

A Coalgebraic Decision Procedure for WS1S

Dmitriy Traytel

January 19, 2015

Contents

1	Equivalence Framework	1
1.1	Abstract Deterministic Automaton	3
1.2	The overall procedure	5
1.3	Abstract Deterministic Finite Automaton	6
2	Preliminaries	7
3	Abstract formulas	7
4	Normalization	21
5	Derivatives of Formulas	23
6	Finiteness of Derivatives Modulo ACI	25
7	Emptiness Check	27
8	Restrictions	30
9	Concrete Atomic WS1S Formulas	34
10	Interpretation	40
11	Examples	50

1 Equivalence Framework

abbreviation $\mathfrak{d}s \equiv \text{fold } (\lambda a L. \mathfrak{d} L a)$

lemma *in-language- $\mathfrak{d}s$* : *in-language* $(\mathfrak{d}s w L) v \longleftrightarrow \text{in-language } L (w @ v)$
by (*induct* *w arbitrary*: *L*) *simp-all*

lemma *o- $\mathfrak{d}s$* : $\mathfrak{o} (\mathfrak{d}s w L) \longleftrightarrow \text{in-language } L w$

```

by (induct w arbitrary: L) auto

lemma in-language-to-language[simp]: in-language (to-language L) w  $\longleftrightarrow$  w  $\in$  L
  by (metis in-language-to-language mem-Collect-eq)

lemma rtrancl-fold-product:
shows {((r, s), (f a r, f a s)) | a r s. a  $\in$  A}  $^*$  =
  {((r, s), (fold f w r, fold f w s)) | w r s. w  $\in$  lists A} (is ?L = ?R)
proof-
  { fix r s r' s'
    have ((r, s), (r', s')) : ?L  $\implies$  ((r, s), (r', s'))  $\in$  ?R
    proof(induction rule: converse-rtrancl-induct2)
      case refl show ?case by (force intro!: fold-simps(1)[symmetric])
    next
      case step thus ?case by (force intro!: fold-simps(2)[symmetric])
    qed
  } moreover
  { fix r s r' s'
    { fix w assume w  $\in$  lists A
      then have ((r, s), fold f w r, fold f w s)  $\in$  ?L
      proof(induction w rule: rev-induct)
        case Nil show ?case by simp
      next
        case snoc thus ?case by (force elim!: rtrancl-into-rtrancl)
      qed
    }
    hence ((r, s), (r', s'))  $\in$  ?R  $\implies$  ((r, s), (r', s'))  $\in$  ?L by auto
  } ultimately show ?thesis by (auto 10 0)
qed

lemma rtrancl-fold-product1:
shows {(r, s).  $\exists a \in A. s = f a r$ }  $^*$  = {(r, s).  $\exists a \in$  lists A.  $s = \text{fold } f a r$ } (is
  ?L = ?R)
proof-
  { fix r s
    have (r, s)  $\in$  ?L  $\implies$  (r, s)  $\in$  ?R
    proof(induction rule: converse-rtrancl-induct)
      case base show ?case by (force intro!: fold-simps(1)[symmetric])
    next
      case step thus ?case by (force intro!: fold-simps(2)[symmetric])
    qed
  } moreover
  { fix r s
    { fix w assume w  $\in$  lists A
      then have (r, fold f w r)  $\in$  ?L
      proof(induction w rule: rev-induct)
        case Nil show ?case by simp
      next
        case snoc thus ?case by (force elim!: rtrancl-into-rtrancl)
      qed
    }
  }

```

```

    qed
  }
  hence  $(r, s) \in ?R \implies (r, s) \in ?L$  by auto
} ultimately show  $?thesis$  by (auto 10 0)
qed

lemma lang-eq-ext-Nil-fold-Deriv:
  fixes  $K L A$ 
  assumes  $\bigwedge w. \text{in-language } K w \implies w \in \text{lists } A \bigwedge w. \text{in-language } L w \implies w \in \text{lists } A$ 
  defines  $\mathfrak{B} \equiv \{(\mathfrak{d}s w K, \mathfrak{d}s w L) \mid w. w \in \text{lists } A\}$ 
  shows  $K = L \longleftrightarrow (\forall (K, L) \in \mathfrak{B}. \mathfrak{o} K \longleftrightarrow \mathfrak{o} L)$ 
proof
  assume  $\forall (K, L) \in \mathfrak{B}. \mathfrak{o} K = \mathfrak{o} L$ 
  then show  $K = L$ 
  unfolding  $\mathfrak{B}$ -def using assms(1,2)
  proof (coinduction arbitrary:  $K L$ )
    case (Lang  $K L$ )
    then have CIH:  $\bigwedge K' L'. \exists w. K' = \mathfrak{d}s w K \wedge L' = \mathfrak{d}s w L \wedge w \in \text{lists } A \implies \mathfrak{o} K' = \mathfrak{o} L'$  and
      [dest, simp]:  $\bigwedge w. \text{in-language } K w \implies w \in \text{lists } A \bigwedge w. \text{in-language } L w \implies w \in \text{lists } A$ 
      by blast+
    show ?case unfolding ex-simps simp-thms
    proof (safe del: iffI)
      show  $\mathfrak{o} K = \mathfrak{o} L$  by (intro CIH[OF exI[where  $x = []$ ]]) simp
    next
      fix  $x w$  assume  $\forall x \in \text{set } w. x \in A$ 
      then show  $\mathfrak{o} (\mathfrak{d}s w (\mathfrak{d} K x)) = \mathfrak{o} (\mathfrak{d}s w (\mathfrak{d} L x))$ 
      proof (cases  $x \in A$ )
        assume  $x \notin A$ 
        then show ?thesis unfolding in-language-ds in-language.simps[symmetric]
        by fastforce
      qed (intro CIH[OF exI[where  $x = x \# w$ ]], auto)
      qed (auto simp add: in-language.simps[symmetric] simp del: in-language.simps)
    qed
  qed (auto simp:  $\mathfrak{B}$ -def)

```

1.1 Abstract Deterministic Automaton

```

locale DA =
  fixes alphabet :: 'a list
  fixes init :: 't  $\Rightarrow$  's
  fixes delta :: 'a  $\Rightarrow$  's  $\Rightarrow$  's
  fixes accept :: 's  $\Rightarrow$  bool
  fixes wellformed :: 's  $\Rightarrow$  bool
  fixes wf :: 't  $\Rightarrow$  bool
  fixes Language :: 's  $\Rightarrow$  'a language
  fixes Lang :: 't  $\Rightarrow$  'a language

