{lang } n \text{ } r)
\]
⟨proof⟩

**lemma** lang-PLUS[simp]:
\[
\text{lang } n \text{ } (\text{PLUS } xs) = (\bigcup r \in \text{set } xs. \text{ lang } n \text{ } r)
\]
⟨proof⟩

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

**lemma** lang-flatten-PLUS:
\[
\text{finite } X \implies \text{lang } n \text{ } (\text{flatten PLUS } X) = (\bigcup r \in X. \text{ lang } n \text{ } r)
\]
⟨proof⟩

**theorem** ACI-norm-lang[simp]:
\[
\text{lang } n \text{ } <r> = \text{lang } n \text{ } r
\]
⟨proof⟩

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

**lemma** in-lang-INTERSECT:
\[
\text{wf-word } n \text{ } w \implies w \in \text{lang } n \text{ } (\text{INTERSECT } xs) = (\forall r \in \text{set } xs. \text{ w } \in \text{lang } n \text{ } r)
\]
⟨proof⟩

**lemma** lang-INTERSECT:
\[
\text{lang } n \text{ } (\text{INTERSECT } xs) = (\text{if } xs = [] \text{ then lists } (\Sigma \text{ } n) \text{ else } \bigcap r \in \text{set } xs. \text{ lang } n \text{ } r)
\]
⟨proof⟩

**lemma** lang-flatten-INTERSECT[simp]:
\[
\text{assumes } \text{finite } X \text{ } X \neq {} \forall r \in X. \text{ wf } n \text{ } r
\]
\[
\text{shows } w \in \text{lang } n \text{ } (\text{flatten INTERSECT } X) = (\forall r \in X. \text{ w } \in \text{lang } n \text{ } r) \text{ (is } ?L = ?R)
\]
⟨proof⟩

end

3 Derivatives of Π-Extended Regular Expressions
locale embed = project \Sigma \ wf-atom project lookup

for \Sigma :: 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: \(\forall a. \ a \in \Sigma \ n \Rightarrow b \in \text{set} (\text{embed} a) = (b \in \Sigma \ (\text{Suc} \ n) \land \text{project} \ b = a)\)

begin

3.1 Syntactic Derivatives

primrec lderiv :: 'a \Rightarrow \rexp \Rightarrow \rexp \ where
lderv - 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 (lderv a r) (lderv a s)
| lderiv a (Times r s) = 
  (let r's = Times (lderv a r) s
   in if final r then Plus r's (lderv a s) else r's)
| lderiv a (Star r) = Times (lderv a r) (Star r)
| lderiv a (Not r) = Not (lderv a r)
| lderiv a (Inter r s) = Inter (lderv a r) (lderv a s)
| lderiv a (Pr r) = Pr (PLUS (map (\a'. lderiv a' r) (embed a)))

primrec lderivs where
ldervs [] r = r
| lderivs (w#ws) r = lderivs ws (lderv w r)

3.2 Finiteness of ACI-Equivalent Derivatives

lemma toplevel-summands-lderiv:
toplevel-summands (lderiv as r) = (\bigcup \ s \in \text{toplevel-summands} r. toplevel-summands (lderiv as s))
(proof)

lemma lderivs-Zero[simp]: lderiv xs Zero = Zero
(proof)

lemma lderivs-Full[simp]: lderiv xs Full = Full
(proof)

lemma lderivs-One: lderiv xs One \in \{Zero, One\}
(proof)

lemma lderivs-Atom: lderiv xs (Atom as) \in \{Zero, One, Atom as\}
(proof)

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

18
lemma lderivs-PLUS: lderiv xs (PLUS ys) = PLUS (map (lderiv xs) ys)
⟨proof⟩

lemma toplevel-summands-lderivs-Times: toplevel-summands (lderiv xs (Times r s)) ⊆
{Times (lderiv xs r) s} ∪
{r'. ∃ys zs. r' ∈ toplevel-summands (lderiv ys s) ∧ ys ≠ [] ∧ zs @ ys = xs}
⟨proof⟩

lemma toplevel-summands-lderivs-Star-nonempty:
xs ≠ [] ⇒ toplevel-summands (lderiv xs (Star r)) ⊆
{Times (lderiv ys r) (Star r) | ys. ∃zs. ys ≠ [] ∧ zs @ ys = xs}
⟨proof⟩

lemma toplevel-summands-lderivs-Pr:
xs ≠ [] ⇒ toplevel-summands (Pr r) = Pr s
⟨proof⟩

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

lemma lderiv-toplevel-summands-Zero:
[lderiv xs (Pr r) = Pr s; toplevel-summands r = {Zero}] ⇒ toplevel-summands s = {Zero}
⟨proof⟩

lemma toplevel-summands-lderivs-Pr:
[xs ≠ []; lderiv xs (Pr r) = Pr s] ⇒
toplevel-summands s ⊆ {Zero} ∨ toplevel-summands s ⊆ (∪ xs. toplevel-summands (lderiv xs r))
⟨proof⟩

lemma lderivs-Pr:
{lderiv xs (Pr r) | xs. True} ⊆
{Pr s | s. toplevel-summands s ⊆ {Zero} ∨ toplevel-summands s ⊆ (∪ xs. toplevel-summands (lderiv 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 ν {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 lderiv-Not: lderivs xs (Not r) = Not (lderivs xs r)
(proof)

lemma lderiv-Inter: lderivs xs (Inter r s) = Inter (lderivs xs r) (liderivs 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 (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

20
3.4 Deriving preserves ACI-equivalence

**Lemma** $ACI$-$norm$-$PLUS$:
\[
\text{list-all2} \ (\lambda r \ s. \ [r] = [s]) \ xs \ ys \implies [PLUS] xs = [PLUS] ys
\]
(\textit{proof})

**Lemma** $toplevel$-$summands$-$ACI$-$norm$-$lderiv$:
\[
\left(\bigcup a \in toplevel$-$summands \ r \ . \ toplevel$-$summands \ [lderiv \ as \ a]\right) = toplevel$-$summands \ [lderiv \ as \ r]
\]
(\textit{proof})

**Theorem** $ACI$-$norm$-$lderiv$:
\[
[lderiv \ as \ [r]] = [lderiv \ as \ r]
\]
(\textit{proof})

**Corollary** $lderiv$-$preserves$: $[r] = [s] \Rightarrow [lderiv \ as \ r] = [lderiv \ as \ s]
(\textit{proof})

**Lemma** $lderivs$-$snoc$[simp]: $lderiv \ (ws \ @ \ [w]) \ r = (lderiv \ w \ (lderiv \ ws \ r))$
(\textit{proof})

**Theorem** $ACI$-$norm$-$lderives$:
\[
[lderiv \ ws \ [r]] = [lderiv \ ws \ r]
\]
(\textit{proof})

**Lemma** $lderivs$-$alt$: $[lderiv \ ws \ [r]] = fold \ (\lambda a \ r. \ [lderiv \ a \ r]) \ w \ [r]
(\textit{proof})

**Lemma** $finite$-$fold$-$lderiv$: $finite \ \{fold \ (\lambda a \ r. \ [lderiv \ a \ r]) \ w \ [s] \ | w. \ True\}$
(\textit{proof})

end

4 Some Useful Regular Operators

**primrec** $REV :: \ 'a \ rexp \Rightarrow \ 'a \ rexp$ \textbf{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 \implies 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 (λa'. rderiv a' r) (embed a)))

**Primrec** rderivs where  
rderivs [] r = r  
rderivs (w#ws) r = rderivs ws (rderiv w r)  

**Lemma** rderivs-snoc: rderivs (ws @ [w]) r = rderiv w (rderivs ws r)  
(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 → wf n (rderiv w r)
⟨proof⟩

lemma wf-rderivws[simp]: wf n r → wf n (rderivs ws r)
⟨proof⟩

lemma lang-rderiv: [wf n r; as ∈ Σ n] → lang n (rderiv as r) = rQuot as (lang n r)
⟨proof⟩

lemma lang-rderivws: [wf n r; wf-word n w] → 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) ↔ rev w ∈ 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 «r»» = «rderiv as r»
⟨proof⟩

lemma ACI-norm-rderivws: «rderivs «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))

23
lemma lang-rderiv-lderivs[simp]: \[ \text{wf } n \ r; \text{wf-word } n \ w; \ a \in \Sigma_n \] \implies
\[ \text{lang } n \ (\text{rderiv } a \ (\text{lderiv } w \ r)) = \text{lang } n \ (\text{lderiv } w \ (\text{rderiv } a \ r)) \]

lemma lang-lderiv-rderivs[simp]: \[ \text{wf } n \ r; \text{wf-word } n \ w; \ a \in \Sigma_n \] \implies
\[ \text{lang } n \ (\text{lderiv } a \ (\text{rderiv } w \ r)) = \text{lang } n \ (\text{rderiv } w \ (\text{lderiv } a \ r)) \]

definition biderivs w1 w2 = \text{rderivs } w2 \circ \text{lderivs } w1

lemma lang-biderivs: \[ \text{wf } n \ r; \text{wf-word } n \ w1; \text{wf-word } n \ w2 \] \implies
\[ \text{lang } n \ (\text{biderivs } w1 \ w2 \ r) = \text{biQuots } w1 \ w2 \ (\text{lang } n \ r) \]

lemma wf-biderivs: \[ \text{wf } n \ r \] \implies \[ \text{wf } n \ (\text{biderivs } w1 \ w2 \ r) \]

corollary biderivs-final:
assumes \[ \text{wf } n \ r \text{ wf-word } n \ w1 \text{ wf-word } n \ w2 \]
shows \[ \text{final } (\text{biderivs } w1 \ w2 \ r) \iff w1 \ @ \ \text{rev } w2 \in \text{lang } n \ r \]

lemma ACI-norm-biderivs:\[ (\text{biderivs } w1 \ w2 <r>) = (\text{biderivs } w1 \ w2 <r>) \] 

lemma finite\{\text{biderivs } w1 \ w2 <r> | w1 w2 . True\}

end

4.1 Quotioning by the same letter

definition fin-cut-same x xs = \text{take } (\text{LEAST } n, \text{drop } n \ xs = \text{replicate } (\text{length } xs - n) \ x) \ xs

lemma fin-cut-same-Nil[simp]: fin-cut-same [] = []

lemma Least-fin-cut-same: (\text{LEAST } n, \text{drop } n \ xs = \text{replicate } (\text{length } xs - n) \ y) = 
\[ \text{length } xs - \text{length } (\text{takeWhile } (\lambda x. x = y) \ (\text{rev } xs)) \]
(is Least ?P = ?min)

lemma takeWhile-takes-all: length xs = m \implies m \leq \text{length } (\text{takeWhile } P \ xs) \iff 
\text{Ball } (\text{set } xs) \ P

24
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 # 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
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) = UNION {≪rderiv (replicate m as) r≫ | m. True}
  (lang n)
  ⟨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⟩

eval

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

\[ \text{Cycle } L = \{ u \triangleleft w \mid u, w \triangleleft u \in L \} \]

**context** embed

**begin**

**context**

**fixes** \( n :: \text{nat} \)

**begin**

**definition** SUFFIX :: 'b rexp ⇒ 'b rexp where

\[ \text{SUFFIX } r = \text{flatten PLUS } \{ \text{lderivs } w \ r \mid \text{w, wf-word } n \ w \} \]

**lemma** finite-lderivs-uf: finite \( \{ \text{lderivs } w \ r \mid \text{w, wf-word } n \ w \} \)

⟨proof⟩

**definition** PREFIX :: 'b rexp ⇒ 'b rexp where

\[ \text{PREFIX } r = \text{REV} \ (\text{SUFFIX} \ (\text{REV } r)) \]

**lemma** wf-SUFFIX[simp]: \( \text{wf } n \ r \implies \text{wf } n \ (\text{SUFFIX } r) \)

⟨proof⟩

**lemma** lang-SUFFIX[simp]: \( \text{wf } n \ r \implies \text{lang } n \ (\text{SUFFIX } r) = \text{Suffix } (\text{lang } n \ r) \)

⟨proof⟩

**lemma** wf-PREFIX[simp]: \( \text{wf } n \ r \implies \text{wf } n \ (\text{PREFIX } r) \)

⟨proof⟩

**lemma** lang-PREFIX[simp]: \( \text{wf } n \ r \implies \text{lang } n \ (\text{PREFIX } r) = \text{Prefix } (\text{lang } n \ r) \)

⟨proof⟩

**end**

**lemma** take-drop-CycleI[intro!]: \( x \in L \implies \text{drop } i \ x \triangleleft \text{take } i \ x \in \text{Cycle } L \)

⟨proof⟩

**lemma** take-drop-CycleI [intro!]: \( \text{drop } i \ x \triangleleft \text{take } i \ x \in L \implies x \in \text{Cycle } L \)

⟨proof⟩

**end**

5  **Π-Extended Dual Regular Expressions**

5.1  **Syntax of regular expressions**

**datatype** 'a rexp-dual =

\[ \text{CoZero} \ (\text{co: bool}) \mid \text{CoOne} \ (\text{co: bool}) \mid \]

28
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 op ∨ else op ∧)
abbreviation set-binop-dual b ≡ (if b then op ∪ else op ∩)

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 {} |
\text{lang-dual } n (\text{CoAtom } b a) = \text{set-unop-dual } n b \{[x] \mid x. \text{lookup } a \ x \land x \in \Sigma \ n\} \ |
\text{lang-dual } n (\text{CoPlus } b r s) = \text{set-binop-dual } b (\text{lang-dual } n r) (\text{lang-dual } n s) \ |
\text{lang-dual } n (\text{CoTimes } b r s) = \text{set-unop-dual } n b
\left(set-unop-dual n b \ (\text{lang-dual } n r) \ \#\# \ \text{set-unop-dual } n b \ (\text{lang-dual } n s))\right) \ |
\text{lang-dual } n (\text{CoStar } b r) = \text{set-unop-dual } n b \ (\text{star } (\text{set-unop-dual } n b \ (\text{lang-dual } n r))) \ |
\text{lang-dual } n (\text{CoPr } b r) = \text{set-unop-dual } n b \ (\text{map project } ' \ (\text{set-unop-dual } (n + 1) b \ (\text{lang-dual } (n + 1) r)))

\textbf{lemma} \text{wf-dual-lang-dual-wf-word}: \text{wf-dual } n r \implies \forall w \in \text{lang-dual } n r. \text{wf-word } n w
\langle \text{proof} \rangle

\textbf{lemma} \text{lang-dual-subset-lists}: \text{wf-dual } n r \implies \text{lang-dual } n r \subseteq \text{lists } (\Sigma \ n)
\langle \text{proof} \rangle

\textbf{lemma} \text{lang-dual-final-dual}: \text{final-dual } r = ([] \in \text{lang-dual } n r)
\langle \text{proof} \rangle

\textbf{lemma} \text{lang-dual-PLUS-dual}[\text{simp}];
\text{lang-dual } n (\text{CoPLUS-dual } \text{True } xs) = (\bigcup r \in \text{set } xs. \text{lang-dual } n r)
\langle \text{proof} \rangle

\textbf{lemma} \text{lang-dual-CoPLUS-dual}[\text{simp}];
\text{lang-dual } n (\text{CoPLUS-dual } \text{False } xs) = (\text{if } \text{xs} = [] \text{ then } \text{lists } (\Sigma \ n) \text{ else } \bigcap r \in \text{set } xs. \text{lang-dual } n r)
\langle \text{proof} \rangle

\textbf{end}

\textbf{context} embed
\begin{primrec}
\textbf{iderv-dual} :: 'a ⇒ 'b rexp-dual ⇒ 'b rexp-dual
\textbf{iderv-dual} - (\text{CoZero } b) = (\text{CoZero } b)
| \textbf{iderv-dual} - (\text{CoOne } b) = (\text{CoZero } b)
| \textbf{iderv-dual} a (\text{CoAtom } b c) = (\text{if } \text{lookup } c \ a \ \text{then } \text{CoOne } b \ \text{else } \text{CoZero } b)
| \textbf{iderv-dual} a (\text{CoPlus } b r s) = \text{CoPlus } b (\text{iderv-dual } a r) (\text{iderv-dual } a s)
| \textbf{iderv-dual} a (\text{CoTimes } b r s) =
\langle \text{let } r's = \text{CoTimes } b (\text{iderv-dual } a r) s
\langle \text{in if bool-unop-dual } b \ (\text{final-dual } r) \ \text{then } \text{CoPlus } b r's \ (\text{iderv-dual } a s) \ \text{else } r's\rangle
| \text{iderv-dual} a (\text{CoStar } b r) = \text{CoTimes } b (\text{iderv-dual } a r) (\text{CoStar } b r)
| \text{iderv-dual} a (\text{CoPr } b r) = \text{CoPr } b (\text{CoPLUS-dual } b \ (\text{map } (\lambda a'. \ \text{iderv-dual } a' r) \ (\text{embed } a)))
\rangle

\textbf{primrec} \text{idervs-dual}
\langle \text{where}
\text{idervs-dual} [] r = r
| \text{idervs-dual } (w#ws) r = \text{idervs-dual } ws (\text{iderv-dual } w r)
\rangle
\end{primrec}
lemma \( \text{wf-dual-lderiv-dual}[\text{simp}]: \) \( \text{wf-dual} n r \Rightarrow \text{wf-dual} n (\text{lderiv-dual} w r) \)

(\text{proof})

lemma \( \text{wf-dual-lderivs-dual}[\text{simp}]: \) \( \text{wf-dual} n r \Rightarrow \text{wf-dual} n (\text{lderivs-dual} ws r) \)

(\text{proof})

lemma \( \text{lang-dual-lderiv-dual}[\text{simp}]: \) \( \text{wf-dual} n r \Rightarrow \text{lang-dual} n (\text{lderiv-dual} w r) = \text{lQuot} w (\text{lang-dual} n r) \)

(\text{proof})

corollary \( \text{lderivs-dual-final-dual} \):
  assumes \( \text{wf-dual} n r \) \( \text{wf-word} n ws \)
  shows \( \text{final-dual} (\text{lderivs-dual} ws r) \leftarrow\rightarrow ws \in \text{lang-dual} n r \)