```

assumes *Language-init*: $\text{Language } (\text{init } t) = \text{Lang } t$
assumes *wellformed-init*: $\text{wf } t \implies \text{wellformed } (\text{init } t)$
assumes *Language-alphabet*: $\llbracket \text{wellformed } s; \text{in-language } (\text{Language } s) \ w \rrbracket \implies w \in \text{lists } (\text{set alphabet})$
assumes *wellformed-delta*: $\llbracket \text{wellformed } s; a \in \text{set alphabet} \rrbracket \implies \text{wellformed } (\text{delta } a \ s)$
assumes *Language-delta*: $\llbracket \text{wellformed } s; a \in \text{set alphabet} \rrbracket \implies \text{Language } (\text{delta } a \ s) = \mathfrak{d} (\text{Language } s) \ a$
assumes *accept-Language*: $\text{wellformed } s \implies \text{accept } s \longleftrightarrow \mathfrak{o} (\text{Language } s)$
begin

lemma *wellformed-deltas*: $\llbracket \text{wellformed } s; w \in \text{lists } (\text{set alphabet}) \rrbracket \implies \text{wellformed } (\text{fold delta } w \ s)$
by (*induction w arbitrary: s*) (*auto simp add: Language-delta wellformed-delta*)

lemma *Language-deltas*: $\llbracket \text{wellformed } s; w \in \text{lists } (\text{set alphabet}) \rrbracket \implies \text{Language } (\text{fold delta } w \ s) = \mathfrak{d} s \ w \ (\text{Language } s)$
by (*induction w arbitrary: s*) (*auto simp add: Language-delta wellformed-delta*)

abbreviation *closure* :: $'s * 's \Rightarrow (('s * 's) \text{ list } * ('s * 's) \text{ set}) \text{ option}$ **where**
closure $\equiv \text{rtrancl-while } (\lambda(p, q). \text{accept } p = \text{accept } q)$
 $(\lambda(p, q). \text{map } (\lambda a. (\text{delta } a \ p, \text{delta } a \ q)) \ \text{alphabet})$

theorem *closure-sound-complete*:

assumes *wf*: $\text{wf } r \ \text{wf } s$

and result: $\text{closure } (\text{init } r, \text{init } s) = \text{Some } (ws, R)$ (**is** $\text{closure } (?r, ?s) = -$)

shows $ws = [] \longleftrightarrow \text{Lang } r = \text{Lang } s$

proof –

from *wf* **have** *wellformed*: $\text{wellformed } ?r \ \text{wellformed } ?s$ **using** *wellformed-init*

by *blast+*

note *Language-alphabets[simp]* =

Language-alphabet[*OF* *wellformed*(1)] *Language-alphabet*[*OF* *wellformed*(2)]

note *Language-deltass* = *Language-deltas*[*OF* *wellformed*(1)] *Language-deltas*[*OF* *wellformed*(2)]

have *bisim*: $(\text{Language } ?r = \text{Language } ?s) =$

$(\forall a \ b. (\exists w. a = \mathfrak{d} s \ w \ (\text{Language } ?r) \wedge b = \mathfrak{d} s \ w \ (\text{Language } ?s) \wedge w \in \text{lists } (\text{set alphabet}))) \longrightarrow$

$\mathfrak{o} \ a = \mathfrak{o} \ b)$

by (*subst lang-eq-ext-Nil-fold-Deriv*) *auto*

have *leq*: $(\text{Language } ?r = \text{Language } ?s) =$

$(\forall (r', s') \in \{((r, s), (\text{delta } a \ r, \text{delta } a \ s)) \mid a \ r \ s. a \in \text{set alphabet}\}^* \text{ `` } \{(?r, ?s)\}.$

accept $r' = \text{accept } s')$ **using** *Language-deltass*

accept-Language[*OF* *wellformed-deltas*[*OF* *wellformed*(1)]]

accept-Language[*OF* *wellformed-deltas*[*OF* *wellformed*(2)]]

unfolding *rtrancl-fold-product in-lists-conv-set bisim*

by (*auto 10 0*)

```

have  $\{(x,y). y \in \text{set } ((\lambda(p,q). \text{map } (\lambda a. (\text{delta } a \ p, \text{delta } a \ q)) \ \text{alphabet}) \ x))\} =$ 
 $\{((r, s), (\text{delta } a \ r, \text{delta } a \ s)) \mid a \ r \ s. a \in \text{set alphabet}\}$  by auto
with rtrancl-while-Some[OF result]
have  $(ws = []) = (\text{Language } ?r = \text{Language } ?s)$  by  $(\text{auto simp add: leq Ball-def}$ 
split: if-splits)
then show ?thesis unfolding Language-init .
qed

```

1.2 The overall procedure

definition *check-equiv* :: $'t \Rightarrow 't \Rightarrow \text{bool}$ **where**
check-equiv $r \ s = (wf \ r \wedge wf \ s \wedge (\text{case closure } (init \ r, init \ s) \text{ of } \text{Some}([], -) \Rightarrow \text{True}$
 $\mid - \Rightarrow \text{False}))$

lemma *soundness*:

assumes *check-equiv* $r \ s$ **shows** $\text{Lang } r = \text{Lang } s$

proof –

obtain R **where** $wf \ r \ wf \ s \ \text{closure } (init \ r, init \ s) = \text{Some}([], R)$

using *assms* **by** $(\text{auto simp: check-equiv-def Let-def split: option.splits list.splits})$

from *closure-sound-complete*[*OF this*] **show** $\text{Lang } r = \text{Lang } s$ **by** *simp*

qed

Auxiliary functions:

definition *reachable* :: $'a \text{ list} \Rightarrow 's \Rightarrow 's \text{ set}$ **where**

reachable $as \ s = \text{snd } (\text{the } (\text{rtrancl-while } (\lambda-. \text{True}) (\lambda s. \text{map } (\lambda a. \text{delta } a \ s) \ as)$
 $s))$

definition *automaton* :: $'a \text{ list} \Rightarrow 's \Rightarrow (('s * 'a) * 's) \text{ set}$ **where**

automaton $as \ s =$

snd $(\text{the}$

$(\text{let } start = (([s], \{s\}), \{\});$

$test = \lambda((ws, Z), A). ws \neq [];$

$step = \lambda((ws, Z), A).$

$(\text{let } s = \text{hd } ws;$

$\text{new-edges} = \text{map } (\lambda a. ((s, a), \text{delta } a \ s)) \ as;$

$\text{new} = \text{remdups } (\text{filter } (\lambda ss. ss \notin Z) (\text{map } \text{snd } \text{new-edges}))$

$\text{in } ((\text{new} @ \text{tl } ws, \text{set } \text{new} \cup Z), \text{set } \text{new-edges} \cup A))$

$\text{in } \text{while-option } test \ step \ start))$

definition *match* :: $'s \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**

match $s \ w = \text{accept } (\text{fold } \text{delta } w \ s)$

lemma *match-correct*:

assumes *wellformed* $s \ w \in \text{lists } (\text{set alphabet})$

shows $\text{match } s \ w \longleftrightarrow \text{in-language } (\text{Language } s) \ w$

unfolding *match-def accept-Language*[*OF wellformed-deltas*[*OF assms*]] *Language-deltas*[*OF assms*]
 $\text{0-}\partial s \ ..$

end

1.3 Abstract Deterministic Finite Automaton

locale *DFA* = *DA* +

assumes *fin*: *wellformed s* \implies *finite* {*fold delta w s* | *w*. *w* \in *lists (set alphabet)*}

begin

lemma *finite-rtrancl-delta-Image*:

$\llbracket \text{wellformed } r; \text{wellformed } s \rrbracket \implies$

finite ({(*(r, s), (delta a r, delta a s)*) | *a r s*. *a* \in *set (alphabet)*})^{*} “ {(*r, s*)}

unfolding *rtrancl-fold-product Image-singleton*

by (*auto intro: finite-subset[OF - finite-cartesian-product[OF fin fin]]*)

lemma *termination*:

assumes *wellformed r wellformed s*

shows $\exists st. \text{closure } (r, s) = \text{Some } st$ (**is** $\exists -. \text{closure } ?i = -$)

proof (*rule rtrancl-while-finite-Some*)

show *finite* ({(*(x, st)*. *st* \in *set ((λ(p,q). map (λa. (delta a p, delta a q)) alphabet*))}^{*} “ {*?i*})

by (*rule finite-subset[OF Image-mono[OF rtrancl-mono] finite-rtrancl-delta-Image[OF assms]]*) *auto*

qed

lemma *completeness*:

assumes *wf r wf s Lang r = Lang s* **shows** *check-equiv r s*

proof –

obtain *ws R* **where** *1: closure (init r, init s) = Some (ws, R)* **using** *termination*

wellformed-init assms **by** *fastforce*

with *closure-sound-complete[OF - - this] assms*

show *check-equiv r s* **by** (*simp add: check-equiv-def Language-alphabet*)

qed

lemma *finite-rtrancl-delta-Image1*:

wellformed r \implies *finite* ({(*(r, s)*. $\exists a \in \text{set alphabet. } s = \text{delta } a \text{ } r$ })^{*} “ {*r*})

unfolding *rtrancl-fold-product1* **by** (*auto intro: finite-subset[OF - fin]*)

lemma

assumes *wellformed r set as \subseteq set alphabet*

shows *reachable: reachable as r* = {*fold delta w r* | *w*. *w* \in *lists (set as)*}

and *finite-reachable: finite (reachable as r)*

proof –

obtain *wsZ* **where** $*$: *rtrancl-while (λ-. True) (λs. map (λa. delta a s) as) r* = *Some wsZ*

using *assms* **by** (*atomize-elim, intro rtrancl-while-finite-Some Image-mono rtrancl-mono*

finite-subset[OF - finite-rtrancl-delta-Image1[of r]]) *auto*

then show *reachable as r* = {*fold delta w r* | *w*. *w* \in *lists (set as)*}

unfolding *reachable-def* **by** (*cases wsZ*)

(*auto dest!: rtrancl-while-Some split: if-splits simp: rtrancl-fold-product1 image-iff*)

then show *finite (reachable as r)* **using** *assms* **by** (*force intro: finite-subset[OF*

```

- fin])
qed

end

```

2 Preliminaries

```

lemma pred-Diff-0[simp]:  $0 \notin A \implies i \in (\lambda x. x - \text{Suc } 0) \text{ ' } A \longleftrightarrow \text{Suc } i \in A$ 
  by (cases i) (fastforce simp: image-iff le-Suc-eq elim: contrapos-np)+

```

```

lemma funpow-cycle-mult:  $(f \text{ ^^ } k) \ x = x \implies (f \text{ ^^ } (m * k)) \ x = x$ 
  by (induct m) (auto simp: funpow-add)

```

```

lemma funpow-cycle:  $(f \text{ ^^ } k) \ x = x \implies (f \text{ ^^ } l) \ x = (f \text{ ^^ } (l \bmod k)) \ x$ 
  by (subst mod-div-equality[symmetric, of l k])
    (simp only: add.commute funpow-add funpow-cycle-mult o-apply)

```

```

lemma funpow-cycle-offset:
  fixes  $f :: 'a \Rightarrow 'a$ 
  assumes  $(f \text{ ^^ } k) \ x = (f \text{ ^^ } i) \ x \ i \leq k \ i \leq l$ 
  shows  $(f \text{ ^^ } l) \ x = (f \text{ ^^ } ((l - i) \bmod (k - i) + i)) \ x$ 
proof -
  from assms have
     $(f \text{ ^^ } (k - i)) ((f \text{ ^^ } i) \ x) = ((f \text{ ^^ } i) \ x)$ 
     $(f \text{ ^^ } l) \ x = (f \text{ ^^ } (l - i)) ((f \text{ ^^ } i) \ x)$ 
  unfolding fun-cong[OF funpow-add[symmetric, unfolded o-def]] by simp-all
  from funpow-cycle[OF this(1), of l - i] this(2) show ?thesis
  by (simp add: funpow-add)
qed

```

```

definition dec k m = (if m > k then m - Suc 0 else m :: nat)

```

3 Abstract formulas

```

datatype-new (discs-sels) ('a, 'k) aformula =
  | FBool bool
  | FBase 'a
  | FNot ('a, 'k) aformula
  | FOr ('a, 'k) aformula ('a, 'k) aformula
  | FAnd ('a, 'k) aformula ('a, 'k) aformula
  | FEx 'k ('a, 'k) aformula
  | FAll 'k ('a, 'k) aformula
datatype-compact aformula
derive linorder aformula