(\text{proof})

fun \( \text{pnCoPlus} :: \text{bool} \Rightarrow 'a::\text{linorder} \\text{rexp-dual} \Rightarrow 'a \\text{rexp-dual} \Rightarrow 'a \\text{rexp-dual} \)

where

\[
\begin{align*}
    \text{pnCoPlus} b1 \; (\text{CoZero} b2) \; r & = (\text{if } b1 = b2 \text{ then } r \text{ else } \text{CoZero} b2) \\
    \text{pnCoPlus} b1 \; r \; (\text{CoZero} b2) & = (\text{if } b1 = b2 \text{ then } r \text{ else } \text{CoZero} b2) \\
    \text{pnCoPlus} b1 \; (\text{CoPlus} b2 \; r \; s) \; t & = \\
        (\text{if } b1 = b2 \text{ then } (\text{pnCoPlus} b2 \; r \; s) \text{ else } \text{CoPlus} b1 \; (\text{CoPlus} b2 \; r \; s)) \\
    \text{pnCoPlus} b1 \; r \; (\text{CoPlus} b2 \; s) & = \\
        (\text{if } r = s \text{ then } r \text{ else if } r \leq s \text{ then } \text{CoPlus} b1 \; r \; s \text{ else } \text{CoPlus} b1 \; s \; r) \\
    \text{pnCoPlus} b \; r \; s & = \\
        (\text{if } r = s \text{ then } r \text{ else if } r \leq s \text{ then } \text{CoPlus} b \; r \; s \text{ else } \text{CoPlus} b \; s \; r)
\end{align*}
\]

lemma (in alphabet) \( \text{wf-dual-pnCoPlus}[\text{simp}]: \) \( \text{wf-dual} n r; \text{wf-dual} n s \Rightarrow \text{wf-dual} n (\text{pnCoPlus} b \; r \; s) \)

(\text{proof})

lemma (in project) \( \text{lang-dual-pnCoPlus}[\text{simp}]: \) \( \text{wf-dual} n r; \text{wf-dual} n s \Rightarrow \text{lang-dual} n (\text{pnCoPlus} b \; r \; s) = \text{lang-dual} n (\text{CoPlus} b \; r \; s) \)

(\text{proof})

fun \( \text{pnCoTimes} :: \text{bool} \Rightarrow 'a::\text{linorder} \\text{rexp-dual} \Rightarrow 'a \\text{rexp-dual} \Rightarrow 'a \\text{rexp-dual} \)

where
\textit{pnCoTimes}\ b1 \ (CoZero\ b2)\ r = (if\ b1 = b2\ then\ CoZero\ b1\ else\ CoTimes\ b1\ (CoZero\ b2)) r\\ | \textit{pnCoTimes}\ b1 \ (CoOne\ b2)\ r = (if\ b1 = b2\ then\ r\ else\ CoTimes\ b1\ (CoOne\ b2)) r\\ | \textit{pnCoTimes}\ b1 \ (CoPlus\ b2\ r\ s)\ t = (if\ b1 = b2\ then\ \textit{pnCoPlus}\ b2\ \textit{pnCoTimes}\ b2\ r\ t\ (\textit{pnCoTimes}\ b2\ s\ t)\ else\ CoTimes\ b1\ (CoPlus\ b2\ r\ s)\ t)\\ | \textit{pnCoTimes}\ b\ r\ s = CoTimes\ b\ r\ s\\

\textbf{lemma (in alphabet)} w\mathit{f-dual-pnCoTimes[simp]}: [w\mathit{f-dual\ n\ r}; w\mathit{f-dual\ n\ s}] \implies w\mathit{f-dual\ n\ (pnCoTimes\ b\ r\ s)}{\langle\text{proof}\rangle}\\

\textbf{lemma (in project)} l\mathit{ang-dual-pnCoTimes[simp]}: [l\mathit{ang-dual\ n\ r}; w\mathit{f-dual\ n\ s}] \implies l\mathit{ang-dual\ n\ (pnCoTimes\ b\ r\ s)} = l\mathit{ang-dual\ n\ (CoTimes\ b\ r\ s)}{\langle\text{proof}\rangle}\\

\textbf{fun} pn\mathit{CoPr} :: bool \Rightarrow 'a::linorder \mathit{rexp-dual} \Rightarrow 'a \mathit{rexp-dual}\ where\\ \textit{pnCoPr}\ b1 \ (CoZero\ b2) = (if\ b1 = b2\ then\ CoZero\ b2\ else\ CoPr\ b1\ (CoZero\ b2))\\ | \textit{pnCoPr}\ b1 \ (CoOne\ b2) = (if\ b1 = b2\ then\ CoOne\ b2\ else\ CoPr\ b1\ (CoOne\ b2))\\ | \textit{pnCoPr}\ b1 \ (CoPlus\ b2\ r\ s) = (if\ b1 = b2\ then\ \textit{pnCoPlus}\ b2\ (\textit{pnCoPr}\ b1\ r)\ (\textit{pnCoPr}\ b1\ s))\\ | \textit{pnCoPr}\ b\ r\ s = CoPr\ b\ r\\

\textbf{lemma (in alphabet)} w\mathit{f-dual-pnCoPr[simp]}: w\mathit{f-dual\ (Suc\ n\ r)} \implies w\mathit{f-dual\ n\ (pnCoPr\ b\ r)}{\langle\text{proof}\rangle}\\

\textbf{lemma (in project)} l\mathit{ang-dual-pnCoPr[simp]}: w\mathit{f-dual\ (Suc\ n\ r)} \implies l\mathit{ang-dual\ n\ (pnCoPr\ b\ r)} = l\mathit{ang-dual\ n\ (CoPr\ b\ r)}{\langle\text{proof}\rangle}\\

\textbf{primrec} p\mathit{norm-dual} :: 'a::linorder \mathit{rexp-dual} \Rightarrow 'a \mathit{rexp-dual}\ where\\ \textit{p\mathit{norm-dual}}\ (CoZero\ b) = (CoZero\ b)\\ | \textit{p\mathit{norm-dual}}\ (CoOne\ b) = (CoOne\ b)\\ | \textit{p\mathit{norm-dual}}\ (Co\mathit{Atom}\ b\ a) = (Co\mathit{Atom}\ b\ a)\\ | \textit{p\mathit{norm-dual}}\ (Co\mathit{Plus}\ b\ r\ s) = \textit{pnCoPlus}\ b\ (\textit{p\mathit{norm-dual}\ r})\ (\textit{p\mathit{norm-dual}\ s})\\ | \textit{p\mathit{norm-dual}}\ (Co\mathit{Times}\ b\ r\ s) = \textit{pnCoTimes}\ b\ (\textit{p\mathit{norm-dual}\ r})\ s\\ | \textit{p\mathit{norm-dual}}\ (Co\mathit{Star}\ b\ r) = Co\mathit{Star}\ b\ r\\ | \textit{p\mathit{norm-dual}}\ (Co\mathit{Pr}\ b\ r) = \textit{pnCoPr}\ b\ (\textit{p\mathit{norm-dual}\ r})\\

\textbf{lemma (in alphabet)} w\mathit{f-dual-pnorm-dual[simp]}: w\mathit{f-dual\ n\ r} \implies w\mathit{f-dual\ n\ (p\mathit{norm-dual}\ r)}{\langle\text{proof}\rangle}\\

\textbf{lemma (in project)} l\mathit{ang-dual-pnorm-dual[simp]}: w\mathit{f-dual\ n\ r} \implies l\mathit{ang-dual\ n\ (p\mathit{norm-dual}\ r)} = l\mathit{ang-dual\ n\ r}
\begin{proof}

\textbf{primrec} \textit{CoNot} \textbf{where}
\begin{align*}
\text{CoNot} (\text{CoZero} \ b) &= \text{CoZero} (\neg \ b) \\
\text{CoNot} (\text{CoOne} \ b) &= \text{CoOne} (\neg \ b) \\
\text{CoNot} (\text{CoAtom} \ b \ a) &= \text{CoAtom} (\neg \ b) \ a \\
\text{CoNot} (\text{CoPlus} \ b \ r \ s) &= \text{CoPlus} (\neg \ b) (\text{CoNot} \ r) (\text{CoNot} \ s) \\
\text{CoNot} (\text{CoTimes} \ b \ r \ s) &= \text{CoTimes} (\neg \ b) (\text{CoNot} \ r) (\text{CoNot} \ s) \\
\text{CoNot} (\text{CoStar} \ b \ r) &= \text{CoStar} (\neg \ b) (\text{CoNot} \ r) \\
\text{CoNot} (\text{CoPr} \ b \ r) &= \text{CoPr} (\neg \ b) (\text{CoNot} \ r)
\end{align*}

\textbf{primrec} \textit{rexp-dual-of} \textbf{where}
\begin{align*}
\text{rexp-dual-of Zero} &= \text{CoZero True} \\
\text{rexp-dual-of Full} &= \text{CoZero False} \\
\text{rexp-dual-of One} &= \text{CoOne True} \\
\text{rexp-dual-of (Atom} \ a) &= \text{CoAtom True} \ a \\
\text{rexp-dual-of (Plus} \ r \ s) &= \text{CoPlus True} (\text{rexp-dual-of} \ r) (\text{rexp-dual-of} \ s) \\
\text{rexp-dual-of (Times} \ r \ s) &= \text{CoTimes True} (\text{rexp-dual-of} \ r) (\text{rexp-dual-of} \ s) \\
\text{rexp-dual-of (Star} \ r) &= \text{CoStar True} (\text{rexp-dual-of} \ r) \\
\text{rexp-dual-of (Not} \ r) &= \text{CoNot} (\text{rexp-dual-of} \ r) \\
\text{rexp-dual-of (Inter} \ r \ s) &= \text{CoPlus False} (\text{rexp-dual-of} \ r) (\text{rexp-dual-of} \ s) \\
\text{rexp-dual-of (Pr} \ r) &= \text{CoPr True} (\text{rexp-dual-of} \ r)
\end{align*}

\textbf{lemma} (in alphabet) \textit{wf-dual-CoNot[simp]}: \textit{wf-dual n r \Rightarrow \text{wf-dual} n (\text{CoNot} \ r)}
\begin{proof}
\end{proof}

\textbf{lemma} (in project) \textit{lang-dual-CoNot[simp]}: \textit{wf-dual n r \Rightarrow \text{lang-dual} n (\text{CoNot} \ r) = \text{lists} \ (\Sigma n) - \text{lang-dual} n r}
\begin{proof}
\end{proof}

\textbf{lemma} (in alphabet) \textit{wf-dual-rexp-dual-of[simp]}: \textit{wf n r \Rightarrow \text{wf-dual} n (\text{rexp-dual-of} \ r)}
\begin{proof}
\end{proof}

\textbf{lemma} (in project) \textit{lang-dual-rexp-dual-of[simp]}: \textit{wf n r \Rightarrow \text{lang-dual} n (\text{rexp-dual-of} \ r) = \text{lang} n r}
\begin{proof}
\end{proof}

end

6 Deciding Equivalence of Π-Extended Regular Expressions

\textbf{lemma} \textit{image2p-in-rel}: \textit{BNF-Greatest-Fixpoint.image2p f g \text{ (in-rel} \ R) = \text{in-rel} \ (\text{map-prod} f \ g \cdot \ R)}
\begin{proof}
\end{proof}
lemma image2p-apply: \( \text{BNF-Greatest-Fixpoint.image2p} \ f \ g \ R \ x \ y = (\exists x' \ y'. \ R \ x' \ y' \land f x' = x \land g y' = y) \)

(\text{proof})

lemma rtrancl-fold-product:
shows \( \{((r, s), (f a r, f a s))| r s a. \ a \in A\}^* = \)
\( \{((r, s), (\text{fold} f w r, \text{fold} f w s))| r s w. \ w \in \text{lists} \ A\} (\text{is } ?L = ?R) \)
(\text{proof})

lemma in-fold-lQuot:
\( v \in \text{fold} lQuot w L \leftrightarrow w @ v \in L \)
(\text{proof})

lemma (in project) lang-eq-ext:
\[ [\text{wf n r}; \text{wf n s}] \implies (\text{lang n r} = \text{lang n s}) = (\forall w \in \text{lists}(\Sigma n). \ w \in \text{lang n r} \leftrightarrow w \in \text{lang n s}) \]
(\text{proof})

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

lemma L-deltas[\text{simp}]: \[ \text{wf-word n w}; \text{wf-state s} \] \implies L (\text{fold delta w s}) = \text{fold}
definition progression (infix → 60) where
  \( R \rightarrow S = (\forall s1 s2. R s1 s2 \implies \text{wf-state } s1 \land \text{wf-state } s2 \land \text{final } s1 = \text{final } s2 \land
  (\forall x \in \text{set } (\sigma n). \text{BNF-Greatest-Fixpoint.image2p post post } S (\text{post } (\text{delta } x
  s1))) (\text{post } (\text{delta } x s2)))) \)

lemma SUPR-progression[intro!]: \( \forall n. \exists m. X n \rightarrow Y m \implies (\text{SUP } n. X n) \rightarrow
  (\text{SUP } n. Y n) \)
  ⟨proof⟩

definition bisimulation where
  bisimulation \( R = R \rightarrow R \)

definition bisimulation-upto where
  bisimulation-upto \( R f = R \rightarrow f R \)

declare image2p[intro!] image2pE[elim!]
lemmas bisim-def = bisimulation-def progression-def
lemmas bisim-upto-def = bisimulation-upto-def progression-def

definition compatible where
  compatible \( f = (\text{mono } f \land (\forall R S. R \rightarrow S \implies f R \rightarrow f S)) \)

lemmas compat-def = compatible-def progression-def

lemma bisimulation-upto-bisimulation:
  assumes compatible \( f \) bisimulation-upto \( R f \)
  obtains \( S \) where bisimulation \( S R \leq S \)
  ⟨proof⟩

lemma bisimulation-eqL: bisimulation \( (\lambda s1 s2. \text{wf-state } s1 \land \text{wf-state } s2 \land L s1
  = L s2) \)
  ⟨proof⟩

lemma coinduction:
  assumes bisim[unfolded bisim-def]: bisimulation \( R \) and
  WF: \( \text{wf-state } s1 \text{ wf-state } s2 \) and \( R: R s1 s2 \)
  shows \( L s1 = L s2 \)
  ⟨proof⟩

lemma coinduction-upto:
  assumes bisimulation-upto \( R f \) and WF: \( \text{wf-state } s1 \text{ wf-state } s2 \) and \( R s1 s2 \)
  compatible \( f \)
  shows \( L s1 = L s2 \)
  ⟨proof⟩

fun test-invariant where

35
test-invariant (ws, - :: (ʼs × ʼs) list , - :: ʼs rel) = (case ws of [] ⇒ False | (w::ʼa list,p,q)#- ⇒ final p = final q)

fun test where test (ws, - :: ʼs 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;
  succ = 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) succ);
    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;
  succ = 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) succ)
  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-invariant-unfold: step-invariant \((w \neq ws, ps, N) = (ws', ps', N') \implies (\exists xs r s)\)
\[
w = (xs, r, s) \land ps' = (r, s) \neq ps \land \]
\[
ws' = ws \@ \text{remdups}' (\text{map-prod post post o snd}) (\text{filter} (\lambda (a \# ws, delta a r, delta a s)) (\sigma n)) \land \]
\[
N' = \text{set} (\text{map} (\lambda (a \# post a r, post a s))) (\sigma n) \cup N \]
⟨proof⟩

lemma invariant: invariant r s st \implies test-invariant st \implies invariant r s (step-invariant st)
⟨proof⟩

lemma step-commute: \(ws \neq \[] \implies (\text{case step-invariant (ws, ps, N) of (ws', ps', N') \implies (map snd ws', N')}) = \text{step} (\text{map snd ws, N})\)
⟨proof⟩

lemma closure-invariant-closure:
\[
\text{map-option} (\lambda (ws, ps, N). (\text{map snd ws, N})) (\text{closure-invariant (ws, ps, N)}) = \text{closure} (\text{map snd ws, N})
\]
⟨proof⟩

lemma assumes result: closure-invariant (\((([]), \text{init r, init s}), [], \{(\text{post (init r), post (init s)})\}) = \text{Some} (\[(\emptyset, N)\]) \land \text{WF: } wf n r wf n s
shows closure-invariant-sound: \(ws = [] \implies \text{lang n r = lang n s and}\)
\[
counterexample: ws \neq [] \implies \text{rev (fst (hd ws))} \in \text{lang n r} \iff \text{rev (fst (hd ws))} \notin \text{lang n s}
\]
⟨proof⟩

lemma closure-sound:
assumes result: closure (\(((\text{init r, init s}), [], \{(\text{post (init r), post (init s)})\})) = \text{Some} (\[(\emptyset, N)\]) \land \text{WF: } wf n r wf n s
shows \(\text{lang n r = lang n s}\)
⟨proof⟩

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

lemma check-eqv-sound:
assumes check-eqv r s and WF: wf n r wf n s
shows \(\text{lang n r = lang n s}\)

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

lemma counterexample-sound:
  assumes result: counterexample r s = Some w and WF: wf n r wf n s
  shows w ∈ lang n r ←→ w /∈ lang n s
(proof)

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

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

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

lemma match-correct: [wf-word n w; wf n s] ⇒ match s w ←→ w ∈ lang n s
(proof)

end

locale rexp-DFA = rexp-DA σ wf-atom project lookup init delta final wf-state post

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 \( post :: 's \Rightarrow 's \)
and \( L :: 's \Rightarrow 'a \ lang \)
and \( n :: \text{nat} \)
assumes \( \text{fin}: \text{finite} \{ \text{fold delta } w \ (\text{init } s) \mid w \ . \ True \} \)
begin

abbreviation \( \text{Reachable } s \equiv \{ \text{fold delta } w \ (\text{init } s) \mid w \ . \ True \} \)

lemma \( \text{closure-invariant-termination} \):
assumes \( \text{WF}: \ \text{wf n } r \ \text{wf n } s \)
and \( \text{result}: \ \text{closure-invariant} \ ([[]], \text{init } r, \text{init } s), [], \{(\text{post (init } r), \text{post (init } s))\}) = \text{None} \)
(is \( \text{closure-invariant} \ ([[]], ?r, ?s]), -) = \text{None} \ \text{is} \ ?cl = \text{None} \)
shows \( \text{False} \)

(\langle proof \rangle)

lemma \( \text{closure-termination} \):
assumes \( \text{WF}: \ \text{wf n } r \ \text{wf n } s \)
and \( \text{result}: \ \text{closure} \ ([\text{init } r, \text{init } s]), \{(\text{post (init } r), \text{post (init } s))\}) = \text{None} \)
shows \( \text{False} \)