```

fun *nFOR* **where**

nFOR [] = *FBool False*
| *nFOR* [x] = *x*
| *nFOR* (x # xs) = *FOr x (nFOR xs)*

fun *nFAND* **where**

nFAND [] = *FBool True*
| *nFAND* [x] = *x*
| *nFAND* (x # xs) = *FAnd x (nFAND xs)*

definition *NFOR* = *nFOR* o *sorted-list-of-set*

definition *NFAND* = *nFAND* o *sorted-list-of-set*

fun *disjuncts* **where**

disjuncts (*FOr* φ ψ) = *disjuncts* $\varphi \cup$ *disjuncts* ψ
| *disjuncts* φ = { φ }

fun *conjuncts* **where**

conjuncts (*FAnd* φ ψ) = *conjuncts* $\varphi \cup$ *conjuncts* ψ
| *conjuncts* φ = { φ }

fun *disjuncts-list* **where**

disjuncts-list (*FOr* φ ψ) = *disjuncts-list* $\varphi @$ *disjuncts-list* ψ
| *disjuncts-list* φ = [φ]

fun *conjuncts-list* **where**

conjuncts-list (*FAnd* φ ψ) = *conjuncts-list* $\varphi @$ *conjuncts-list* ψ
| *conjuncts-list* φ = [φ]

lemma *finite-juncts*: *finite* (*disjuncts* φ) *finite* (*conjuncts* φ)

and *nonempty-juncts*: *disjuncts* $\varphi \neq \{\}$ *conjuncts* $\varphi \neq \{\}$

by (*induct* φ) *auto*

lemma *juncts-eq-set-juncts-list*:

disjuncts φ = *set* (*disjuncts-list* φ)

conjuncts φ = *set* (*conjuncts-list* φ)

by (*induct* φ) *auto*

lemma *notin-juncts*:

$\llbracket \psi \in \text{disjuncts } \varphi; \text{is-FOr } \psi \rrbracket \implies \text{False}$

$\llbracket \psi \in \text{conjuncts } \varphi; \text{is-FAnd } \psi \rrbracket \implies \text{False}$

by (*induct* φ) *auto*

lemma *juncts-list-singleton*:

$\neg \text{is-FOr } \varphi \implies \text{disjuncts-list } \varphi = [\varphi]$

$\neg \text{is-FAnd } \varphi \implies \text{conjuncts-list } \varphi = [\varphi]$

by (*induct* φ) *auto*

lemma *nonempty-juncts-list*: *conjuncts-list* $\varphi \neq []$ *disjuncts-list* $\varphi \neq []$

using *nonempty-juncts*[*of* φ] **by** (*auto simp: Suc-le-eq juncts-eq-set-juncts-list*)

primrec *norm-ACI* ($\langle \cdot \rangle$) **where**

$\langle FBool\ b \rangle = FBool\ b$
 $\langle FBase\ a \rangle = FBase\ a$
 $\langle FNot\ \varphi \rangle = FNot\ \langle \varphi \rangle$
 $\langle FOr\ \varphi\ \psi \rangle = NFOR\ (disjuncts\ (FOr\ \langle \varphi \rangle\ \langle \psi \rangle))$
 $\langle FAnd\ \varphi\ \psi \rangle = NFAND\ (conjuncts\ (FAnd\ \langle \varphi \rangle\ \langle \psi \rangle))$
 $\langle FEx\ k\ \varphi \rangle = FEx\ k\ \langle \varphi \rangle$
 $\langle FAll\ k\ \varphi \rangle = FAll\ k\ \langle \varphi \rangle$

fun *nf-ACI* **where**

nf-ACI (*FOr* $\psi1\ \psi2$) = (\neg *is-FOr* $\psi1 \wedge$ (*let* $\varphi s = \psi1 \# disjuncts-list\ \psi2$ in
sorted $\varphi s \wedge distinct\ \varphi s \wedge nf-ACI\ \psi1 \wedge nf-ACI\ \psi2$))
 \mid *nf-ACI* (*FAnd* $\psi1\ \psi2$) = (\neg *is-FAnd* $\psi1 \wedge$ (*let* $\varphi s = \psi1 \# conjuncts-list\ \psi2$ in
sorted $\varphi s \wedge distinct\ \varphi s \wedge nf-ACI\ \psi1 \wedge nf-ACI\ \psi2$))
 \mid *nf-ACI* (*FNot* φ) = *nf-ACI* φ
 \mid *nf-ACI* (*FEx* $k\ \varphi$) = *nf-ACI* φ
 \mid *nf-ACI* (*FAll* $k\ \varphi$) = *nf-ACI* φ
 \mid *nf-ACI* $\varphi = True$

lemma *nf-ACI-D*:

nf-ACI $\varphi \implies sorted\ (disjuncts-list\ \varphi)$
nf-ACI $\varphi \implies sorted\ (conjuncts-list\ \varphi)$
nf-ACI $\varphi \implies distinct\ (disjuncts-list\ \varphi)$
nf-ACI $\varphi \implies distinct\ (conjuncts-list\ \varphi)$
nf-ACI $\varphi \implies list-all\ nf-ACI\ (disjuncts-list\ \varphi)$
nf-ACI $\varphi \implies list-all\ nf-ACI\ (conjuncts-list\ \varphi)$
by (*induct* φ) (*auto simp: juncts-list-singleton*)

lemma *disjuncts-list-nFOR*:

$\llbracket list-all\ (\lambda x. \neg is-FOr\ x)\ \varphi s; \varphi s \neq [] \rrbracket \implies disjuncts-list\ (nFOR\ \varphi s) = \varphi s$
by (*induct* φs *rule: nFOR.induct*) (*auto simp: juncts-list-singleton*)

lemma *conjuncts-list-nFAND*:

$\llbracket list-all\ (\lambda x. \neg is-FAnd\ x)\ \varphi s; \varphi s \neq [] \rrbracket \implies conjuncts-list\ (nFAND\ \varphi s) = \varphi s$
by (*induct* φs *rule: nFAND.induct*) (*auto simp: juncts-list-singleton*)

lemma *disjuncts-NFOR*:

$\llbracket finite\ X; X \neq \{\}; \forall x \in X. \neg is-FOr\ x \rrbracket \implies disjuncts\ (NFOR\ X) = X$
unfolding *NFOR-def* **by** (*auto simp: juncts-eq-set-juncts-list list-all-iff disjuncts-list-nFOR*)

lemma *conjuncts-NFAND*:

$\llbracket finite\ X; X \neq \{\}; \forall x \in X. \neg is-FAnd\ x \rrbracket \implies conjuncts\ (NFAND\ X) = X$
unfolding *NFAND-def* **by** (*auto simp: juncts-eq-set-juncts-list list-all-iff conjuncts-list-nFAND*)

lemma *nf-ACI-nFOR*:

$\llbracket sorted\ \varphi s; distinct\ \varphi s; list-all\ nf-ACI\ \varphi s; list-all\ (\lambda x. \neg is-FOr\ x)\ \varphi s \rrbracket \implies$
nf-ACI (*nFOR* φs)

by (*induct* φs *rule*: *nFOR.induct*)
 (*auto simp*: *juncts-list-singleton disjuncts-list-nFOR nf-ACI-D*)

lemma *nf-ACI-nFAND*:
 $\llbracket \text{sorted } \varphi s; \text{distinct } \varphi s; \text{list-all nf-ACI } \varphi s; \text{list-all } (\lambda x. \neg \text{is-FAnd } x) \varphi s \rrbracket \implies$
nf-ACI (nFAND φs)
by (*induct* φs *rule*: *nFAND.induct*)
 (*auto simp*: *juncts-list-singleton conjuncts-list-nFAND nf-ACI-D*)

lemma *nf-ACI-juncts*:
 $\llbracket \psi \in \text{disjuncts } \varphi; \text{nf-ACI } \varphi \rrbracket \implies \text{nf-ACI } \psi$
 $\llbracket \psi \in \text{conjuncts } \varphi; \text{nf-ACI } \varphi \rrbracket \implies \text{nf-ACI } \psi$
by (*induct* φ) *auto*

lemma *nf-ACI-norm-ACI*: *nf-ACI* $\langle \varphi \rangle$
by (*induct* φ)
 (*force simp*: *NFOR-def NFAND-def finite-juncts list-all-iff*
intro!: *nf-ACI-nFOR nf-ACI-nFAND elim: nf-ACI-juncts notin-juncts*) $+$

lemma *nFOR-Cons*: *nFOR* ($x \# xs$) = (*if* $xs = []$ *then* x *else* *FOr* x (*nFOR* xs))
by (*cases* xs) *simp-all*

lemma *nFAND-Cons*: *nFAND* ($x \# xs$) = (*if* $xs = []$ *then* x *else* *FAnd* x (*nFAND* xs))
by (*cases* xs) *simp-all*

lemma *nFOR-disjuncts*: *nf-ACI* $\psi \implies \text{nFOR} (\text{disjuncts-list } \psi) = \psi$
by (*induct* ψ) (*auto simp*: *juncts-list-singleton nFOR-Cons*)

lemma *nFAND-conjuncts*: *nf-ACI* $\psi \implies \text{nFAND} (\text{conjuncts-list } \psi) = \psi$
by (*induct* ψ) (*auto simp*: *juncts-list-singleton nFAND-Cons*)

lemma *NFOR-disjuncts*: *nf-ACI* $\psi \implies \text{NFOR} (\text{disjuncts } \psi) = \psi$
using *nFOR-disjuncts*[*of* ψ] **unfolding** *NFOR-def o-apply juncts-eq-set-juncts-list*
by (*metis finite-set finite-sorted-distinct-unique nf-ACI-D*(1,3) *sorted-list-of-set*)

lemma *NFAND-conjuncts*: *nf-ACI* $\psi \implies \text{NFAND} (\text{conjuncts } \psi) = \psi$
using *nFAND-conjuncts*[*of* ψ] **unfolding** *NFAND-def o-apply juncts-eq-set-juncts-list*
by (*metis finite-set finite-sorted-distinct-unique nf-ACI-D*(2,4) *sorted-list-of-set*)

lemma *norm-ACI-if-nf-ACI*: *nf-ACI* $\varphi \implies \langle \varphi \rangle = \varphi$
by (*induct* φ)
 (*auto simp*: *juncts-list-singleton juncts-eq-set-juncts-list nonempty-juncts-list*
NFOR-def NFAND-def nFOR-Cons nFAND-Cons nFOR-disjuncts nFAND-conjuncts
sorted-Cons sorted-list-of-set-sort-remdups distinct-remdups-id sorted-sort-id
insort-is-Cons)

lemma *norm-ACI-idem*: $\langle \langle \varphi \rangle \rangle = \langle \varphi \rangle$
by (*metis nf-ACI-norm-ACI norm-ACI-if-nf-ACI*)

lemma *norm-ACI-juncts*:
nf-ACI $\varphi \implies \text{norm-ACI} \text{ ' } \text{disjuncts } \varphi = \text{disjuncts } \varphi$
nf-ACI $\varphi \implies \text{norm-ACI} \text{ ' } \text{conjuncts } \varphi = \text{conjuncts } \varphi$
by (*drule nf-ACI-D(5,6)*, *force simp: list-all-iff juncts-eq-set-juncts-list norm-ACI-if-nf-ACI*) +

lemma
norm-ACI-NFOR: *nf-ACI* $\varphi \implies \varphi = \text{NFOR} (\text{norm-ACI} \text{ ' } \text{disjuncts } \varphi)$ **and**
norm-ACI-NFAND: *nf-ACI* $\varphi \implies \varphi = \text{NFAND} (\text{norm-ACI} \text{ ' } \text{conjuncts } \varphi)$
by (*simp-all add: norm-ACI-juncts NFOR-disjuncts NFAND-conjuncts*)

locale *Formula-Operations* =
fixes *TYPEVARS* :: 'a :: linorder \times 'i \times 'k :: linorder \times 'n \times 'x \times 'v

and *SUC* :: 'k \Rightarrow 'n \Rightarrow 'n
and *LESS* :: 'k \Rightarrow nat \Rightarrow 'n \Rightarrow bool

and *assigns* :: nat \Rightarrow 'i \Rightarrow 'k \Rightarrow 'v (- - [900, 999, 999] 999)
and *nvars* :: 'i \Rightarrow 'n ($\#_V$ - [1000] 900)
and *Extend* :: 'k \Rightarrow nat \Rightarrow 'i \Rightarrow 'v \Rightarrow 'i
and *CONS* :: 'x \Rightarrow 'i \Rightarrow 'i
and *SNOC* :: 'x \Rightarrow 'i \Rightarrow 'i
and *Length* :: 'i \Rightarrow nat

and *extend* :: 'k \Rightarrow bool \Rightarrow 'x \Rightarrow 'x
and *size* :: 'x \Rightarrow 'n
and *zero* :: 'n \Rightarrow 'x

and *eval* :: 'v \Rightarrow nat \Rightarrow bool
and *downshift* :: 'v \Rightarrow 'v
and *upshift* :: 'v \Rightarrow 'v
and *add* :: nat \Rightarrow 'v \Rightarrow 'v
and *cut* :: nat \Rightarrow 'v \Rightarrow 'v
and *len* :: 'v \Rightarrow nat