(\langle proof \rangle)

lemma \( \text{closure-invariant-complete} \):
assumes \( \text{eq}: \ \text{lang } n \ r \equiv \text{lang } n \ s \)
and \( \text{WF}: \ \text{wf n } r \ \text{wf n } s \)
shows \( \exists \ ps \ N. \ \text{closure-invariant} \ ([[]], \text{init } r, \text{init } s), [], \{(\text{post (init } r), \text{post (init } s))\}) = \text{Some}([], \text{ps}, N) \)
is \( \exists - -. \ \text{closure-invariant} \ ([[]], ?r, ?s)], -) = - \ \text{is} \ \exists - -. \ ?cl = - \)
(\langle proof \rangle)

lemma \( \text{closure-complete} \):
assumes \( \text{lang } n \ r \equiv \text{lang } n \ s \ \text{wf n } r \ \text{wf n } s \)
shows \( \exists N. \ \text{closure} \ ([\text{init } r, \text{init } s]), \{(\text{post (init } r), \text{post (init } s))\}) = \text{Some} ([], N) \)

(\langle proof \rangle)

lemma \( \text{check-eqv-complete} \):
assumes \( \text{lang } n \ r \equiv \text{lang } n \ s \ \text{wf n } r \ \text{wf n } s \)
shows \( \text{check-eqv } r \ s \)

(\langle proof \rangle)

lemma \( \text{counterexample-complete} \):
assumes \( \text{lang } n \ r \not= \text{lang } n \ s \) and \( \text{WF}: \ \text{wf n } r \ \text{wf n } s \)
shows \( \exists w. \ \text{counterexample } r \ s = \text{Some } w \)

(\langle proof \rangle)

end

locale \( \text{rexp-DA-no-post} = \text{rexp-DA } \sigma \ \text{wf-atom } \text{project } \text{lookup } \text{init } \text{delta } \text{final } \text{wf-state} \)
id L n

for σ :: nat ⇒ 'a list

and wf-atom :: nat ⇒ 'b :: linorder ⇒ bool

and project :: 'a ⇒ 'a

and lookup :: 'b ⇒ 'a ⇒ bool

and init :: 'b rexp ⇒ 's

and delta :: 'a ⇒ 's ⇒ 's

and final :: 's ⇒ bool

and wf-state :: 's ⇒ bool

and L :: 's ⇒ 'a lang

and n :: nat

begin

lemma step-efficient[code]: step (ws, N) =

(let

  (r, s) = hd ws;

  new = remdups (filter (λ(r,s). (r,s) /∈ N) (map (λa. (delta a r, delta a s)) (σ n)))

  in (tl ws @ new, set new ∪ N))

(proof)

end

locale rexp-DFA-no-post = rexp-DFA σ wf-atom project lookup init delta final

wf-state id L

for σ :: nat ⇒ 'a list

and wf-atom :: nat ⇒ 'b :: linorder ⇒ bool

and project :: 'a ⇒ 'a

and lookup :: 'b ⇒ 'a ⇒ bool

and init :: 'b rexp ⇒ 's

and delta :: 'a ⇒ 's ⇒ 's

and final :: 's ⇒ bool

and wf-state :: 's ⇒ bool

and L :: 's ⇒ 'a lang

begin

sublocale rexp-DA-no-post {proof}

end

locale rexp-DA-sim = project set o σ wf-atom project lookup +

fixes init :: 'b rexp ⇒ 's

fixes sim-delta :: 's ⇒ 's list

fixes final :: 's ⇒ bool

fixes wf-state :: 's ⇒ bool


40
fixes $L :: 's \Rightarrow 'a$ lang
fixes $post :: 's \Rightarrow 's$
fixes $n :: nat$

assumes $L$-init[simp]: $\text{wf n r} \implies L\ (\text{init} \ r) = \text{lang} \ n \ r$
assumes $\text{final-iff-Nil}$[simp]: $\text{final} \ s \iff [] \in L \ s$
assumes $L$-wf-state[dest]: $\text{wf-state} \ s \implies L \ s \subseteq \text{lists (set} \ (\sigma \ n))$
assumes $\text{init-wf-state}$[simp]: $\text{wf n r} \implies \text{wf-state (init} \ r)$
assumes $L$-post[simp]: $\text{wf-state} \ s \implies L\ (\text{post} \ s) = L \ s$
assumes $\text{wf-state-post}$[simp]: $\text{wf-state} \ s \implies \text{wf-state (post} \ s)$
assumes $L$-sim-delta[simp]: $\text{wf-state} \ s \implies \text{map} \ (\text{sim-delta} \ s) = \text{map} \ (\lambda a. \text{lQuot} a \ (L \ s)) \ (\sigma \ n)$
assumes $\text{sim-delta-wf-state}$[simp]: $\text{wf-state} \ s \implies \forall s' \in \text{set (sim-delta} \ s). \text{wf-state} \ s'$

begin

definition $\text{delta} \ a \ s = \text{sim-delta} \ s \! \downarrow \text{index} \ (\sigma \ n) \ a$

lemma $\text{length-sim-delta}$[simp]: $\text{wf-state} \ s \implies \text{length} \ (\text{sim-delta} \ s) = \text{length} \ (\sigma \ n)$
  ⟨proof⟩

lemma $L$-delta[simp]: $[a \in \text{set} \ (\sigma \ n); \text{wf-state} \ s] \implies L\ (\text{delta} \ a \ s) = \text{lQuot} \ a \ (L \ s)$
  ⟨proof⟩

lemma $\text{delta-wf-state}$[simp]: $[a \in \text{set} \ (\sigma \ n); \text{wf-state} \ s] \implies \text{wf-state} \ (\text{delta} \ a \ s)$
  ⟨proof⟩

sublocale rexp-DA $\sigma$ wf-atom project lookup init delta final wf-state post L
  ⟨proof⟩

sublocale rexp-DA-sim-no-post!: rexp-DA-no-post $\sigma$ 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 = (∀ r ∈ 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⟩

42
lemma distinct-merge-distinct[simp]: \([\text{sorted } xs; \text{ distinct } xs; \text{ sorted } ys; \text{ distinct } ys]\) 
\[ \Rightarrow \text{distinct } (\text{merge-distinct } xs \; ys) \]
\langle proof \rangle

lemma sorted-list-of-set-merge-distinct[simp]: \([\text{sorted } xs; \text{ distinct } xs; \text{ sorted } ys; \text{ dis-

distinct } ys]\) 
\[ \Rightarrow \text{merge-distinct } xs \; ys = \text{sorted-list-of-set } (\text{set } xs \; \cup \; \text{set } ys) \]
\langle proof \rangle

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 \(z\) \(\iff\) \((\exists \; a \; b). \; z = f \; a \; b \; \land \; x = \text{Some } a \; \land \; y = \text{Some } b)\)
\langle proof \rangle

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 \(\iff\) Full \(\in\) toplevel-summands \(r\)
\langle proof \rangle

lemma Pluss-Some: Pluss \(r\) = Some \(xs\) \(\iff\) (Full \(\notin\) set \(xs\) \(\land\) \(xs\) = sorted-list-of-set (toplevel-summands \(r\) - \{Zero\}))
\langle proof \rangle

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 \(\iff\) Zero \(\in\) toplevel-inters \(r\)
\langle proof \rangle

lemma Inters-Some: Inters \(r\) = Some \(xs\) \(\iff\) (Zero \(\notin\) set \(xs\) \(\land\) \(xs\) = sorted-list-of-set (toplevel-inters \(r\) - \{Full\}))
\langle proof \rangle

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)
context alphabet begin

lemma wf-inPlus[simp]: \[ \text{wf } n \ r; \text{wf } n \ s \implies \text{wf } n \ (\text{inPlus } r \ s) \]
⟨proof⟩

lemma wf-inTimes[simp]: \[ \text{wf } n \ r; \text{wf } n \ s \implies \text{wf } n \ (\text{inTimes } r \ s) \]
⟨proof⟩

lemma wf-inStar[simp]: \[ \text{wf } n \ r \implies \text{wf } n \ (\text{inStar } r) \]
⟨proof⟩

lemma wf-inInter[simp]: \[ \text{wf } n \ r; \text{wf } n \ s \implies \text{wf } n \ (\text{inInter } r \ s) \]
⟨proof⟩

lemma wf-inNot[simp]: \[ \text{wf } n \ r \implies \text{wf } n \ (\text{inNot } r) \]
⟨proof⟩

lemma wf-inPr[simp]: \[ \text{wf } (\text{Suc } n) \ r \implies \text{wf } n \ (\text{inPr } r) \]
⟨proof⟩

lemma wf-inorm[simp]: \[ \text{wf } n \ r \implies \text{wf } n \ (\text{inorm } r) \]
⟨proof⟩

end

context project begin

lemma lang-inPlus[simp]: \[ \text{wf } n \ r; \text{wf } n \ s \implies \text{lang } n \ (\text{inPlus } r \ s) = \text{lang } n \ (\text{Plus } r \ s) \]
⟨proof⟩

lemma lang-inTimes[simp]: \[ \text{wf } n \ r; \text{wf } n \ s \implies \text{lang } n \ (\text{inTimes } r \ s) = \text{lang } n \ (\text{Times } r \ s) \]
⟨proof⟩

lemma lang-inStar[simp]: \[ \text{wf } n \ r \implies \text{lang } n \ (\text{inStar } r) = \text{lang } n \ (\text{Star } r) \]
⟨proof⟩

lemma Zero-toplevel-inters[dest]: \[ \text{Zero } \in \text{toplevel-inters } r \implies \text{lang } n \ r = \{ \} \]
⟨proof⟩

lemma toplevel-inters-Full: \[ \text{toplevel-inters } r = \{ \text{Full} \}; \text{wf } n \ r \implies \text{lang } n \ r = \text{lists } (\Sigma n) \]
⟨proof⟩

end
lemma toplevel-inters-subset-singleton[simp]: toplevel-inters \( r \subseteq \{s\} \Longleftrightarrow \text{toplevel-inters} r = \{s\} \) 
(proof)

lemma lang-inInter[simp]: \([\text{wf } n \ r; \ \text{wf } n \ s]\) \(\Rightarrow\) \(\text{lang } n \ (\text{inInter } r \ s) = \text{lang } n \ (\text{Inter } r \ s)\) 
(proof)

lemma lang-inNot[simp]: \(\text{wf } n \ r\) \(\Rightarrow\) \(\text{lang } n \ (\text{inNot } r) = \text{lang } n \ (\text{Not } r)\) 
(proof)

lemma lang-inPr[simp]: \(\text{wf } (\text{Suc } n) \ r\) \(\Rightarrow\) \(\text{lang } n \ (\text{inPr } r) = \text{lang } n \ (\text{Pr } r)\) 
(proof)

lemma lang-inorm[simp]: \(\text{wf } n \ r\) \(\Rightarrow\) \(\text{lang } n \ (\text{inorm } r) = \text{lang } n \ r\) 
(proof)

end

8 Partial Derivatives-like Normalization

fun pnPlus :: 'a::linorder rexp \(\Rightarrow\) 'a rexp \(\Rightarrow\) 'a rexp where
  pnPlus Zero r = r 
| pnPlus r Zero = r 
| pnPlus One r = 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 \leq s then Plus r (Plus s t) 
   else Plus s (pnPlus r t)) 
| pnPlus r s = 
  (if r = s then r 
   else if r \leq s then Plus r s 
   else Plus s r)

lemma (in alphabet) wf-pnPlus[simp]: \([\text{wf } n \ r; \ \text{wf } n \ s]\) \(\Rightarrow\) \(\text{wf } n \ (\text{pnPlus } r \ s)\) 
(proof)

lemma (in project) lang-pnPlus[simp]: \([\text{wf } n \ r; \ \text{wf } n \ s]\) \(\Rightarrow\) \(\text{lang } n \ (\text{pnPlus } r \ s)\) = \(\text{lang } n \ (\text{Plus } r \ s)\) 
(proof)

fun pnTimes :: 'a::linorder rexp \(\Rightarrow\) 'a rexp \(\Rightarrow\) '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]: \[ \text{wf } n \text{ r; wf } n \text{ s} \implies \text{wf } n \text{ (pnTimes r s)} \]
(proof)

lemma (in project) lang-pnTimes[simp]: \[ \text{wf } n \text{ r; wf } n \text{ s} \implies \text{lang } n \text{ (pnTimes r s)} = \text{lang } n \text{ (Times r s)} \]
(proof)

fun pnInter :: 'a::linorder rexp \Rightarrow 'a rexp \Rightarrow 'a rexp where
\[ \text{pnInter Zero } r = \text{Zero} \]
\[ \text{pnInter } r \text{ Zero } = \text{Zero} \]
\[ \text{pnInter } \text{Full } r = r \]
\[ \text{pnInter } (\text{Plus } r \text{ s } t) = \text{pnPlus } (\text{pnInter } r \text{ t}) \text{ (pnInter } s \text{ t}) \]
\[ \text{pnInter } r \text{ (Plus } s \text{ t }) = \text{pnPlus } (\text{pnInter } r \text{ s}) \text{ (pnInter } r \text{ t}) \]
\[ \text{pnInter } (\text{Inter } r \text{ s } t) = \text{pnInter } r \text{ (pnInter } s \text{ t}) \]
\[ \text{pnInter } r \text{ (Inter } s \text{ t }) = \]
\[ \text{if } r = s \text{ then Inter } s \text{ t} \]
\[ \text{else if } r \leq s \text{ then Inter } r \text{ (Inter } s \text{ t}) \]
\[ \text{else Inter } s \text{ (pnInter } r \text{ t}) \]
\[ \text{pnInter } r \text{ s } = \]
\[ \text{if } r = s \text{ then } s \]
\[ \text{else if } r \leq s \text{ then Inter } r \text{ s} \]
\[ \text{else Inter } s \text{ r} \]

lemma (in alphabet) wf-pnInter[simp]: \[ \text{wf } n \text{ r; wf } n \text{ s} \implies \text{wf } n \text{ (pnInter r s)} \]
(proof)

lemma (in project) lang-pnInter[simp]: \[ \text{wf } n \text{ r; wf } n \text{ s} \implies \text{lang } n \text{ (pnInter r s)} = \text{lang } n \text{ (Inter r s)} \]
(proof)

fun pnNot :: 'a::linorder rexp \Rightarrow 'a rexp \Rightarrow 'a rexp where
\[ \text{pnNot } (\text{Plus } r \text{ s } r) = \text{pnInter } (\text{pnNot } r) \text{ (pnNot } s) \]
\[ \text{pnNot } (\text{Inter } r \text{ s } r) = \text{pnPlus } (\text{pnNot } r) \text{ (pnNot } s) \]
\[ \text{pnNot } \text{Full } = \text{Zero} \]
\[ \text{pnNot } \text{Zero } = \text{Full} \]
\[ \text{pnNot } (\text{Not } r) = r \]
\[ \text{pnNot } r = \text{Not } r \]

lemma (in alphabet) wf-pnNot[simp]: \[ \text{wf } n \text{ r } \implies \text{wf } n \text{ (pnNot } r) \]
(proof)

lemma (in project) lang-pnNot[simp]: \[ \text{wf } n \text{ r } \implies \text{lang } n \text{ (pnNot } r) = \text{lang } n \text{ (Not } r) \]
(proof)

fun pnPr :: 'a::linorder rexp \Rightarrow 'a rexp \Rightarrow 'a rexp where
\[ \text{pnPr } \text{Zero } = \text{Zero} \]
\[ \text{pnPr } \text{One } = \text{One} \]
\[ pnPr \ (Plus \ r \ s) = pnPlus \ (pnPr \ r) \ (pnPr \ s) \]
\[ \] \[ pnPr \ r = Pr \ r \]

**lemma (in alphabet)** \( \text{wf-pnPr[simp]} \): \( \text{wf} \ (Suc \ n) \ r \implies \text{wf} \ n \ (pnPr \ r) \)

\( \langle \text{proof} \rangle \)

**lemma (in project)** \( \text{lang-pnPr[simp]} \): \( \text{wf} \ (Suc \ n) \ r \implies \text{lang} \ n \ (pnPr \ r) = \text{lang} \ n \ (Pr \ r) \)

\( \langle \text{proof} \rangle \)

**primrec** \( \text{pnorm} :: 'a::linorder rexp \Rightarrow 'a \ rexp \) where
\[ \text{pnorm} \ Zero = Zero \]
\[ \text{pnorm} \ Full = Full \]
\[ \text{pnorm} \ One = One \]
\[ \text{pnorm} \ (Plus \ r \ s) = pnPlus \ (pnorm \ r) \ (pnorm \ s) \]
\[ \text{pnorm} \ (Times \ r \ s) = pnTimes \ (pnorm \ r) \ s \]
\[ \text{pnorm} \ (Star \ r) = Star \ r \]
\[ \text{pnorm} \ (Inter \ r \ s) = pnInter \ (pnorm \ r) \ (pnorm \ s) \]
\[ \text{pnorm} \ (Not \ r) = pnNot \ (pnorm \ r) \]
\[ \text{pnorm} \ (Pr \ r) = pnPr \ (pnorm \ r) \]

**lemma (in alphabet)** \( \text{wf-pnNorm[simp]} \): \( \text{wf} \ n \ r \implies \text{wf} \ n \ (pnorm \ r) \)

\( \langle \text{proof} \rangle \)

**lemma (in project)** \( \text{lang-pnNorm[simp]} \): \( \text{wf} \ n \ r \implies \text{lang} \ n \ (pnorm \ r) = \text{lang} \ n \ r \)

\( \langle \text{proof} \rangle \)

### 9 Monadic Second-Order Logic Formulas

#### 9.1 Interpretations and Encodings

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

**abbreviation** \( \text{enc-atom-bool} \ I \ n \equiv \text{map} \ (\lambda x. \ \text{case} \ x \ \text{of} \ \text{Inl} \ p \ \Rightarrow \ n = p \ | \ \text{Inr} \ P \ \Rightarrow \ n \in P) \ I \)

**abbreviation** \( \text{enc-atom} \ I \ n \ a \equiv (a, \ \text{enc-atom-bool} \ I \ n) \)

#### 9.2 Syntax and Semantics of MSO

**datatype** \( 'a \ formula = \)
\[ \ FQ \ 'a \ nat \]
\[ \ FLess \ nat \ nat \]
\[ \ FIn \ nat \ nat \]
\[ \ FNot \ 'a \ formula \]
\[ \ FOr \ 'a \ formula \ 'a \ formula \]
\[ \ FAnd \ 'a \ formula \ 'a \ formula \]
\[ \ FExists \ 'a \ formula \]

48
\[ \text{FEXISTS 'a formula} \]

\[
\begin{align*}
\text{primrec } & \text{FOV :: 'a formula } \Rightarrow \text{ nat set where} \\
& \text{FOV (FQ a m)} = \{m\} \\
& \text{FOV (FLess m1 m2)} = \{m1, m2\} \\
& \text{FOV (FIn m M)} = \{m\} \\
& \text{FOV (FNot } \varphi) = \text{ FOV } \varphi \\
& \text{FOV (FOr } \varphi_1 \varphi_2) = \text{ FOV } \varphi_1 \cup \text{ FOV } \varphi_2 \\
& \text{FOV (FAnd } \varphi_1 \varphi_2) = \text{ FOV } \varphi_1 \cup \text{ FOV } \varphi_2 \\
& \text{FOV (FExists } \varphi) = (\lambda x. x - 1) \cdot (\text{ FOV } \varphi - \{0\}) \\
& \text{FOV (FEXISTS } \varphi) = (\lambda x. x - 1) \cdot \text{ FOV } \varphi
\end{align*}
\]

\[
\begin{align*}
\text{primrec } & \text{SOV :: 'a formula } \Rightarrow \text{ nat set where} \\
& \text{SOV (FQ a m)} = \{\} \\
& \text{SOV (FLess m1 m2)} = \{\} \\
& \text{SOV (FIn m M)} = \{M\} \\
& \text{SOV (FNot } \varphi) = \text{ SOV } \varphi \\
& \text{SOV (FOr } \varphi_1 \varphi_2) = \text{ SOV } \varphi_1 \cup \text{ SOV } \varphi_2 \\
& \text{SOV (FAnd } \varphi_1 \varphi_2) = \text{ SOV } \varphi_1 \cup \text{ SOV } \varphi_2 \\
& \text{SOV (FExists } \varphi) = (\lambda x. x - 1) \cdot \text{ SOV } \varphi \\
& \text{SOV (FEXISTS } \varphi) = (\lambda x. x - 1) \cdot (\text{ SOV } \varphi - \{0\})
\end{align*}
\]

\[
\begin{align*}
\text{definition } & \sigma = (\Lambda \Sigma. \ n. \ \text{concat (map (}\lambda a. \ (a, bs)) \Sigma) \ \text{(List.n-lists n [True, False]))}) \\
\text{definition } & \pi = (\lambda(a, bs). (a, tl bs)) \\
\text{definition } & \varepsilon = (\Lambda \Sigma. \ \text{if } a \in \text{ set } \Sigma \text{ then } [(a, True \# bs), (a, False \# bs)] \text{ else []})
\end{align*}
\]

\[
\text{datatype 'a atom =} \\
\quad \text{Singleton 'a bool list} \\
\quad \text{AQ nat 'a} \\
\quad \text{Arbitrary-Except nat bool} \\
\quad \text{Arbitrary-Except2 nat nat}
\]

\[
\text{derive linorder atom}
\]

\[
\begin{align*}
\text{fun } & \text{wf-atom where} \\
& \text{wf-atom } \Sigma n \ (\text{Singleton } a \ bs) = (a \in \text{ set } \Sigma \land \text{ length } bs = n) \\
& \text{wf-atom } \Sigma n \ (\text{AQ m a}) = (a \in \text{ set } \Sigma \land m < n) \\
& \text{wf-atom } \Sigma n \ (\text{Arbitrary-Except m } \cdot) = (m < n) \\
& \text{wf-atom } \Sigma n \ (\text{Arbitrary-Except2 m1 m2}) = (m1 < n \land m2 < n)
\end{align*}
\]

\[
\begin{align*}
\text{fun } & \text{lookup where} \\
& \text{lookup (Singleton } a' \ bs') \ (a, bs) = (a = a' \land bs = bs') \\
& \text{lookup (AQ m a') } (a, bs) = (a = a' \land bs \cdot m) \\
& \text{lookup (Arbitrary-Except m b) } (\cdot, bs) = (bs \cdot m = b) \\
& \text{lookup (Arbitrary-Except2 m1 m2) } (\cdot, bs) = (bs \cdot m1 \land bs \cdot m2)
\end{align*}
\]

\[
\text{lemma } \pi-\sigma: \pi \cdot (\text{set } a \ \sigma \ \Sigma) \ (n + 1) = (\text{set } a \ \sigma \ \Sigma) \ n
\]

\[\langle \text{proof} \rangle\]
locale formula = embed2 set o (σ Σ) wf-atom Σ π lookup ε Σ split Singleton
for Σ :: 'a :: linorder list
assumes nonempty: Σ ≠ []
begin

abbreviation Σ-product-lists n ≡ List.maps (λbools. map (λa. (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 n M) = (m < n ∧ M < n)
| pre-wf-formula n (FNot ϕ) = pre-wf-formula n ϕ
| pre-wf-formula n (FOr ϕ1 ϕ2) = (pre-wf-formula n ϕ1 ∧ pre-wf-formula n ϕ2)
| pre-wf-formula n (FAnd ϕ1 ϕ2) = (pre-wf-formula n ϕ1 ∧ pre-wf-formula n ϕ2)
| 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: \( i < \text{length } w \之事 \\text{length } I = n \land p \in \text{set } \Sigma \) \iff \text{enc-atom } I \ i \ p \in \text{set } (\sigma \Sigma \ n)

\(\text{proof}\)

lemmas enc-atom-σ = iffD1[\text{OF enc-atom-σ-eq, OF - conjI}]

lemma enc-atom-bool-take-drop-True:
\[ r < \text{length } I; \ case \ I \ ! \ r \ of \ \text{Inl } p' \Rightarrow p = p' \mid \text{Inr } P \Rightarrow p \in P \] \implies
\(\text{enc-atom-bool } I \ p = \text{take } r \ (\text{enc-atom-bool } I \ p) \circ \text{True} \# \text{drop } (\text{Suc } r) \)

\(\text{proof}\)

lemma enc-atom-bool-take-drop-True2:
\[ r < \text{length } I; \ case \ I \ ! \ r \ of \ \text{Inl } p' \Rightarrow p = p' \mid \text{Inr } P \Rightarrow p \in P; \ s < \text{length } I; \ case \ I \ ! \ s \ of \ \text{Inl } p' \Rightarrow p = p' \mid \text{Inr } P \Rightarrow p \in P; \ r < s \] \implies
\(\text{enc-atom-bool } I \ p = \text{take } r \ (\text{enc-atom-bool } I \ p) \circ \text{True} \# \text{drop } (\text{Suc } r) \)

\(\text{take } (s - \text{Suc } r) \ (\text{drop } (\text{Suc } r) \ (\text{enc-atom-bool } I \ p)) \circ \text{True} \# \text{drop } (\text{Suc } s) \)

\(\text{enc-atom-bool } I \ p \)

\(\text{proof}\)

lemma enc-atom-bool-take-drop-False:
\[ r < \text{length } I; \ case \ I \ ! \ r \ of \ \text{Inl } p' \Rightarrow p \neq p' \mid \text{Inr } P \Rightarrow p \notin P \] \implies
\(\text{enc-atom-bool } I \ p = \text{take } r \ (\text{enc-atom-bool } I \ p) \circ \text{False} \# \text{drop } (\text{Suc } r) \)

\(\text{proof}\)

lemma enc-atom-lang-AQ:
\[ r < \text{length } I; \ case \ I \ ! \ r \ of \ \text{Inl } p' \Rightarrow p = p' \mid \text{Inr } P \Rightarrow p \in P; \ length \ I = n; \ a \in \text{set } \Sigma \] \implies
\(\text{enc-atom } I \ p \ a \in \text{lang } n \ (\text{Atom } (AQ \ r \ a)) \)

\(\text{proof}\)

lemma enc-atom-lang-Arbitrary-Except-True:
\[ r < \text{length } I; \ case \ I \ ! \ r \ of \ \text{Inl } p' \Rightarrow p = p' \mid \text{Inr } P \Rightarrow p \in P; \ length \ I = n; \ a \in \text{set } \Sigma \] \implies
\(\text{enc-atom } I \ p \ a \in \text{lang } n \ (\text{Atom } (\text{Arbitrary-Except } r \ True)) \)

\(\text{proof}\)

lemma enc-atom-lang-Arbitrary-Except2:
\[ r < \text{length } I; \ s < \text{length } I; \ case \ I \ ! \ r \ of \ \text{Inl } p' \Rightarrow p = p' \mid \text{Inr } P \Rightarrow p \in P; \ case \ I \ ! \ s \ of \ \text{Inl } p' \Rightarrow p = p' \mid \text{Inr } P \Rightarrow p \in P; \ length \ I = n; \ a \in \text{set } \Sigma \] \implies
\(\text{enc-atom } I \ p \ a \in \text{lang } n \ (\text{Atom } (\text{Arbitrary-Except2 } r \ s)) \)

\(\text{proof}\)

lemma enc-atom-lang-Arbitrary-Except-False:
\[ r < \text{length } I; \ case \ I \ ! \ r \ of \ \text{Inl } p' \Rightarrow p \neq p' \mid \text{Inr } P \Rightarrow p \notin P; \ length \ I = n; \ a \in \text{set } \Sigma \] \implies
\(\text{enc-atom } I \ p \ a \in \text{lang } n \ (\text{Atom } (\text{Arbitrary-Except } r \ False)) \)

\(\text{proof}\)
lemma AQ-D:
  assumes \( v \in \text{lang } n \,(\text{Atom } (AQ \, m \, a)) \) \( m < n \ a \in \text{set } \Sigma \)
  shows \( \exists x. \, v = [x] \land \text{fst } x = a \land \text{snd } x \uparrow \)
  \( \langle \text{proof} \rangle \)

lemma Arbitrary-ExceptD:
  assumes \( v \in \text{lang } n \,(\text{Atom } (\text{Arbitrary-Except } r \, b)) \) \( r < n \)
  shows \( \exists x. \, v = [x] \land \text{snd } x \uparrow \)
  \( \langle \text{proof} \rangle \)

lemma Arbitrary-Except2D:
  assumes \( v \in \text{lang } n \,(\text{Atom } (\text{Arbitrary-Except2 } r \, s)) \) \( r < n \ s < n \)
  shows \( \exists x. \, v = [x] \land \text{snd } x \uparrow \)
  \( \langle \text{proof} \rangle \)

lemma star-Arbitrary-ExceptD:
  \[ [v \in \text{star } (\text{lang } n \,(\text{Atom } (\text{Arbitrary-Except } r \, b))); \, r < n; \, i < \text{length } v] \implies \]
  \( \text{snd } (v \uparrow i) \uparrow \)
  \( \langle \text{proof} \rangle \)

end

end

10 M2L

10.1 Encodings

custom context formula
begin

fun enc :: 'a interp \Rightarrow ('a \times \text{bool list}) list where
  enc (w, I) = map-index (enc-atom I) w

abbreviation wf-interp w I \equiv (\text{length } w > 0 \land
  (\forall a \in \text{set } w. \, a \in \text{set } \Sigma) \land
  (\forall x \in \text{set } I. \, \text{case } x \text{ of Inl } p \Rightarrow p < \text{length } w \mid \text{Inr } P \Rightarrow \forall p \in P. \, p < \text{length } w))

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

fun satisfies :: 'a interp \Rightarrow 'a formula \Rightarrow \text{bool} (\text{infix } \cong 50) where
  (w, I) \models FQ a m = (w \uparrow (\text{case } I \mid m \text{ of Inl } p \Rightarrow p) = a)
  | (w, I) \models FLess m1 m2 = ((\text{case } I \mid m1 \text{ of Inl } p \Rightarrow p) \cong (\text{case } I \mid m2 \text{ of Inl } p \Rightarrow p))
  | (w, I) \models FIn m M = ((\text{case } I \mid m \text{ of Inl } p \Rightarrow p) \in (\text{case } I \mid M \text{ of Inr } P \Rightarrow P))
\[ (w, I) \models \text{FNot } \varphi = (\neg (w, I) \models \varphi) \]
\[ (w, I) \models (\text{FOr } \varphi_1 \varphi_2) = ((w, I) \models \varphi_1 \lor (w, I) \models \varphi_2) \]
\[ (w, I) \models (\text{FAnd } \varphi_1 \varphi_2) = ((w, I) \models \varphi_1 \land (w, I) \models \varphi_2) \]
\[ (w, I) \models (\text{FExists } \varphi) = (\exists p. p \in \{0 \ldots \text{length } w - 1\} \land (w, \text{Inl } p \# I) \models \varphi) \]
\[ (w, I) \models (\text{FEXISTS } \varphi) = (\exists P. P \subseteq \{0 \ldots \text{length } w - 1\} \land (w, \text{Inr } P \# I) \models \varphi) \]

**Definition**\( \text{lang}_{M2L} :: \text{nat} \Rightarrow \text{'a formula} \Rightarrow (\text{'a} \times \text{bool list}) \text{ list set} \) where
\[ \text{lang}_{M2L} n \varphi = \{ \text{enc} (w, I) \mid \text{length } I = n \land \text{wf-interp-for-formula} (w, I) \varphi \land \text{satisfies} (w, I) \varphi \} \]

**Definition**\( \text{dec-word} \equiv \text{map} \text{ fst} \)

**Definition**\( \text{positions-in-row } w i = \) Option.\text{these} (set (map-index (\lambda a-bs. \text{if nth} (\text{snd} a-bs) i then Some p else None) w))

**Definition**\( \text{dec-interp } n \text{ FO} (w :: (\text{'a} \times \text{bool list}) \text{ list}) \equiv \text{map} (\lambda i. \) if \( i \in \text{FO} \) then Inl (the-elem (\text{positions-in-row } w i)) else Inr (\text{positions-in-row } w i)) [0..<n]

**Lemma** \( \text{positions-in-row: positions-in-row } w i = \{ p. \ p < \text{length } w \land \text{snd} (w ! p) ! i \} \)
\( \langle \text{proof} \rangle \)

**Lemma** \( \text{positions-in-row-unique: } \exists! p. \ p < \text{length } w \land \text{snd} (w ! p) ! i \implies \) the-elem (\text{positions-in-row } w i) = (THE p. \ p < \text{length } w \land \text{snd} (w ! p) ! i) \)
\( \langle \text{proof} \rangle \)

**Lemma** \( \text{positions-in-row-length: } \exists! p. \ p < \text{length } w \land \text{snd} (w ! p) ! i \implies \) the-elem (\text{positions-in-row } w i) < \text{length } w \)
\( \langle \text{proof} \rangle \)

**Lemma** \( \text{dec-interp-Inl: } [i \in \text{FO}; i < n] \implies \exists p. \text{ dec-interp } n \text{ FO } x \ i = \text{Inl } p \)
\( \langle \text{proof} \rangle \)

**Lemma** \( \text{dec-interp-not-Inr: } [\text{dec-interp } n \text{ FO } x \ i = \text{Inr } P; i \in \text{FO}; i < n] \implies \) False
\( \langle \text{proof} \rangle \)

**Lemma** \( \text{dec-interp-Inr: } [i \notin \text{FO}; i < n] \implies \exists P. \text{ dec-interp } n \text{ FO } x \ i = \text{Inr } P \)
\( \langle \text{proof} \rangle \)

**Lemma** \( \text{dec-interp-not-Inl: } [\text{dec-interp } n \text{ FO } x \ i = \text{Inl } p; i \notin \text{FO}; i < n] \implies \) False
\( \langle \text{proof} \rangle \)

**Lemma** \( \text{Inl-dec-interp-length:} \)

53
assumes $\forall i \in FO. \exists! p. p < length w \land \text{snd} (w ! p) ! i$
shows $\text{Inl } p \in \text{set } (\text{dec-interp } n \ FO \ w) \implies p < length w$

lemma $\text{Inr-dec-interp-length}$:
assumes $\exists! x. P x$
shows $\text{the-elem } (\text{Collect } P) = (\text{The } P)$

lemma $\text{enc-atom-dec}$:
assumes $\forall i \in FO. i < n \implies (\exists! p. p < length w \land \text{snd} (w ! p) ! i); p < length w$
shows $\text{enc-atom } (\text{dec-interp } n \ FO \ w) \ p \ (\text{fst} (w ! p)) = w ! p$

lemma $\text{dec-word-enc}$:
assumes $\forall i \in FO. i < n \implies (\exists! p. p < length w \land \text{snd} (w ! p) ! i);
\forall \ p \in P. \ p < length w; \text{wf-interp } w \ I$
shows $\text{dec-word } w, \text{dec-interp } n \ FO \ w = w$

lemma $\text{dec-interp-enc}$:
assumes $\text{dec-interp } n \ FO \ (\text{enc } (w, I)) ! i = \text{Inl } p'; I ! i = \text{Inl } p; i \in FO; i < n; \text{length } I = n; p < length w; \text{wf-interp } w \ I$
shows $p = p'$

lemma $\text{length-dec-interp}$:
assumes $\text{dec-interp } n \ FO \ (\text{enc } (w, I)) ! i = \text{Inr } P'; I ! i = \text{Inr } P; i \notin FO; i < n; \text{length } I = n; \forall \ p \in P. \ p < length w$
shows $P = P'$

lemma $\text{enc-unique}$:
assumes $\text{wf-interp } w \ I i < length I$
shows $\exists! p. I ! i = \text{Inl } p \implies \exists! p. p < length (\text{enc } (w, I)) \land \text{snd} (\text{enc } (w, I) ! p) ! i$

lemma $\text{length-dec-interp}$:
assumes $\text{dec-interp } n \ FO \ x$
shows $\text{length } (\text{dec-interp } n \ FO \ x) = n$
lemma nth-dec-interp[simp]: $i < n \implies \text{dec-interp } n \{\} \ x ! i = \text{Inr (positions-in-row } x i)\$

⟨proof⟩

lemma set-σ D[simp]: $(a, bs) \in \text{set } (\sigma \Sigma n) \implies a \in \text{set } \Sigma$