and *restrict* :: 'k \Rightarrow 'v \Rightarrow bool
and *Restrict* :: 'k \Rightarrow nat \Rightarrow ('a, 'k) *aformula*

and *left-formula0* :: 'a \Rightarrow bool
and *FV0* :: 'a \Rightarrow ('k \times nat) *list*
and *find0* :: 'k \Rightarrow nat \Rightarrow 'a \Rightarrow bool
and *wf0* :: 'n \Rightarrow 'a \Rightarrow bool

and *decr0* :: 'k \Rightarrow nat \Rightarrow 'a \Rightarrow 'a
and *satisfies0* :: 'i \Rightarrow 'a \Rightarrow bool (**infix** \models_0 50)
and *nullable0* :: 'a \Rightarrow bool
and *lderiv0* :: 'x \Rightarrow 'a \Rightarrow ('a, 'k) aformula
and *rderiv0* :: 'x \Rightarrow 'a \Rightarrow ('a, 'k) aformula
begin

abbreviation *LEQ* k l n \equiv *LESS* k l (*SUC* k n)

primrec *FV* **where**

FV (*FBool* -) = []
| *FV* (*FBase* a) = *FV0* a
| *FV* (*FNot* φ) = *FV* φ
| *FV* (*FOr* φ ψ) = *List.union* (*FV* φ) (*FV* ψ)
| *FV* (*FAnd* φ ψ) = *List.union* (*FV* φ) (*FV* ψ)
| *FV* (*FEx* k φ) = *map* ($\lambda(k', x). (k', \text{if } k = k' \text{ then } x - 1 \text{ else } x)$) (*List.remove1* (k, 0) (*FV* φ))
| *FV* (*FAll* k φ) = *map* ($\lambda(k', x). (k', \text{if } k = k' \text{ then } x - 1 \text{ else } x)$) (*List.remove1* (k, 0) (*FV* φ))

primrec *find* **where**

find k l (*FBool* -) = *False*
| *find* k l (*FBase* a) = *find0* k l a
| *find* k l (*FNot* φ) = *find* k l φ
| *find* k l (*FOr* φ ψ) = (*find* k l φ \vee *find* k l ψ)
| *find* k l (*FAnd* φ ψ) = (*find* k l φ \vee *find* k l ψ)
| *find* k l (*FEx* k' φ) = *find* k (*if* k = k' *then* *Suc* l *else* l) φ
| *find* k l (*FAll* k' φ) = *find* k (*if* k = k' *then* *Suc* l *else* l) φ

primrec *wf* :: 'n \Rightarrow ('a, 'k) aformula \Rightarrow bool **where**

wf n (*FBool* -) = *True*
| *wf* n (*FBase* a) = *wf0* n a
| *wf* n (*FNot* φ) = *wf* n φ
| *wf* n (*FOr* φ ψ) = (*wf* n φ \wedge *wf* n ψ)
| *wf* n (*FAnd* φ ψ) = (*wf* n φ \wedge *wf* n ψ)
| *wf* n (*FEx* k φ) = *wf* (*SUC* k n) φ
| *wf* n (*FAll* k φ) = *wf* (*SUC* k n) φ

primrec *left-formula* :: ('a, 'k) aformula \Rightarrow bool **where**

left-formula (*FBool* -) = *True*
| *left-formula* (*FBase* a) = *left-formula0* a
| *left-formula* (*FNot* φ) = *left-formula* φ
| *left-formula* (*FOr* φ ψ) = (*left-formula* φ \wedge *left-formula* ψ)
| *left-formula* (*FAnd* φ ψ) = (*left-formula* φ \wedge *left-formula* ψ)
| *left-formula* (*FEx* k φ) = *left-formula* φ
| *left-formula* (*FAll* k φ) = *left-formula* φ

primrec *decr* :: 'k \Rightarrow nat \Rightarrow ('a, 'k) aformula \Rightarrow ('a, 'k) aformula **where**

decr k l (*FBool* b) = *FBool* b

$\mid \text{decr } k \ l \ (FBase \ a) = FBase \ (\text{decr0 } k \ l \ a)$
 $\mid \text{decr } k \ l \ (FNot \ \varphi) = FNot \ (\text{decr } k \ l \ \varphi)$
 $\mid \text{decr } k \ l \ (FOr \ \varphi \ \psi) = FOr \ (\text{decr } k \ l \ \varphi) \ (\text{decr } k \ l \ \psi)$
 $\mid \text{decr } k \ l \ (FAnd \ \varphi \ \psi) = FAnd \ (\text{decr } k \ l \ \varphi) \ (\text{decr } k \ l \ \psi)$
 $\mid \text{decr } k \ l \ (FEx \ k' \ \varphi) = FEx \ k' \ (\text{decr } k \ (\text{if } k = k' \text{ then } Suc \ l \text{ else } l) \ \varphi)$
 $\mid \text{decr } k \ l \ (FAll \ k' \ \varphi) = FAll \ k' \ (\text{decr } k \ (\text{if } k = k' \text{ then } Suc \ l \text{ else } l) \ \varphi)$

primrec *satisfies-gen* :: $('k \Rightarrow 'v \Rightarrow nat \Rightarrow bool) \Rightarrow 'i \Rightarrow ('a, 'k) \text{ aformula} \Rightarrow bool$
where

$\text{satisfies-gen } r \ \mathfrak{A} \ (FBool \ b) = b$
 $\mid \text{satisfies-gen } r \ \mathfrak{A} \ (FBase \ a) = (\mathfrak{A} \models_0 a)$
 $\mid \text{satisfies-gen } r \ \mathfrak{A} \ (FNot \ \varphi) = (\neg \text{satisfies-gen } r \ \mathfrak{A} \ \varphi)$
 $\mid \text{satisfies-gen } r \ \mathfrak{A} \ (FOr \ \varphi_1 \ \varphi_2) = (\text{satisfies-gen } r \ \mathfrak{A} \ \varphi_1 \vee \text{satisfies-gen } r \ \mathfrak{A} \ \varphi_2)$
 $\mid \text{satisfies-gen } r \ \mathfrak{A} \ (FAnd \ \varphi_1 \ \varphi_2) = (\text{satisfies-gen } r \ \mathfrak{A} \ \varphi_1 \wedge \text{satisfies-gen } r \ \mathfrak{A} \ \varphi_2)$
 $\mid \text{satisfies-gen } r \ \mathfrak{A} \ (FEx \ k \ \varphi) = (\exists P. r \ k \ P \ (\text{Length } \mathfrak{A}) \wedge \text{satisfies-gen } r \ (\text{Extend } k \ 0 \ \mathfrak{A} \ P) \ \varphi)$
 $\mid \text{satisfies-gen } r \ \mathfrak{A} \ (FAll \ k \ \varphi) = (\forall P. r \ k \ P \ (\text{Length } \mathfrak{A}) \longrightarrow \text{satisfies-gen } r \ (\text{Extend } k \ 0 \ \mathfrak{A} \ P) \ \varphi)$