⟨proof⟩

lemma lang-ENC:
assumes $\text{FO} \subseteq \{0 ..< n\} \text{ SO} \subseteq \{0 ..< n\} - \text{FO}$
shows $\text{lang } n \ (\text{ENC } n \text{ FO}) - \{\} = \{\text{enc (w, I) | w I . length I = n} \cdot \text{wf-interp } w I \\land \langle \forall i \in \text{FO. case } I ! i \text{ of } \text{Inl } - \implies \text{True} | \text{Inr } - \implies \text{False} \rangle \land \langle \forall i \in \text{SO. case } I ! i \text{ of } \text{Inl } - \implies \text{False} | \text{Inr } - \implies \text{True} \rangle \langle \text{is } ?L = ?R \rangle \rangle$

⟨proof⟩

lemma lang-ENC-formula:
assumes $\text{wf-formula } n \varphi$
shows $\text{lang } n \ (\text{ENC } n \ (\text{FOV } \varphi)) - \{\} = \{\text{enc (w, I) | w I . length I = n} \cdot \text{wf-interp-for-formula } (w, I) \varphi\}

⟨proof⟩

10.2 Welldefinedness of enc wrt. Models

lemma enc-alt-def:
$\text{enc (w, x \# I)} = \text{map-index } (\lambda (a, bs). (a, (\text{case } x \text{ of } \text{Inl } p - \implies n = p | \text{Inr } P \implies n \in P) \# bs)) (\text{enc (w, I)})$

⟨proof⟩

lemma enc-extend-interp: $\text{enc (w, I)} = \text{enc (w', I')} \implies \text{enc (w, x \# I)} = \text{enc (w', x \# I')}$

⟨proof⟩

lemma wf-interp-for-formula-FExists:
$[\text{wf-formula (length I) (FExists } \varphi); w \neq \{\}] \implies \text{wf-interp-for-formula } (w, I) \ (FExists \varphi) \iff \langle \forall p < \text{length } w. \text{wf-interp-for-formula } (w, \text{Inl } p \# I) \varphi \rangle$

⟨proof⟩

lemma wf-interp-for-formula-any-Inl: $\text{wf-interp-for-formula } (w, \text{Inl } p \# I) \varphi \iff \forall p < \text{length } w. \text{wf-interp-for-formula } (w, \text{Inl } p \# I) \varphi$

⟨proof⟩

lemma wf-interp-for-formula-FEXISTS:
$[\text{wf-formula (length I) (FEXISTS } \varphi); w \neq \{\}] \implies \text{wf-interp-for-formula } (w, I) \ (FEXISTS \varphi) \iff \langle \forall P \subseteq \{0 .. \text{length } w - 1\}. \text{wf-interp-for-formula } (w, \text{Inr } P \# I) \varphi \rangle$

⟨proof⟩

55
**lemma** \text{wf-interp-for-formula-angy-Inr}: \text{wf-interp-for-formula} (w, \, \text{Inr} \, P \neq I) \, \varphi \implies \\
\forall P \subseteq \{0 \ldots \text{length} \, w - 1\}. \, \text{wf-interp-for-formula} (w, \, \text{Inr} \, P \neq I) \, \varphi \\
(\text{proof})

**lemma** \text{enc-word-length}: \text{enc} \,(w, \, I) = \text{enc} \,(w', \, I') \implies \text{length} \, w = \text{length} \, w' \\
(\text{proof})

**lemma** \text{enc-length}: \\
\text{assumes} \, w \neq [] \, \text{enc} \,(w, \, I) = \text{enc} \,(w', \, I') \\
\text{shows} \, \text{length} \, I = \text{length} \, I' \\
(\text{proof})

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

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

**lemma** \text{enc-wf-interp}: \\
\text{assumes} \, \text{wf-formula} \,(\text{length} \, I) \, \varphi \, \text{wf-interp-for-formula} \,(w, \, I) \, \varphi \\
\text{shows} \, \text{wf-interp-for-formula} \,(\text{dec-word} \,(\text{enc} \,(w, \, I)), \, \text{dec-interp} \,(\text{length} \, I) \, (\text{FOV} \, \varphi) \,(\text{enc} \,(w, \, I))) \, \varphi \\
\text{(is \, wf-interp-for-formula} \,(\ldots, \, \text{dec}) \, \varphi) \\
(\text{proof})

**lemma** \text{enc-welldef}: \llbracket \text{enc} \,(w, \, I) = \text{enc} \,(w', \, I'); \, \text{wf-formula} \,(\text{length} \, I) \, \varphi; \, \text{wf-interp-for-formula} \,(w, \, I) \, \varphi; \, \text{wf-interp-for-formula} \,(w', \, I') \, \varphi \rrbracket \implies \text{satisfies} \,(w, \, I) \, \varphi \iff \text{satisfies} \,(w', \, I') \, \varphi \\
(\text{proof})

**lemma** \text{langM2L-FOr}: \\
\text{assumes} \, \text{wf-formula} \, n \, (\text{FOr} \, \varphi \, 1 \, \varphi \, 2) \\
\text{shows} \, \text{langM2L} \, n \, (\text{FOr} \, \varphi \, 1 \, \varphi \, 2) \subseteq \\
\{ \text{langM2L} \, n \, \varphi \, 1 \, \cup \, \text{langM2L} \, n \, \varphi \, 2 \} \cap \{ \text{enc} \,(w, \, I) \mid w, \, I. \, \text{length} \, I = n \land \, \text{wf-interp-for-formula} \,(w, \, I) \, (\text{FOr} \, \varphi \, 1 \, \varphi \, 2) \} \\
\text{(is} \, \subseteq \, (\text{?L1} \, \cup \, \text{?L2}) \cap \, \text{?ENC}) \\
(\text{proof})

**lemma** \text{langM2L-FAnd}: \\
\text{assumes} \, \text{wf-formula} \, n \, (\text{FAnd} \, \varphi \, 1 \, \varphi \, 2) \\
\text{shows} \, \text{langM2L} \, n \, (\text{FAnd} \, \varphi \, 1 \, \varphi \, 2) \subseteq \\
\{ \text{langM2L} \, n \, \varphi \, 1 \, \cap \, \text{langM2L} \, n \, \varphi \, 2 \} \cap \{ \text{enc} \,(w, \, I) \mid w, \, I. \, \text{length} \, I = n \land \, \text{wf-interp-for-formula} \,(w, \, I) \, (\text{FAnd} \, \varphi \, 1 \, \varphi \, 2) \} \\
\text{(is} \, \subseteq \, (\text{?L1} \, \cap \, \text{?L2} \, \cap \, \text{?ENC}) \\
(\text{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-of (FAnd ϕ) (ENC n (FOV (FNot ϕ))))
| rexp-of n (FOr ϕ1 ϕ2) = Inter (Plus (rexp-of ϕ1) (rexp-of ϕ2)) (ENC n (FOV (FAnd ϕ1 ϕ2))
| rexp-of n (FAnd ϕ1 ϕ2) = INTERSECT [rexp-of ϕ1, rexp-of ϕ2, ENC n (FOV (FAnd ϕ1 ϕ2))]
| rexp-of n (FExists φ) = Pr (rexp-of (n + 1) φ)
| rexp-of n (FEXISTS φ) = Pr (rexp-of (n + 1) φ)

fun rexp-of-alt :: nat ⇒ 'a formula ⇒ ('a atom) rexp where
  rexp-of-alt n (FQ a m) = TIMES [Full, Atom (AQ m a), Full]
| rexp-of-alt n (FLess m1 m2) = (if m1 = m2 then Zero else TIMES
  [Full, Atom (Arbitrary-Except m1 True), Full, Atom (Arbitrary-Except m2 True), Full])
| rexp-of-alt n (FIn m M) = TIMES [Full, Atom (Arbitrary-Except2 m M), Full]
| rexp-of-alt n (FNot ϕ) = rexp-alt (FAnd ϕ) (ENC n (FOV (FNot ϕ)))
| rexp-of-alt n (FOr ϕ1 ϕ2) = Plus (rexp-alt ϕ1) (rexp-alt ϕ2)
| rexp-of-alt n (FAnd ϕ1 ϕ2) = Inter (rexp-alt ϕ1) (rexp-alt ϕ2)
| rexp-of-alt n (FExists φ) = Pr (Inter (rexp-alt (n + 1) φ) (ENC (n + 1) (FOV φ)))
| rexp-of-alt n (FEXISTS φ) = Pr (Inter (rexp-alt (n + 1) φ) (ENC (n + 1) (FOV φ)))

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

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

definition lang_M2L-rexp-of :: wf-formula n ϕ ⇒ lang_M2L n ϕ ⇒ lang n (rexp of n
\( \varphi \) – \{\[]\}
\[ (\text{is} - \implies - = \text{?L n} \varphi) \]

**lemma** \text{wf-rex-of}:  \text{wf-formula n} \varphi \implies \text{wf n (rexp-of n} \varphi) 
\begin{proof}

**lemma** \text{wf-rex-of-alt}:  \text{wf-formula n} \varphi \implies \text{wf n (rexp-of-alt n} \varphi) 
\begin{proof}

**lemma** \text{wf-rex-of’}:  \text{wf-formula n} \varphi \implies \text{wf n (rexp-of’ n} \varphi) 
\begin{proof}

**lemma** \text{wf-rex-of-alt’}:  \text{wf-formula n} \varphi \implies \text{wf n (rexp-of-alt’ n} \varphi) 
\begin{proof}

**lemma** \text{wf-rex-of”}:  \text{wf-formula n} \varphi \implies \text{wf n (rexp-of” n} \varphi) 
\begin{proof}

**lemma** \text{ENC-Not}:  \text{ENC n (FOV (FNot} \varphi)) = \text{ENC n (FOV} \varphi) 
\begin{proof}

**lemma** \text{ENC-And}:  
\text{wf-formula n (FAnd} \varphi \psi) \implies \text{lang n (ENC n (FOV} (\text{FAnd} \varphi \psi))) - \{[]\} \subseteq \text{lang n (ENC n (FOV} \varphi)) \cap \text{lang n (ENC n (FOV} \psi)) - \{[]\} 
\begin{proof}

**lemma** \text{ENC-Or}:  
\text{wf-formula n (For} \varphi \psi) \implies \text{lang n (ENC n (FOV} (\text{For} \varphi \psi))) - \{[]\} \subseteq \text{lang n (ENC n (FOV} \varphi)) \cap \text{lang n (ENC n (FOV} \psi)) - \{[]\} 
\begin{proof}

**lemma** \text{project-enc}:  \text{map} \pi (\text{enc} (w, x \neq I)) = \text{enc} (w, I) 
\begin{proof}

**lemma** \text{list-list-eqI}:
\begin{proof}

**lemma** \text{project-enc-extend}:
\begin{proof}

**lemma** \text{ENC-Exists}:
\begin{proof}

58
\begin{proof}

\textbf{lemma ENC-EXISTS:}

\[ \text{wf-formula } n \ (FEXISTS \ \varphi) \implies \text{lang } n \ (ENC \ n \ (FOV \ (FEXISTS \ \varphi))) - \{[]\} \]
\[ = \text{map } \pi \ ' \ \text{lang } (Suc \ n) \ (ENC \ (Suc \ n) \ (FOV \ \varphi)) - \{[]\} \]
\end{proof}

\begin{proof}

\textbf{lemma map-project-empty: map } \pi \ ' \ A - \{[]\} = \text{map } \pi \ ' \ (A - \{[]\})
\end{proof}

\begin{proof}

\textbf{lemma lang$_{M2L}$-rexp-of-rexp-of':}

\[ \text{wf-formula } n \ \varphi \implies \text{lang } n \ (rexp-of \ n \ \varphi) - \{[]\} = \text{lang } n \ (rexp-of' \ n \ \varphi) - \{[]\} \]
\end{proof}

\begin{proof}

\textbf{lemma Int-Diff-both:} \ A \cap B - C = (A - C) \cap (B - C)
\end{proof}

\begin{proof}

\textbf{lemma map-project-Int-ENC:}

\[ \text{assumes } 0 \notin X \ X \subseteq \{0 ..< n + 1\} \ Z \subseteq \text{lists } ((\text{set } \sigma \ \Sigma) \ (n + 1)) \]
\[ \text{shows } \text{map } \pi \ ' \ (Z \cap \text{lang } (n + 1) \ (ENC \ (n + 1) \ X) - \{[]\}) = \]
\[ \text{map } \pi \ ' \ Z \cap \text{lang } n \ (ENC \ n \ ((\lambda x. \ x - 1) \ ' X)) - \{[]\} \]
\end{proof}

\begin{proof}

\textbf{lemma map-project-ENC:}

\[ \text{assumes } X \subseteq \{0 ..< n + 1\} \ Z \subseteq \text{lists } ((\text{set } \sigma \ \Sigma) \ (n + 1)) \]
\[ \text{shows } \text{map } \pi \ ' \ (Z \cap \text{lang } (n + 1) \ (ENC \ (n + 1) \ X) - \{[]\}) = \]
\[ (\text{if } 0 \in X \]
\[ \text{then map } \pi \ ' \ (Z \cap \text{lang } (n + 1) \ (ENC \ (n + 1) \ \{0\})) \cap \text{lang } n \ (ENC \ n \ ((\lambda x. \ x - 1) \ ' X - \{0\})) - \{[]\}) \]
\[ \text{else map } \pi \ ' \ Z \cap \text{lang } n \ (ENC \ n \ ((\lambda x. \ x - 1) \ ' X - \{0\})) - \{[]\}) \]
\[ (\text{is } ?L = (\text{if } - \text{ then } ?R1 \text{ else } ?R2)) \]
\end{proof}

\begin{proof}

\textbf{abbreviation } \Sigma \equiv \text{project.lang } (\text{set } \sigma \ \Sigma) \ \pi
\end{proof}

\begin{proof}

\textbf{lemma lang$_{M2L}$-rexp-of'-rexp-of''':}

\[ \text{wf-formula } n \ \varphi \implies \text{lang } n \ (rexp-of' \ n \ \varphi) - \{[]\} = \text{lang } n \ (rexp-of'' \ n \ \varphi) - \{[]\} \]
\end{proof}

\begin{proof}

\textbf{theorem lang$_{M2L}$-rexp-of':} \text{wf-formula } n \ \varphi \implies \text{lang$_{M2L}$ } n \ \varphi = \text{lang } n \ (rexp-of' \ n \ \varphi) - \{[]\}
\end{proof}

59
\textbf{theorem} $\text{lang}_{M^{2L}} \cdot \text{rexp-of}'' \colon \text{wf-formula} \ n \ \varphi \implies \text{lang}_{M^{2L}} \ n \ \varphi = \text{lang} \ n \ (\text{rexp-of}'' \ n \ \varphi) - \{[]\}$

\textit{(proof)}

\section{Normalization of M2L Formulas}

\textbf{fun} $\text{nNot}$ where
\begin{enumerate}
  \item $\text{nNot} \ (\text{FNot} \ \varphi) = \varphi$
  \item $\text{nNot} \ (\text{FAnd} \ \varphi_1 \ \varphi_2) = \text{FOr} \ (\text{nNot} \ \varphi_1) \ (\text{nNot} \ \varphi_2)$
  \item $\text{nNot} \ (\text{FOr} \ \varphi_1 \ \varphi_2) = \text{FAnd} \ (\text{nNot} \ \varphi_1) \ (\text{nNot} \ \varphi_2)$
  \item $\text{nNot} \ \varphi = \text{FNot} \ \varphi$
\end{enumerate}

\textbf{primrec} $\text{norm}$ where
\begin{enumerate}
  \item $\text{norm} \ (\text{FQ} \ a \ m) = \text{FQ} \ a \ m$
  \item $\text{norm} \ (\text{FLess} \ m \ n) = \text{FLess} \ m \ n$
  \item $\text{norm} \ (\text{FIn} \ m \ M) = \text{FIn} \ m \ M$
  \item $\text{norm} \ (\text{FOr} \ \varphi \ \psi) = \text{FOr} \ (\text{norm} \ \varphi) \ (\text{norm} \ \psi)$
  \item $\text{norm} \ (\text{FAnd} \ \varphi \ \psi) = \text{FAnd} \ (\text{norm} \ \varphi) \ (\text{norm} \ \psi)$
  \item $\text{norm} \ (\text{FNot} \ \varphi) = \text{nNot} \ (\text{norm} \ \varphi)$
  \item $\text{norm} \ (\text{FExists} \ \varphi) = \text{FExists} \ (\text{norm} \ \varphi)$
  \item $\text{norm} \ (\text{FEXISTS} \ \varphi) = \text{FEXISTS} \ (\text{norm} \ \varphi)$
\end{enumerate}

\textbf{context} formula
\textbf{begin}

\textbf{lemma} satisfies-nNot[simp]: satisfies $\langle w, I \rangle$ $(\text{nNot} \ \varphi) = \text{satisfies} \ (w, I) \ (\text{FNot} \ \varphi)$
\textit{(proof)}

\textbf{lemma} FOV-nNot[simp]: FOV $(\text{nNot} \ \varphi) = \text{FOV} \ (\text{FNot} \ \varphi)$
\textit{(proof)}

\textbf{lemma} SOV-nNot[simp]: SOV $(\text{nNot} \ \varphi) = \text{SOV} \ (\text{FNot} \ \varphi)$
\textit{(proof)}

\textbf{lemma} pre-wf-formula-nNot[simp]: pre-wf-formula $\ n \ (\text{nNot} \ \varphi) = \text{pre-wf-formula} \ n \ (\text{FNot} \ \varphi)$
\textit{(proof)}

\textbf{lemma} FOV-norm[simp]: FOV $(\text{norm} \ \varphi) = \text{FOV} \ \varphi$
\textit{(proof)}

\textbf{lemma} SOV-norm[simp]: SOV $(\text{norm} \ \varphi) = \text{SOV} \ \varphi$
\textit{(proof)}

\textbf{lemma} pre-wf-formula-norm[simp]: pre-wf-formula $\ n \ (\text{norm} \ \varphi) = \text{pre-wf-formula} \ n \ (\text{norm} \ \varphi)$
\textit{(proof)}
\[ n \varphi \]

\[ \text{proof} \]

**lemma** satisfies-norm\([\text{simp}]:\) satisfies \((w, I) (\text{norm } \varphi) = \text{satisfies } (w, I) \varphi \]

\[ \text{proof} \]

**lemma** lang\(_{M2L}\)-norm\([\text{simp}]:\) lang\(_{M2L}\) \(n (\text{norm } \varphi) = \text{lang}_{M2L} n \varphi \]

\[ \text{proof} \]

end

12 Deciding Equivalence of M2L Formulas

**permanent-interpretation** embed set \(o \sigma \Sigma \text{wf-atom } \Sigma \pi \text{lookup } \varepsilon \Sigma \)

**for** \(\Sigma::'a :: \text{linorder list} \)

**defining**

- \(\mathcal{D} = \text{embed.lderiv lookup } (\varepsilon \Sigma) \)
- \(\text{Co}\mathcal{D} = \text{embed.lderiv-dual lookup } (\varepsilon \Sigma) \)

\[ \text{proof} \]

**lemma** enum-not-empty\([\text{simp}]:\) Enum.enum \(\neq [] \) (is \(?\text{enum} \neq []\))

\[ \text{proof} \]

**permanent-interpretation** \(\Phi::\text{formula } \Sigma::'a :: \{\text{enum, linorder}\} \text{list} \)

**defining**

- \(\text{pre-wf-formula } = \Phi.\text{pre-wf-formula} \)
- \(\text{wf-formula } = \Phi.\text{wf-formula} \)
- \(\text{rexp-of } = \Phi.\text{rexp-of} \)
- \(\text{rexp-of-alt } = \Phi.\text{rexp-of-alt} \)
- \(\text{rexp-of-alt}' = \Phi.\text{rexp-of-alt}' \)
- \(\text{rexp-of}'' = \Phi.\text{rexp-of}'' \)
- \(\text{valid-ENC } = \Phi.\text{valid-ENC} \)
- \(\text{ENC } = \Phi.\text{ENC} \)
- \(\text{dec-interp } = \Phi.\text{dec-interp} \)

\[ \text{proof} \]

**lemma** lang-Plus-Zero: lang \(\Sigma\) \(n (\text{Plus } r \text{ One}) = \text{lang } \Sigma\) \(n (\text{Plus } s \text{ One}) \iff \text{lang } \Sigma\) \(n r - \{[]\} = \text{lang } \Sigma\) \(n s - \{[]\} \)

\[ \text{proof} \]

**lemmas** lang\(_{M2L}\)-rexp-of-norm = trans[OF sym][OF \(\Phi.\text{lang}_{M2L}\)-norm] \(\Phi.\text{lang}_{M2L}\)-rexp-of]

**lemmas** lang\(_{M2L}\)-rexp-of-'-norm = trans[OF sym][OF \(\Phi.\text{lang}_{M2L}\)-norm] \(\Phi.\text{lang}_{M2L}\)-rexp-of']

**lemmas** lang\(_{M2L}\)-rexp-of''-norm = trans[OF sym][OF \(\Phi.\text{lang}_{M2L}\)-norm] \(\Phi.\text{lang}_{M2L}\)-rexp-of'']

\[ \text{ML} \]
permanent-interpretation $D$: rexp-DFA $\sigma$ wf-atom $\Sigma$ $\pi$ lookup $\lambda x. \langle \text{inorm } x \rangle$

$\lambda a. \langle \mathcal{D} \Sigma a r \rangle$ final alphabet.wf $\langle$ wf-atom $\Sigma \rangle$ n $\text{pnorm lang } \Sigma$ n n

for $\Sigma :: \ 'a :: \ \text{linorder list and } n :: \ \text{nat}$

defining

test = rexp-DFA.test (final :: \ 'a atom rexp $\Rightarrow$ bool)

and step = rexp-DFA.step ($\sigma$ $\Sigma$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) $\text{pnorm n}$

and closure = rexp-DFA.closure ($\sigma$ $\Sigma$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) final $\text{pnorm n}$

and check-eqvRE = rexp-DFA.check-eqv ($\sigma$ $\Sigma$) ($\lambda x. \langle \text{pnorm (inorm x)} \rangle$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) final $\text{pnorm n}$

and test-invariant = rexp-DFA.test-invariant (final :: \ 'a atom rexp $\Rightarrow$ bool) ::

$\langle \langle \lambda a. \langle \text{linorder list} \rangle \times -$ list $\times -$ $\Rightarrow$ bool

and step-invariant = rexp-DFA.step-invariant ($\sigma$ $\Sigma$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) $\text{pnorm n}$

and closure-invariant = rexp-DFA.closure-invariant ($\sigma$ $\Sigma$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) final $\text{pnorm n}$

and counterexampleRE = rexp-DFA.counterexample ($\sigma$ $\Sigma$) ($\lambda x. \langle \text{pnorm (inorm x)} \rangle$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) final $\text{pnorm n}$

and reachable = rexp-DFAreachable ($\sigma$ $\Sigma$) ($\lambda x. \langle \text{pnorm (inorm x)} \rangle$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) $\text{pnorm n}$

and automaton = rexp-DFA.automaton ($\sigma$ $\Sigma$) ($\lambda x. \langle \text{pnorm (inorm x)} \rangle$) ($\lambda a. \langle \mathcal{D} \Sigma a r \rangle$) $\text{pnorm n}$

(proof)

definition check-eqv where

check-eqv $n \varphi \psi \longleftarrow$ wf-formula $n$ ($\text{FOV } \varphi \psi$) $\land$

$\text{slow.check-eqvRE Enum.enum } n \ (\text{Plus } (\text{rexp-of" } n \ (\text{norm } \varphi)) \text{ One}) \ (\text{Plus } (\text{rexp-of" } n \ (\text{norm } \psi)) \text{ One})$

definition counterexample where

counterexample $n \varphi \psi =$

$\langle \text{map-option } (\lambda w. \text{dec-interp } n \ (\text{FOV } \varphi \psi)) w \rangle$

$\text{slow.counterexampleRE Enum.enum } n \ (\text{Plus } (\text{rexp-of" } n \ (\text{norm } \varphi)) \text{ One}) \ (\text{Plus } (\text{rexp-of" } n \ (\text{norm } \psi)) \text{ One}))$

lemma soundness: slow.check-eqv $n \varphi \psi \Rightarrow \Phi.\text{lang}_{M2L} n \varphi = \Phi.\text{lang}_{M2L} n \psi$

(proof)

lemma completeness:

assumes $\Phi.\text{lang}_{M2L} n \varphi = \Phi.\text{lang}_{M2L} n \psi$ $\text{wf-formula } n$ ($\text{FOV } \varphi \psi$)

shows slow.check-eqv $n \varphi \psi$

(proof)

$\langle \text{ML} \rangle$

permanent-interpretation $D$: rexp-DFA-no-post $\sigma$ wf-atom $\Sigma$ $\pi$ lookup $\lambda x. \langle \text{inorm } x \rangle$

$\lambda a. \text{pnorm } \langle \mathcal{D} \Sigma a r \rangle$ final alphabet.wf $\langle$ wf-atom $\Sigma \rangle$ n $\text{lang } \Sigma$ n n

for $\Sigma :: \ 'a :: \ \text{linorder list and } n :: \ \text{nat}$

defining
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 (D Σ a r)) final id n
and reachable = rexp-DA.reachable (σ Σ) (λx. pnorm (inorm x)) (λa r. pnorm (D Σ a r)) id n
and automaton = rexp-DA.automaton (σ Σ) (λx. pnorm (inorm x)) (λa r. pnorm (D Σ a r)) id n
(proof)

definition check-eqv where
check-eqv n φ ψ =⇒wf-formula n (FOR φ ψ) ∧
    fast.check-eqvRE Enum.enum n (Plus (rexp-of’’ n (norm φ)) One) (Plus (rexp-of’’ n (norm ψ)) One)

definition counterexample where
counterexample n φ ψ =
    map-option (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 =⇒ Φ.Ilang M2L n φ = Φ.Ilang M2L n ψ
(proof)

⟨ML⟩

permanent-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 (CoD Σ a r) final-dual
alphabet wf-dual (rexp-DA Σ) n lang-dual Σ n n
for Σ :: 'a :: linorder list and n :: nat
defining
    test = rexp-DA.test (final-dual :: 'a atom rexp dual ⇒ bool)
    and step = rexp-DA.step (σ Σ) (λa r. pnorm-dual (CoD Σ a r)) id n
    and closure = rexp-DA.closure (σ Σ) (λa r. pnorm-dual (CoD 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 (CoD Σ a r)) final-dual id n
    and test-invariant = rexp-DA.test-invariant (final-dual :: 'a atom rexp-dual ⇒ bool) ::
        ((’a × bool list) list × -) list × - ⇒ bool

63
and step-invariant = rexp-DA.step-invariant (σ Σ) (λ a r. pnorm-dual (CoΣ a r)) id n
and closure-invariant = rexp-DA.closure-invariant (σ Σ) (λ a r. pnorm-dual (CoΣ 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 (CoΣ a r)) final-dual id n
and reachable = rexp-DAreachable (σ Σ) (λ x. pnorm-dual (rexp-dual-of (inorm x))) (λ a r. pnorm-dual (CoΣ a r)) final-dual id n
and automaton = rexp-DA.automaton (σ Σ) (λ x. pnorm-dual (rexp-dual-of (inorm x))) (λ a r. pnorm-dual (CoΣ a r)) final-dual id n

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 M L n ϕ = φ. lang M L 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
  ⟨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:

65
∀ 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 ≠ 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 Ṽ: 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 \textit{sdrop-sconst}: \textit{sdrop} \ n \ s = \textit{sconst} \ x \implies n \leq m \implies s \ !! m = x
\langle \text{proof} \rangle

lemma \textit{fin-cut-same-tl}:
  \textbf{assumes} \exists \ n. \ \textit{sdrop} \ n \ s = \textit{sconst} \ x
  \textbf{shows} \ \textit{fin-cut-same} (\pi \ x) (\textit{map} \ \pi \ (\textit{cut-same} \ x \ s)) = \textit{cut-same} (\pi \ x) (\textit{smap} \ \pi \ s)
\langle \text{proof} \rangle

lemma \textit{tl-enc}[simp]:
  \textbf{assumes} \ \forall \ x \in \textit{set} \ (x \neq I). \ \textit{case} \ x \ \textit{of} \ \textit{Inr} \ P \ \Rightarrow \ \textit{finite} \ P \ | - \Rightarrow \text{True}
  \textbf{shows} \ \textit{SAMEQUOT} (\textit{any}, \\textit{replicate} (\textit{length} \ I) \ \textit{False}) (\textit{map} \ \pi \ ' (\textit{enc} \ (w, x \neq I)))
\langle \text{proof} \rangle

lemma \textit{encD}:
  \[ v \in \textit{enc} (w, I); (\forall x \in \textit{set} \ I. \ \textit{case} \ x \ \textit{of} \ \textit{Inr} \ P \ \Rightarrow \ \textit{finite} \ P \ | - \Rightarrow \text{True}) \] \implies
  v = \textit{map} (\textit{split} (\textit{enc-atom} \ I)) (\textit{zip} [0 \ldots< \textit{length} \ v] (\textit{stake} (\textit{length} \ v) (w \ \textit{\&- sconst} \ \textit{any})))
\langle \text{proof} \rangle

lemma \textit{enc-Inl}: \[ x \in \textit{enc} (w, I); (\forall x \in \textit{set} \ I. \ \textit{case} \ x \ \textit{of} \ \textit{Inr} \ P \ \Rightarrow \ \textit{finite} \ P \ | - \Rightarrow \text{True}); \]
  \[ m < \textit{length} \ I; \ \textit{I} ! m = \textit{Inl} \ p \] \implies
  p < \textit{length} \ x \land \textit{snd} (x ! \ p) ! m
\langle \text{proof} \rangle

lemma \textit{enc-Inr}: \textbf{assumes} \ x \in \textit{enc} (w, I) \ \forall x \in \textit{set} \ I. \ \textit{case} \ x \ \textit{of} \ \textit{Inr} \ P \ \Rightarrow \ \textit{finite} \ P
  \ | - \Rightarrow \text{True}
  M < \textit{length} \ I \ \textit{I} ! M = \textit{Inr} \ P
  \textbf{shows} \ p \in P \iff p < \textit{length} \ x \land \textit{snd} (x ! \ p) ! M
\langle \text{proof} \rangle

lemma \textit{enc-length}:
  \textbf{assumes} \ \textit{enc} \ (w, I) = \textit{enc} \ (w', I')
  \textbf{shows} \ \textit{length} \ I = \textit{length} \ I'
\langle \text{proof} \rangle

lemma \textit{enc-stream-enc}:
  \[(\forall x \in \textit{set} \ I. \ \textit{case} \ x \ \textit{of} \ \textit{Inr} \ P \ \Rightarrow \ \textit{finite} \ P \ | - \Rightarrow \text{True}); \]
  \[(\forall x \in \textit{set} \ I'. \ \textit{case} \ x \ \textit{of} \ \textit{Inr} \ P \ \Rightarrow \ \textit{finite} \ P \ | - \Rightarrow \text{True}); \]
  \[ \textit{enc} \ (w, I) = \textit{enc} \ (w', I') \] \implies \textit{stream-enc} (w, I) = \textit{stream-enc} (w', I')
\langle \text{proof} \rangle

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

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

fun satisfies :: 'a interop ⇒ 'a formula ⇒ bool (infix |= 50) where
(w, I) |= FQ a m = ((case I ! m of Inl p ⇒ if p < length w then w ! p else any) = a)
| (w, I) |= FLess m1 m2 = ((case I ! m1 of Inl p ⇒ p) < (case I ! m2 of Inl p ⇒ p))
| (w, I) |= FLn 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. (w, Inl p # I) |= ϕ)
| (w, I) |= (FEXISTS ϕ) = (∃ P. finite P ∧ (w, Inr P # I) |= ϕ)

definition langW_S1S :: nat ⇒ 'a formula ⇒ ('a × bool list) list set where
langW_S1S n ϕ = \{ \enc (w, I) | w I . length I = n ∧ wf-interp-for-formula (w, I) ϕ \ ∧ \ (w, I) |= ϕ \}

lemma encD-ex: [x ∈ \enc (w, I); (∀ x ∈ set I. case x of Inr p ⇒ finite P | -⇒ True)] \implies
∃ n. x = map (split (enc-atom I)) (zip [\0..<n] (stake n (w @– sconst any))))
(\proof)

lemma enc-set-σ: [x ∈ \enc (w, I); (∀ x ∈ set I. case x of Inr p ⇒ finite P | -⇒ True);
 length I = n; a ∈ set x; set w ⊆ set Σ] \implies a ∈ set (σ Σ n)
(\proof)

definition positions-in-row s i =
Option.these (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: \exists! p. snd (s !! p) ! i \implies
the-elem (positions-in-row s i) = (THE p. snd (s !! p) ! i)
(\proof)

lemma positions-in-row-nth: \exists! p. snd (s !! p) ! i \implies
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 (λi. 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: \[ \forall i ∈ FO; i < n \] \implies \exists p. stream-dec n FO s ! i = Inl p

**proof**

**lemma** stream-dec-not-Inr: \[ \forall i \notin FO; i < n \] \implies False

**proof**

**lemma** stream-dec-Inr: \[ \forall i \notin FO; i < n \] \implies \exists P. stream-dec n FO s ! i = Inr P

**proof**

**lemma** stream-dec-not-Inl: \[ \forall i \notin FO; i < n \] \implies False

**proof**

**lemma** Inr-dec-finite: \[ \forall i < n. finite \{ p. snd (s !! p) ! i \}; Inr P \in set (stream-dec n FO s) \] \implies finite P

**proof**

**lemma** enc-atom-dec: \[ \forall p. length (snd (s !! p)) = n; \forall i ∈ FO. i < n \implies (\exists !p. snd (s !! p) ! i); a = fst (s !! p) \] \implies 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:

\[ \exists n. sdrop n (smap fst s) = sconst any; \]

\[ \forall i ∈ FO. i < n \implies (\exists !p. snd (s !! p) ! i) \]

\[ stream-enc (dec-word s, stream-dec n FO s) = s \]

**proof**

**lemma** stream-enc-unique:

\[ i < length I \implies \exists p. I ! i = Inl p \implies \exists !p. snd (stream-enc (w, I) !! p) ! i \]

**proof**

**lemma** stream-dec-enc-Inl:

\[ \forall i ∈ FO. i < n; length I = n \] \implies p = p'

**proof**

69
lemma stream-dec-enc-Inr:
\[ \text{stream-dec } n \text{ FO } (\text{stream-enc } (w, I))! \; i = \text{Inr } P' ; I! \; i = \text{Inr } P ; i \notin \text{FO} ; i < n; \text{length } I = n \implies P = P' \]
(\text{proof})

lemma Collect-snth:
\{ p. \; P ((x #\# s) !! p)\} \subseteq \{0\} \cup \text{Suc} ' \{ p. \; P (s !! p)\}
(\text{proof})

lemma finite-True-in-row:
\forall i < n. \text{finite } \{ p. \; \text{snd} ((w @= \text{scost} \text{(any, replicate n False))} ) !! p) ! i\}
(\text{proof})

lemma lang-ENC:
\text{assumes } \text{FO} \subseteq \{0 ..< n\} \; \text{SO} \subseteq \{0 ..< n\} - \text{FO}
\text{shows } \text{lang } n \; (\text{ENC } n \; \text{FO}) = \bigcup \{ \text{enc} (w, I) | \; w I . \text{length } I = n \land \text{wf-interp } w I \land \begin{align*}
(\forall i \in \text{FO}. \text{case } I! \; i \; \text{of Inl - } \Rightarrow \text{True} | \; \text{Inr } \Rightarrow \text{False} ) \land \\
(\forall i \in \text{SO}. \text{case } I! \; i \; \text{of Inl - } \Rightarrow \text{False} | \; \text{Inr } \Rightarrow \text{True})
\end{align*}
\text{is } \text{?L = ?R}
(\text{proof})

lemma lang-ENC-formula:
\text{assumes } \text{wf-formula } n \; \varphi
\text{shows } \text{lang } n \; (\text{ENC } n \; (\text{FOV } \varphi)) = \bigcup \{ \text{enc} (w, I) | \; w I . \text{length } I = n \land \text{wf-interp-for-formula } (w, I) \; \varphi\}
(\text{proof})