abbreviation *satisfies* (**infix** \models 50) **where**

$\mathfrak{A} \models \varphi \equiv \text{satisfies-gen } (\lambda - \ -. \ True) \ \mathfrak{A} \ \varphi$

abbreviation *satisfies-bounded* (**infix** \models_b 50) **where**

$\mathfrak{A} \models_b \varphi \equiv \text{satisfies-gen } (\lambda - \ P \ n. \ \text{len } P \leq n) \ \mathfrak{A} \ \varphi$

abbreviation *sat-vars-gen* **where**

$\text{sat-vars-gen } r \ V \ \mathfrak{A} \ \varphi \equiv$
 $\text{satisfies-gen } (\lambda k \ P \ n. \ \text{restrict } k \ P \wedge r \ k \ P \ n) \ \mathfrak{A} \ \varphi \wedge (\forall (k, x) \in \text{set } V. \ \text{restrict } k \ (x \mathfrak{A}_k))$

definition *sat* **where**

$\text{sat } \mathfrak{A} \ \varphi \equiv \text{sat-vars-gen } (\lambda - \ -. \ True) \ (FV \ \varphi) \ \mathfrak{A} \ \varphi$

definition *sat_b* **where**

$\text{sat}_b \ \mathfrak{A} \ \varphi \equiv \text{sat-vars-gen } (\lambda - \ P \ n. \ \text{len } P \leq n) \ (FV \ \varphi) \ \mathfrak{A} \ \varphi$

fun *RESTR* **where**

$RESTR \ (FOr \ \varphi \ \psi) = FOr \ (RESTR \ \varphi) \ (RESTR \ \psi)$
 $\mid RESTR \ (FAnd \ \varphi \ \psi) = FAnd \ (RESTR \ \varphi) \ (RESTR \ \psi)$
 $\mid RESTR \ (FNot \ \varphi) = FNot \ (RESTR \ \varphi)$
 $\mid RESTR \ (FEx \ k \ \varphi) = FEx \ k \ (FAnd \ (\text{Restrict } k \ 0) \ (RESTR \ \varphi))$
 $\mid RESTR \ (FAll \ k \ \varphi) = FAll \ k \ (FOr \ (FNot \ (\text{Restrict } k \ 0)) \ (RESTR \ \varphi))$
 $\mid RESTR \ \varphi = \varphi$

abbreviation *RESTRICT-VARS* **where**

$RESTRICT-VARS \ vs \ \varphi \equiv \text{foldr } (\lambda (k, x) \ \varphi. \ FAnd \ (\text{Restrict } k \ x) \ \varphi) \ vs \ (RESTR \ \varphi)$

definition *RESTRICT* **where**

$RESTRICT \ \varphi \equiv RESTRICT\text{-}VARS \ (FV \ \varphi) \ \varphi$

primrec *nullable* :: ('a, 'k) aformula \Rightarrow bool **where**

| *nullable* (FBool b) = b
| *nullable* (FBase a) = nullable0 a
| *nullable* (FNot φ) = (\neg nullable φ)
| *nullable* (FOr φ ψ) = (nullable $\varphi \vee$ nullable ψ)
| *nullable* (FAnd φ ψ) = (nullable $\varphi \wedge$ nullable ψ)
| *nullable* (FEx k φ) = nullable φ
| *nullable* (FAll k φ) = nullable φ

fun *nFOr* :: ('a, 'k) aformula \Rightarrow ('a, 'k) aformula \Rightarrow ('a, 'k) aformula **where**

| *nFOr* (FBool b1) (FBool b2) = FBool (b1 \vee b2)
| *nFOr* (FBool b) ψ = (if b then FBool True else ψ)
| *nFOr* φ (FBool b) = (if b then FBool True else φ)
| *nFOr* (FOr $\varphi1$ $\varphi2$) ψ = *nFOr* $\varphi1$ (*nFOr* $\varphi2$ ψ)
| *nFOr* φ (FOr $\psi1$ $\psi2$) =
| (if $\varphi = \psi1$ then FOr $\psi1$ $\psi2$
| else if $\varphi < \psi1$ then FOr φ (FOr $\psi1$ $\psi2$)
| else FOr $\psi1$ (*nFOr* φ $\psi2$))
| *nFOr* φ ψ =
| (if $\varphi = \psi$ then φ
| else if $\varphi < \psi$ then FOr φ ψ
| else FOr ψ φ)

fun *nFAnd* :: ('a, 'k) aformula \Rightarrow ('a, 'k) aformula \Rightarrow ('a, 'k) aformula **where**

| *nFAnd* (FBool b1) (FBool b2) = FBool (b1 \wedge b2)
| *nFAnd* (FBool b) ψ = (if b then ψ else FBool False)
| *nFAnd* φ (FBool b) = (if b then φ else FBool False)
| *nFAnd* (FAnd $\varphi1$ $\varphi2$) ψ = *nFAnd* $\varphi1$ (*nFAnd* $\varphi2$ ψ)
| *nFAnd* φ (FAnd $\psi1$ $\psi2$) =
| (if $\varphi = \psi1$ then FAnd $\psi1$ $\psi2$
| else if $\varphi < \psi1$ then FAnd φ (FAnd $\psi1$ $\psi2$)
| else FAnd $\psi1$ (*nFAnd* φ $\psi2$))
| *nFAnd* φ ψ =
| (if $\varphi = \psi$ then φ
| else if $\varphi < \psi$ then FAnd φ ψ
| else FAnd ψ φ)

fun *nFNot* :: ('a, 'k) aformula \Rightarrow ('a, 'k) aformula **where**

| *nFNot* (FNot φ) = φ
| *nFNot* (FOr φ ψ) = *nFAnd* (*nFNot* φ) (*nFNot* ψ)
| *nFNot* (FAnd φ ψ) = *nFOr* (*nFNot* φ) (*nFNot* ψ)
| *nFNot* (FEx b φ) = FAll b (*nFNot* φ)
| *nFNot* (FAll b φ) = FEx b (*nFNot* φ)
| *nFNot* (FBool b) = FBool (\neg b)
| *nFNot* φ = FNot φ

fun *nFEx* :: 'k \Rightarrow ('a, 'k) aformula \Rightarrow ('a, 'k) aformula **where**