13.2 Welldefinedness of enc wrt. Models

lemma wf-interp-for-formula-FExists:
\text{assumes } \text{wf-formula } (\text{length } I) \; (\text{FExists } \varphi)\implies \text{wf-interp-for-formula } (w, I) \; (\text{FExists } \varphi) \iff (\forall p. \; \text{wf-interp-for-formula } (w, \text{Inl } p \# I) \; \varphi)
(\text{proof})

lemma wf-interp-for-formula-any-Inl:
\forall p. \; \text{wf-interp-for-formula } (w, \text{Inl } p \# I) \; \varphi \implies \text{wf-interp-for-formula } (w, \text{Inl } p \# I) \; \varphi
(\text{proof})

lemma wf-interp-for-formula-FEXISTS:
\text{assumes } \text{wf-formula } (\text{length } I) \; (\text{FEXISTS } \varphi)\implies \text{wf-interp-for-formula } (w, I) \; (\text{FEXISTS } \varphi) \iff (\forall P. \; \text{finite } P \implies \text{wf-interp-for-formula } (w, \text{Inr } P \# I) \; \varphi)
(\text{proof})

lemma wf-interp-for-formula-any-Inr:
\forall P. \; \text{finite } P \implies \text{wf-interp-for-formula } (w, \text{Inr } P \# I) \; \varphi \implies \text{wf-interp-for-formula } (w, \text{Inr } P \# I) \; \varphi
(\text{proof})
(proof)

lemma wf-interp-for-formula-FOr:
\[ \text{wf-interp-for-formula} \left( w, I \right) (\text{FOr} \ \varphi_1 \ \varphi_2) = \]
\[ \left( \text{wf-interp-for-formula} \left( w, I \right) \varphi_1 \land \text{wf-interp-for-formula} \left( w, I \right) \varphi_2 \right) \]
(\proof)

lemma wf-interp-for-formula-FAnd:
\[ \text{wf-interp-for-formula} \left( w, I \right) (\text{FAnd} \ \varphi_1 \ \varphi_2) = \]
\[ \left( \text{wf-interp-for-formula} \left( w, I \right) \varphi_1 \land \text{wf-interp-for-formula} \left( w, I \right) \varphi_2 \right) \]
(\proof)

lemma enc-wf-interp:
\[ [\text{wf-formula} \left( \text{length} \ I \right) \varphi; \text{wf-interp-for-formula} \left( w, I \right) \varphi; x \in \text{enc} \left( w, I \right)] \implies \]
\[ \text{wf-interp-for-formula} \left( \text{dec-word} \ (x \cdot \text{- const} \ (\text{any}, \text{replicate} \ (\text{length} \ I) \ False)), \right) \]
\[ \text{stream-dec} \ (\text{length} \ I) \ (\text{FOV} \ \varphi) \ (x \cdot \text{- const} \ (\text{any}, \text{replicate} \ (\text{length} \ I) \ False))) \]
\[ \varphi \]
(\proof)

lemma enc-atom-welldef:
\[ \forall \ x \ a \ \text{enc-atom} \ I \ x \ a = \text{enc-atom} \ I' \ x \ a \implies m < \text{length} \ I \implies \]
\[ \text{case} \ (I \ ! \ m, I' ! m) \ of \ (\text{Inl} \ p, \text{Inl} \ q) \Rightarrow p = q \ | \ (\text{Inr} \ P, \text{Inr} \ Q) \Rightarrow P = Q \ | \ - \Rightarrow \text{True} \]
(\proof)

lemma stream-enc-welldef:
\[ [\text{stream-enc} \ (w, I) = \text{stream-enc} \ (w', I'); \text{wf-formula} \ (\text{length} \ I) \varphi; \]
\[ \text{wf-interp-for-formula} \left( w, I \right) \varphi; \text{wf-interp-for-formula} \left( w', I' \right) \varphi] \implies \]
\[ (w, I) \models \varphi \iff (w', I') \models \varphi \]
(\proof)

lemma lang_{WS1S}-FOr:
\[ \text{assumes} \ \text{wf-formula} \ n \ (\text{FOr} \ \varphi_1 \ \varphi_2) \]
\[ \text{shows} \ \text{lang}_{WS1S} \ n \ (\text{FOr} \ \varphi_1 \ \varphi_2) \subseteq \]
\[ \text{lang}_{WS1S} \ n \ \varphi_1 \cup \text{lang}_{WS1S} \ n \ \varphi_2 \cap \bigcup \{ \text{enc} \ (w, I) \mid w. \text{length} \ I = n \land \]
\[ \text{wf-interp-for-formula} \ (w, I) (\text{FOr} \ \varphi_1 \ \varphi_2) \} \]
\[ \text{is} - \subseteq \ \{ ?L1 \cup ?L2 \} \cap \ ?ENC \}
(\proof)

lemma lang_{WS1S}-FAnd:
\[ \text{assumes} \ \text{wf-formula} \ n \ (\text{FAnd} \ \varphi_1 \ \varphi_2) \]
\[ \text{shows} \ \text{lang}_{WS1S} \ n \ (\text{FAnd} \ \varphi_1 \ \varphi_2) \subseteq \]
\[ \text{lang}_{WS1S} \ n \ \varphi_1 \cap \text{lang}_{WS1S} \ n \ \varphi_2 \cap \bigcup \{ \text{enc} \ (w, I) \mid w. \text{length} \ I = n \land \]
\[ \text{wf-interp-for-formula} \ (w, I) (\text{FAnd} \ \varphi_1 \ \varphi_2) \} \]
(\proof)

13.3 From WS1S to Regular expressions

fun rexp-of :: \text{nat} \Rightarrow \text{'a \ formula} \Rightarrow (\text{'a \ atom}) \ \text{rexp} \ \text{where}
\[\text{rexp-of } n \ (FQ \ a \ m) =
\text{Inter} \ (\text{TIMES} \ [\text{rexp.Not Zero, Atom} \ (AQ \ m \ a), \ \text{rexp.Not Zero}])
(\text{ENC} \ n \ (\text{FOV} \ (FQ \ a \ m)))]

\[\text{rexp-of } n \ (F\text{Less} \ m1 \ m2) = (\text{if } m1 = m2 \ \text{then Zero else})
\text{Inter} \ (\text{TIMES} \ [\text{rexp.Not Zero, Atom} \ (\text{Arbitrary-Except} \ m1 \ \text{True}),
\text{rexp.Not Zero, Atom} \ (\text{Arbitrary-Except} \ m2 \ \text{True}),
\text{rexp.Not Zero}]) \ (\text{ENC} \ n \ (\text{FOV} \ (F\text{Less} \ m1 \ m2 :: \ 'a \ formula))))\]

\[\text{rexp-of } n \ (\text{Fln} \ m \ M) =
\text{Inter} \ (\text{TIMES} \ [\text{rexp.Not Zero, Atom} \ (\text{Arbitrary-Except}2 \ m \ M), \ \text{rexp.Not Zero}])
(\text{ENC} \ n \ (\text{FOV} \ (\text{Fln} \ m \ M :: \ 'a \ formula))))\]

\[\text{rexp-of } n \ (\text{FNot} \ \phi) = \text{Inter} \ (\text{rexp.Not} \ (\text{rexp-of } n \ \phi)) \ (\text{ENC} \ n \ (\text{FOV} \ (\text{FNot} \ \phi)))\]

\[\text{rexp-of } n \ (\text{FOr} \ \phi1 \ \phi2) = \text{Inter} \ (\text{Plus} \ (\text{rexp-of } n \ \phi1) \ (\text{rexp-of } n \ \phi2)) \ (\text{ENC} \ n \ (\text{FOV} \ (\text{FOr} \ \phi1 \ \phi2)))\]

\[\text{rexp-of } n \ (\text{FAnd} \ \phi1 \ \phi2) = \text{INTERSECT} \ [\text{rexp-of } n \ \phi1, \ \text{rexp-of } n \ \phi2, \ \text{ENC} \ n \ (\text{FOV} \ (\text{FAnd} \ \phi1 \ \phi2))]\]

\[\text{rexp-of } n \ (\text{FExists} \ \phi) = \text{samequot-exec} \ (\text{any, replicate } n \ \text{False}) \ (\text{Pr} \ (\text{rexp-of } (n + 1) \ \phi))\]

\[\text{rexp-of } n \ (\text{FEXISTS} \ \phi) = \text{samequot-exec} \ (\text{any, replicate } n \ \text{False}) \ (\text{Pr} \ (\text{rexp-of } (n + 1) \ \phi))\]

\textbf{func} \text{rexp-of-alt} :: \text{nat} \Rightarrow \ ('a \ \text{formula} \Rightarrow \ ('a \ \text{atom}) \ \text{rexp}) \ \text{where}
\text{rexp-of-alt} \ n \ (FQ \ a \ m) =
\text{TIMES} \ [\text{rexp.Not Zero, Atom} \ (AQ \ m \ a), \ \text{rexp.Not Zero}]\]

\[\text{rexp-of-alt} \ n \ (\text{FLess} \ m1 \ m2) = (\text{if } m1 = m2 \ \text{then Zero else})
\text{TIMES} \ [\text{rexp.Not Zero, Atom} \ (\text{Arbitrary-Except} \ m1 \ \text{True}),
\text{rexp.Not Zero, Atom} \ (\text{Arbitrary-Except} \ m2 \ \text{True}),
\text{rexp.Not Zero}]) \ (\text{ENC} \ n \ (\text{FOV} \ (\text{FLess} \ m1 \ m2 :: \ 'a \ formula))))\]

\textbf{definition} \text{rexp-of'} \ n \ \phi = \text{Inter} \ (\text{rexp-of-alt} \ n \ \phi) \ (\text{ENC} \ n \ (\text{FOV} \ \phi))\]

\textbf{func} \text{rexp-of-alt'} :: \text{nat} \Rightarrow \ ('a \ \text{formula} \Rightarrow \ ('a \ \text{atom}) \ \text{rexp}) \ \text{where}
\text{rexp-of-alt'} \ n \ (FQ \ a \ m) = \text{TIMES} \ [\text{Full}, \ \text{Atom} \ (AQ \ m \ a), \ \text{Full}]\]

\[\text{rexp-of-alt'} \ n \ (\text{FLess} \ m1 \ m2) = (\text{if } m1 = m2 \ \text{then Zero else})
\text{TIMES} \ [\text{Full}, \ \text{Atom} \ (\text{Arbitrary-Except} \ m1 \ \text{True}), \ \text{Full}, \ \text{Atom} \ (\text{Arbitrary-Except} \ m2 \ \text{True}), \ \text{Full})]\]

\[\text{rexp-of-alt'} \ n \ (\text{Fln} \ m \ M) = \text{TIMES} \ [\text{Full}, \ \text{Atom} \ (\text{Arbitrary-Except}2 \ m \ M), \ \text{Full}]\]

\[\text{rexp-of-alt'} \ n \ (\text{FNot} \ \phi) = \text{rexp.Not} \ (\text{rexp-of-alt'} \ n \ \phi)\]

\[\text{rexp-of-alt'} \ n \ (\text{FOr} \ \phi1 \ \phi2) = \text{Plus} \ (\text{rexp-of-alt'} \ n \ \phi1) \ (\text{rexp-of-alt'} \ n \ \phi2)\]

\[\text{rexp-of-alt'} \ n \ (\text{FAnd} \ \phi1 \ \phi2) = \text{Inter} \ (\text{rexp-of-alt'} \ n \ \phi1) \ (\text{rexp-of-alt'} \ n \ \phi2)\]

\[\text{rexp-of-alt'} \ n \ (\text{FExists} \ \phi) = \text{samequot-exec} \ (\text{any, replicate } n \ \text{False}) \ (\text{Pr} \ (\text{rexp-of-alt'} \ n + 1) \ \phi))\]
\[(\text{rexp-of-alt'} (n + 1) \varphi) (\text{ENC} (n + 1) \{\emptyset\}))\]
\[| \text{rexp-of-alt'} n (\text{FEXISTS} \varphi) = \text{samequot-exec} (\text{any}, \text{replicate} n \text{False}) (\text{Pr} (\text{rexp-of-alt'} (n + 1) \varphi))\]

**Definition** \(\text{rexp-of'} n \varphi = \text{Inter} (\text{rexp-of-alt'} n \varphi) (\text{ENC} n (\text{FOV} \varphi))\)

**Lemma** \(\text{enc-eqI}:\)
- **Assumes** \(x \in \text{enc} (w, I) x \in \text{enc} (w', I') \text{wf-interp-for-formula} (w, I) \varphi \text{wf-interp-for-formula} (w', I') \varphi \text{length} I = \text{length} I'\)
- **Shows** \(\text{enc} (w, I) = \text{enc} (w', I')\)

**Proof**

**Lemma** \(\text{enc-eq-welldef}:\)
- \(\llbracket \text{enc} (w, I) = \text{enc} (w', I') ; \text{wf-formula} (\text{length} I) \varphi ; \text{wf-interp-for-formula} (w, I) \varphi ; \text{wf-interp-for-formula} (w', I') \varphi \rrbracket \implies (w, I) \models \varphi \iff (w', I') \models \varphi\)

**Proof**

**Lemma** \(\text{wf-rexp-of}: \text{wf-formula} n \varphi \implies \text{wf} n (\text{rexp-of} n \varphi)\)

**Proof**

**Theorem** \(\text{lang} W S 1 S -\text{rexp-of}: \text{wf-formula} n \varphi \implies \text{lang} W S 1 S n \varphi = \text{lang} n (\text{rexp-of} n \varphi)\)
- \((\text{is} - \implies \_ = ?L n \varphi)\)

**Proof**

**Lemma** \(\text{wf-rexp-of-alt}: \text{wf-formula} n \varphi \implies \text{wf} n (\text{rexp-of-alt} n \varphi)\)

**Proof**

**Lemma** \(\text{wf-rexp-of'}: \text{wf-formula} n \varphi \implies \text{wf} n (\text{rexp-of'} n \varphi)\)

**Proof**

**Lemma** \(\text{wf-rexp-of-alt'}: \text{wf-formula} n \varphi \implies \text{wf} n (\text{rexp-of-alt'} n \varphi)\)

**Proof**

**Lemma** \(\text{wf-rexp-of''}: \text{wf-formula} n \varphi \implies \text{wf} n (\text{rexp-of''} n \varphi)\)

**Proof**

**Lemma** \(\text{ENC-FNot}: \text{ENC} n (\text{FOV} (\text{FNot} \varphi)) = \text{ENC} n (\text{FOV} \varphi)\)

**Proof**

73
lemma ENC-FAnd:
\[ \text{wf-formula} n (\text{FAnd} \, \varphi \, \psi) \implies \text{lang} n (\text{ENC} n (\text{FOV} (\text{FAnd} \, \varphi \, \psi))) \subseteq \text{lang} n (\text{ENC} n (\text{FOV} \, \varphi)) \cap \text{lang} n (\text{ENC} n (\text{FOV} \, \psi)) \]
⟨proof⟩

lemma ENC-FOr:
\[ \text{wf-formula} n (\text{FOr} \, \varphi \, \psi) \implies \text{lang} n (\text{ENC} n (\text{FOV} (\text{FOr} \, \varphi \, \psi))) \subseteq \text{lang} n (\text{ENC} n (\text{FOV} \, \varphi)) \cap \text{lang} n (\text{ENC} n (\text{FOV} \, \psi)) \]
⟨proof⟩

lemma ENC-FExists:
\[ \text{wf-formula} n (\text{FExists} \, \varphi) \implies \text{lang} n (\text{ENC} n (\text{FOV} (\text{FExists} \, \varphi))) = \text{SAMEQUOT} (\text{any}, \text{replicate} n \text{False}) (\text{map} \pi \text{ lang} (\text{Suc} n) (\text{ENC} (\text{Suc} n) (\text{FOV} \, \varphi))) (\text{is} - \implies ?L = ?R) \]
⟨proof⟩

lemma ENC-FEXISTS:
\[ \text{wf-formula} n (\text{FEXISTS} \, \varphi) \implies \text{lang} n (\text{ENC} n (\text{FOV} (\text{FEXISTS} \, \varphi))) = \text{SAMEQUOT} (\text{any}, \text{replicate} n \text{False}) (\text{map} \pi \text{ lang} (\text{Suc} n) (\text{ENC} (\text{Suc} n) (\text{FOV} \, \varphi))) (\text{is} - \implies ?L = ?R) \]
⟨proof⟩

lemma lang_W S_1 S-rexp-of-rexp-of':
\[ \text{wf-formula} n \varphi \implies \text{lang} n (\text{rexp-of} n \varphi) = \text{lang} n (\text{rexp-of}' \, n \, \varphi) \]
⟨proof⟩

lemma SAMEQUOT-UN[simp]: SAMEQUOT x (\bigcup y \in A. B y) = (\bigcup y \in A. SAMEQUOT x (B y))
⟨proof⟩

lemma finite-positions-in-row[simp]:
\[ n > 0 \implies \text{finite} (\text{positions-in-row} (x \oplus' - \text{sconst} (\text{any}, \text{replicate} n \text{False})) \, 0) \]
⟨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 \oplus' - \text{sconst} x) = fin-cut-same x xs
⟨proof⟩

lemma length-cut-same: length (cut-same x s) = (\text{LEAST} n. \text{sdrop} n s = \text{sconst} s)
⟨proof⟩

lemma enc-alt: \text{wf-interp} w I \implies
\[ x \in \text{enc} \ (w, \ I) \iff x @- \ sconst \ ((\text{any}, \ \text{replicate} \ (\text{length} \ I) \ \text{False})) = \text{stream-enc} \ (w, \ I) \]

\begin{proof}
\end{proof}

**Lemma stream-stream-eqI:** \[ \forall (x, \ y) \in \text{sset} \ x. \ x \neq []; \ \forall (x, \ y) \in \text{sset} \ y. \ x \neq []; \]
\[ \text{smap} \ (\lambda (x, \ \text{hd} \ x) \ x) = \text{smap} \ (\lambda (x, \ \text{hd} \ y). \ y) \; \text{smap} \ x = \text{smap} \ y \quad \text{implies} \quad \text{xs} = \text{ys} \]

\begin{proof}
\end{proof}

**Lemma project-enc-extend:**
\[
\text{fixes} \ x \ I
\text{defines} \ n \equiv \text{length} \ I
\text{defines} \ z \equiv \lambda n. \ (\text{any}, \ \text{replicate} \ n \ \text{False})
\text{defines} \ I' \equiv \text{Inr} \ (\text{positions-in-row} \ (x @- \ sconst \ (z \ (\text{Suc} \ n)))) \ 0 \# I
\text{assumes} \ w: \ \text{wf-interp} \ w \ I
\text{assumes} \ enc: \ \text{fin-cut-same} \ (z \ n) \ (\text{map} \ \pi \ x) @ \text{replicate} \ m \ (z \ n) \in \text{enc} \ (w, \ I)
\text{assumes} \ \text{nonempty}: \ \forall (x, \ y) \in \text{set} \ x. \ x \neq []
\text{shows} \ x \in \text{enc} \ (w, \ I')
\]

\begin{proof}
\end{proof}

**Lemma pred-case-conv:** \[ x - \text{Suc} \ 0 = (\text{case} \ x \ of \ 0 \Rightarrow 0 \mid \text{Suc} \ m \Rightarrow m) \]

\begin{proof}
\end{proof}

**Lemma in-pred-image-iff:** \[ 0 \notin X \quad \Rightarrow \quad (x \in (\lambda x. \ x - \text{Suc} \ 0) \triangleq X) = (\text{Suc} \ x \in X) \]

\begin{proof}
\end{proof}

**Lemma map-project-Int-ENC:**
\[
\text{fixes} \ X \ Z \ n
\text{defines} \ z \equiv (\text{any}, \ \text{replicate} \ n \ \text{False})
\text{assumes} \ 0 \notin X \ W \subseteq \{0 \ldots < n + 1\} \ Z \subseteq \text{lists} \ ((\text{set} \ o \ \Sigma) \ (n + 1))
\text{shows} \ \text{SAMEQUOT} \ z \ (\text{map} \ \pi \cdot (Z \cap \text{lang} \ (n + 1) \ (\text{ENC} \ (n + 1) \ X))) = \text{SAMEQUOT} \ z \ (\text{map} \ (\pi \cdot Z) \cap \text{lang} \ n \ (\text{ENC} \ n \ (\lambda x. \ (x - 1) \cdot X))
\]

\begin{proof}
\end{proof}

**Lemma lang-ENC-split:**
\[
\text{assumes} \ \text{finite} \ X \ W = W1 \cap W2 \ n = 0 \lor (\forall p \in X. \ p < n)
\text{shows} \ \text{lang} \ n \ (\text{ENC} \ n \ X) = \text{lang} \ n \ (\text{ENC} \ n \ W1) \cap \text{lang} \ n \ (\text{ENC} \ n \ W2)
\]

\begin{proof}
\end{proof}

**Lemma map-project-ENC:**
\[
\text{fixes} \ n
\text{assumes} \ X \subseteq \{0 \ldots < n + 1\} \ Z \subseteq \text{lists} \ ((\text{set} \ o \ \Sigma) \ (n + 1))
\text{defines} \ z \equiv (\text{any}, \ \text{replicate} \ n \ \text{False})
\text{shows} \ \text{SAMEQUOT} \ z \ (\text{map} \ \pi \cdot (Z \cap \text{lang} \ (n + 1) \ (\text{ENC} \ (n + 1) \ X))) =
\text{if} \ 0 \in X
\text{then} \ \text{SAMEQUOT} \ z \ (\text{map} \ (\pi \cdot (Z \cap \text{lang} \ (n + 1) \ (\text{ENC} \ (n + 1) \ \{0\}))) \cap \text{lang} \ n \ (\text{ENC} \ n \ ((\lambda x. \ (x - 1) \cdot (X - \{0\}))))
\text{else} \ \text{SAMEQUOT} \ z \ (\text{map} \ (\pi \cdot Z) \cap \text{lang} \ n \ (\text{ENC} \ n \ ((\lambda x. \ (x - 1) \cdot (X - \{0\}))))
\text{is} \ i = (\text{if} \ - \text{then} \ ?R1 \ \text{else} \ ?R2))
\]

75
\[\begin{aligned}\text{lemma } & \text{lang}_{M_{2L}}\text{-rexp-of}'\text{-rexp-of}'' : \\
& \text{wf-formula } n \varphi \implies \text{lang } n \text{ (rexp-of}' n \varphi) = \text{lang } n \text{ (rexp-of}'' n \varphi) \\
\end{aligned}\]

\[\begin{aligned}\text{theorem } & \text{lang}_{W S_{1S}}\text{-rexp-of} : \\
& \text{wf-formula } n \varphi \implies \text{lang}_{W S_{1S}} n \varphi = \text{lang } n \text{ (rexp-of}' n \varphi) \\
\end{aligned}\]

\[\begin{aligned}\text{theorem } & \text{lang}_{W S_{1S}}\text{-rexp-of}'' : \\
& \text{wf-formula } n \varphi \implies \text{lang}_{W S_{1S}} n \varphi = \text{lang } n \text{ (rexp-of}'' n \varphi) \\
\end{aligned}\]

\[\begin{aligned}14 \text{ Normalization of WS1S Formulas}\end{aligned}\]

\[\begin{aligned}\text{fun } & \text{nNot } \text{where} \\
& \text{nNot } (\text{FNot } \varphi) = \varphi \\
& \text{nNot } (\text{FAnd } \varphi_1 \varphi_2) = \text{FOR} \ (\text{nNot } \varphi_1) \ (\text{nNot } \varphi_2) \\
& \text{nNot } (\text{FOR } \varphi_1 \varphi_2) = \text{FAnd} \ (\text{nNot } \varphi_1) \ (\text{nNot } \varphi_2) \\
& \text{nNot } \varphi = \text{FNot } \varphi \\
\text{primrec } & \text{norm } \text{where} \\
& \text{norm } (\text{FQ } a \ m) = \text{FQ } a \ m \\
& \text{norm } (\text{FLess } m \ n) = \text{FLess } m \ n \\
& \text{norm } (\text{FIN } m \ M) = \text{FIN } m \ M \\
& \text{norm } (\text{FOR } \varphi \psi) = \text{FOR} \ (\text{norm } \varphi) \ (\text{norm } \psi) \\
& \text{norm } (\text{FAnd } \varphi \psi) = \text{FAnd} \ (\text{norm } \varphi) \ (\text{norm } \psi) \\
& \text{norm } (\text{FNot } \varphi) = \text{nNot } (\text{norm } \varphi) \\
& \text{norm } (\text{FEXISTS } \varphi) = \text{FEXISTS} \ (\text{norm } \varphi) \\
& \text{norm } (\text{FEXISTS } \varphi) = \text{FEXISTS} \ (\text{norm } \varphi) \\
\text{context } & \text{formula} \\
\text{begin} \\
\text{lemma } & \text{satisfies-nNot[simp]: } (w, I) \models \text{nNot } \varphi \iff (w, I) \models \text{FNot } \varphi \\
\text{lemma } & \text{FOV-nNot[simp]: } \text{FOV} (\text{nNot } \varphi) = \text{FOV} (\text{FNot } \varphi) \\
\text{lemma } & \text{SOV-nNot[simp]: } \text{SOV} (\text{nNot } \varphi) = \text{SOV} (\text{FNot } \varphi) \\
\text{lemma } & \text{pre-wf-formula-nNot[simp]: } \text{pre-wf-formula } n \ (\text{nNot } \varphi) = \text{pre-wf-formula} \\
\end{aligned}\]
n (FNot \varphi)
⟨proof⟩

**lemma** \textit{FOV-norm[simp]}: \textit{FOV} (norm \varphi) = \textit{FOV} \varphi
⟨proof⟩

**lemma** \textit{SOV-norm[simp]}: \textit{SOV} (norm \varphi) = \textit{SOV} \varphi
⟨proof⟩

**lemma** \textit{pre-wf-formula-norm[simp]}: \textit{pre-wf-formula} n (norm \varphi) = \textit{pre-wf-formula} n \varphi
⟨proof⟩

**lemma** \textit{satisfies-norm[simp]}: wI \models \textit{norm} \varphi \iff wI \models \varphi
⟨proof⟩

**lemma** \textit{lang}_{WS1S} n (norm \varphi) = \textit{lang}_{WS1S} n \varphi
⟨proof⟩

end

15 Deciding Equivalence of WS1S Formulas

**permanent-interpretation** embed2 set o \sigma \Sigma \textit{wf-atom} \Sigma \pi \textit{lookup} \varepsilon \Sigma split Singleton

for \Sigma :: 'a :: \textit{linorder} list

**defining**

\[ \mathcal{D} = \text{embed.llderiv lookup} (\varepsilon \Sigma) \]

and \[ \text{Co}\mathcal{D} = \text{embed.llderiv-dual lookup} (\varepsilon \Sigma) \]

and \[ r\mathcal{D} = \text{embed.rlderiv lookup} (\varepsilon \Sigma) \]

and \[ r\mathcal{D}.\text{add} = \text{embed2.rlderiv-and-add lookup} (\varepsilon \Sigma) \]

and \[ \mathcal{Q} = \text{embed2.samequot-exec lookup} (\varepsilon \Sigma) \text{ (split Singleton)} \]

⟨proof⟩

**lemma** \textit{enum-not-empty[simp]}: Enum.enum \neq [] (is ?enum \neq [])
⟨proof⟩

**permanent-interpretation** \Phi: formula Enum.enum :: 'a :: \{\textit{enum, linorder}\} list

**defining**

\[ \text{pre-wf-formula} = \Phi.\text{pre-wf-formula} \]

and \[ \text{wf-formula} = \Phi.\text{wf-formula} \]

and \[ \text{rexp-of} = \Phi.\text{rexp-of} \]

and \[ \text{rexp-of-alt} = \Phi.\text{rexp-of-alt} \]

and \[ \text{rexp-of-alt}' = \Phi.\text{rexp-of-alt}' \]

and \[ \text{rexp-of}'' = \Phi.\text{rexp-of}'' \]

and \[ \text{valid-ENC} = \Phi.\text{valid-ENC} \]
and \( ENC = \Phi.ENC \)
and \( \text{dec-interp} = \Phi.\text{stream-dec} \)
and \( \text{any} = \Phi.\text{any} \)

where \( \text{embed2.somequot-exec lookup} (\varepsilon (\text{Enum.enum} :: 'a \text{ list})) \) (split Singleton)
= \( \Omega \) \( \text{Enum.enum} \)

⟨proof⟩

\textbf{lemmas} \( \text{lang}_{\text{WSIS}}\text{-rexp-of-norm} = \text{trans}[\text{OF sym}[\text{OF} \Phi.\text{lang}_{\text{WSIS}}\text{-norm}] \Phi.\text{lang}_{\text{WSIS}}\text{-rexp-of}] \)
\textbf{lemmas} \( \text{lang}_{\text{WSIS}}\text{-rexp-of'}\text{-norm} = \text{trans}[\text{OF sym}[\text{OF} \Phi.\text{lang}_{\text{WSIS}}\text{-norm}] \Phi.\text{lang}_{\text{WSIS}}\text{-rexp-of'}] \)
\textbf{lemmas} \( \text{lang}_{\text{WSIS}}\text{-rexp-of''-norm} = \text{trans}[\text{OF sym}[\text{OF} \Phi.\text{lang}_{\text{WSIS}}\text{-norm}] \Phi.\text{lang}_{\text{WSIS}}\text{-rexp-of''}] \)

⟨ML⟩

\textbf{permanent-interpretation} \( D : \text{rexp-DFA} \sigma \Sigma \text{wf-atom} \Sigma \pi \text{lookup} \lambda x. \langle \text{pnorm} \rangle \)
\textbf{(inorm \( x \rangle \)}
\( \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{final alphabet.wf (wf-atom} \Sigma) n \text{pnorm lang} \Sigma n n \)
\textbf{for} \( \Sigma :: 'a :: \text{linorder list} \) \textbf{and} \( n :: \text{nat} \)

\textbf{defining}
\textbf{test} = \text{rexp-DA.test} \textbf{(final :: 'a atom rexp ⇒ bool)}
\textbf{and step} = \text{rexp-DA.step} \langle \sigma \Sigma \rangle \langle \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{pnorm n} \rangle
\textbf{and closure} = \text{rexp-DA.closure} \langle \sigma \Sigma \rangle \langle \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{final pnorm n} \rangle
\textbf{and check-eqvRE} = \text{rexp-DA.check-eqv} \langle \sigma \Sigma \rangle \langle \lambda x. \langle \text{pnorm} \langle \text{inorm} x \rangle \rangle \rangle \langle \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{final pnorm n} \rangle
\textbf{and test-invariant} = \text{rexp-DA.test-invariant} \textbf{(final :: 'a atom rexp ⇒ bool)} ::
\langle (\langle a × \text{bool list} \rangle \text{list} × -) \rangle \text{list} × - ⇒ \text{bool}
\textbf{and closure-invariant} = \text{rexp-DA.closure-invariant} \langle \sigma \Sigma \rangle \langle \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{final pnorm n} \rangle
\textbf{and counterexampleRE} = \text{rexp-DA.counterexample} \langle \sigma \Sigma \rangle \langle \lambda x. \langle \text{pnorm} \langle \text{inorm} x \rangle \rangle \rangle \langle \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{final pnorm n} \rangle
\textbf{and reachable} = \text{rexp-DAreachable} \langle \sigma \Sigma \rangle \langle \lambda x. \langle \text{pnorm} \langle \text{inorm} x \rangle \rangle \rangle \langle \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{pnorm n} \rangle
\textbf{and automaton} = \text{rexp-DA.automaton} \langle \sigma \Sigma \rangle \langle \lambda x. \langle \text{pnorm} \langle \text{inorm} x \rangle \rangle \rangle \langle \lambda a. \langle \mathcal{D} \Sigma a \mathcal{r} \rangle \text{pnorm n} \rangle

⟨proof⟩

\textbf{definition} \text{check-eqv where}
\textbf{check-eqv} n \varphi \psi \longmapsto \text{uf-formula} n (\text{FOV} \varphi \psi) \land
\textbf{slow.check-eqvRE} \text{Enum.enum} n (\text{rexp-of''} n (\text{norm} \varphi)) (\text{rexp-of''} n (\text{norm} \psi))

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

\textbf{lemma} \text{soundness:} \textbf{slow.check-eqv} n \varphi \psi \implies \Phi.\text{lang}_{\text{WSIS}} n \varphi = \Phi.\text{lang}_{\text{WSIS}} n \psi

78
lemma completeness:
assumes $\Phi.\text{lang} W S^1 S n \varphi = \Phi.\text{lang} W S^1 S n \psi$ \text{wf-formula} n (FOr \varphi \psi)
shows slow.check-eqv n \varphi \psi

<proof>

<ML>

permanent-interpretation $D$: rexp-DA-no-post \sigma \Sigma \text{wf-atom} \Sigma \pi \text{lookup} \lambda x. pnorm (inorm x)
\lambda a. pnorm (\Sigma a r) \text{final alphabet.\text{uf} (\text{wf-atom} \Sigma) n lang S n n}
for \Sigma :: 'a :: \text{linorder list} and n :: \text{nat}
defining
test = rexp-DA.test (final :: 'a atom rexp => bool)
and step = rexp-DA.step (\sigma \Sigma) (\lambda a. r. pnorm (\Sigma a r)) id n
and closure = rexp-DA.closure (\sigma \Sigma) (\lambda a. r. pnorm (\Sigma a r)) final id n
and check-eqvRE = rexp-DA.check-eqv (\sigma \Sigma) (\lambda x. pnorm (inorm x)) (\lambda a. r. pnorm (\Sigma a r)) final id n
and test-invariant = rexp-DA.test-invariant (final :: 'a atom rexp => bool) :: (('a \times \text{bool list}) \text{list} \times \cdot) \text{list} \times \cdot => \text{bool}
and step-invariant = rexp-DA.step-invariant (\sigma \Sigma) (\lambda a. r. pnorm (\Sigma a r)) id n
and closure-invariant = rexp-DA.closure-invariant (\sigma \Sigma) (\lambda a. r. pnorm (\Sigma a r)) final id n
and counterexampleRE = rexp-DA.counterexample (\sigma \Sigma) (\lambda x. pnorm (inorm x)) (\lambda a. r. pnorm (\Sigma a r)) final id n
and reachable = rexp-DAreachable (\sigma \Sigma) (\lambda x. pnorm (inorm x)) (\lambda a. r. pnorm (\Sigma a r)) id n
and automaton = rexp-DA.automaton (\sigma \Sigma) (\lambda x. pnorm (inorm x)) (\lambda a. r. pnorm (\Sigma a r)) id n
<proof>

definition check-eqv where
check-eqv n \varphi \psi \leftarrow \text{wf-formula} n (FOr \varphi \psi) \land
\text{fast.check-eqvRE Enum.enum n (rexp-of"n (norm \varphi)) (rexp-of"n (norm \psi))}

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

lemma soundness: fast.check-eqv n \varphi \psi => \Phi.\text{lang} W S^1 S n \varphi = \Phi.\text{lang} W S^1 S n \psi
<proof>

<ML>

79
permanent-interpretation $D$: rexp-DA-no-post $\sigma \Sigma \pi$ lookup
$\lambda x. \text{pnorm-dual (rexp-dual-of (inorm x))} \lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$ final-dual alphabet wf-dual (wf-atom $\Sigma$) n lang-dual $\Sigma$ n n

for $\Sigma :: \text{a :: linorder list and } n :: \text{nat}$

defining

test = rexp-DA.test (final-dual :: \text{a atom rexp-dual \Rightarrow bool})
and step = rexp-DA.step ($\sigma \Sigma$) ($\lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$) final-dual id n
and closure = rexp-DA.closure ($\sigma \Sigma$) ($\lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$) final-dual id n
and check-eqvRE = rexp-DA.check-eqv ($\sigma \Sigma$) ($\lambda x. \text{pnorm-dual (rexp-dual-of (inorm x)))} \lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$) final-dual id n
and test-invariant = rexp-DA.test-invariant (final-dual :: \text{a atom rexp-dual \Rightarrow bool}) ::

((\text{a x bool list) list x -} \Rightarrow \text{bool}) ::
and step-invariant = rexp-DA.step-invariant ($\sigma \Sigma$) ($\lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$) id n
and closure-invariant = rexp-DA.closure-invariant ($\sigma \Sigma$) ($\lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$) final-dual id n
and counterexampleRE = rexp-DA.counterexample ($\sigma \Sigma$) ($\lambda x. \text{pnorm-dual (rexp-dual-of (inorm x)))} \lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$) final-dual id n
and reachable = rexp-DAreachable ($\sigma \Sigma$) ($\lambda x. \text{pnorm-dual (rexp-dual-of (inorm x)))} \lambda a r. \text{pnorm-dual (Co} \Sigma \text{ a r)}$) id n

(ML)

References