$nFEx\ k\ (FOr\ \varphi\ \psi) = nFOr\ (nFEx\ k\ \varphi)\ (nFEx\ k\ \psi)$
 $| nFEx\ k\ \varphi = (if\ find\ k\ 0\ \varphi\ then\ FEx\ k\ \varphi\ else\ decr\ k\ 0\ \varphi)$

fun *nFall* **where**

$nFall\ k\ (FAnd\ \varphi\ \psi) = nFAnd\ (nFall\ k\ \varphi)\ (nFall\ k\ \psi)$
 $| nFall\ k\ \varphi = (if\ find\ k\ 0\ \varphi\ then\ FAll\ k\ \varphi\ else\ decr\ k\ 0\ \varphi)$

fun *norm* **where**

$norm\ (FOr\ \varphi\ \psi) = nFOr\ (norm\ \varphi)\ (norm\ \psi)$
 $| norm\ (FAnd\ \varphi\ \psi) = nFAnd\ (norm\ \varphi)\ (norm\ \psi)$
 $| norm\ (FNot\ \varphi) = nFNot\ (norm\ \varphi)$
 $| norm\ (FEx\ k\ \varphi) = nFEx\ k\ (norm\ \varphi)$
 $| norm\ (FAll\ k\ \varphi) = nFAll\ k\ (norm\ \varphi)$
 $| norm\ \varphi = \varphi$

context

fixes *deriv0* :: '*x* \Rightarrow '*a* \Rightarrow ('*a*, '*k*) *aformula*

begin

primrec *deriv* :: '*x* \Rightarrow ('*a*, '*k*) *aformula* \Rightarrow ('*a*, '*k*) *aformula* **where**

$deriv\ x\ (FBool\ b) = FBool\ b$
 $| deriv\ x\ (FBase\ a) = deriv0\ x\ a$
 $| deriv\ x\ (FNot\ \varphi) = FNot\ (deriv\ x\ \varphi)$
 $| deriv\ x\ (FOr\ \varphi\ \psi) = FOr\ (deriv\ x\ \varphi)\ (deriv\ x\ \psi)$
 $| deriv\ x\ (FAnd\ \varphi\ \psi) = FAnd\ (deriv\ x\ \varphi)\ (deriv\ x\ \psi)$
 $| deriv\ x\ (FEx\ k\ \varphi) = FEx\ k\ (FOr\ (deriv\ (extend\ k\ True\ x)\ \varphi)\ (deriv\ (extend\ k\ False\ x)\ \varphi))$
 $| deriv\ x\ (FAll\ k\ \varphi) = FAll\ k\ (FAnd\ (deriv\ (extend\ k\ True\ x)\ \varphi)\ (deriv\ (extend\ k\ False\ x)\ \varphi))$

end

abbreviation *lderv* $\equiv deriv\ lderiv0$

abbreviation *rderiv* $\equiv deriv\ rderiv0$

lemma *fold-deriv-FBool*: $fold\ (deriv\ d0)\ xs\ (FBool\ b) = FBool\ b$

by (*induct xs*) *auto*

lemma *fold-deriv-FNot*:

$fold\ (deriv\ d0)\ xs\ (FNot\ \varphi) = FNot\ (fold\ (deriv\ d0)\ xs\ \varphi)$

by (*induct xs arbitrary: \varphi*) *auto*

lemma *fold-deriv-FOr*:

$fold\ (deriv\ d0)\ xs\ (FOr\ \varphi\ \psi) = FOr\ (fold\ (deriv\ d0)\ xs\ \varphi)\ (fold\ (deriv\ d0)\ xs\ \psi)$

by (*induct xs arbitrary: \varphi \psi*) *auto*

lemma *fold-deriv-FAnd:*

fold (deriv d0) xs (FAnd φ ψ) = FAnd (fold (deriv d0) xs φ) (fold (deriv d0) xs ψ)
by (*induct xs arbitrary: φ ψ*) *auto*

lemma *fold-deriv-FEx:*

{⟨fold (deriv d0) xs (FEx k φ)⟩ | xs. True} ⊆
{FEx k ψ | ψ . nf-ACI $\psi \wedge$ disjuncts $\psi \subseteq (\bigcup xs. \text{disjuncts } \langle \text{fold (deriv d0) xs } \varphi \rangle)$ }

proof –

{ fix xs
have $\exists \psi. \langle \text{fold (deriv d0) xs (FEx k } \varphi) \rangle = \text{FEx k } \psi \wedge$
 $\text{nf-ACI } \psi \wedge \text{disjuncts } \psi \subseteq (\bigcup xs. \text{disjuncts } \langle \text{fold (deriv d0) xs } \varphi \rangle)$
proof (*induct xs arbitrary: φ*)
case (*Cons x xs*)
let $?\varphi = \text{FOr (deriv d0 (extend k True x) } \varphi) (\text{deriv d0 (extend k False x) } \varphi)$
from *Cons[of ? φ]* **obtain** ψ **where** $\langle \text{fold (deriv d0) xs (FEx k } ?\varphi) \rangle = \text{FEx k } \psi$
 $\text{nf-ACI } \psi$ **and** $*: \text{disjuncts } \psi \subseteq (\bigcup xs. \text{disjuncts } \langle \text{fold (deriv d0) xs } ?\varphi \rangle)$ **by**
blast+
then show *?case*
proof (*intro exI conjI*)
have $(\bigcup xs. \text{disjuncts } \langle \text{fold (deriv d0) xs } ?\varphi \rangle) \subseteq$
 $(\bigcup xs. \text{disjuncts } \langle \text{fold (Formula-Operations.deriv extend d0) xs } \varphi \rangle)$
by (*force simp: fold-deriv-FOr finite-juncts nonempty-juncts nf-ACI-juncts*
nf-ACI-norm-ACI
 $\text{dest: notin-juncts set-mp[OF equalityD1[OF disjuncts-NFOR], rotated } -1]$
 $\text{intro: exI[of - extend k b x \# xs for b xs]}$
with $*$ **show** $\text{disjuncts } \psi \subseteq \dots$ **by** *blast*
qed *simp-all*
qed (*auto simp: nf-ACI-norm-ACI intro!: exI[of - []]*)
}
then show *?thesis* **by** *blast*
qed

lemma *fold-deriv-FAll:*

{⟨fold (deriv d0) xs (FAll k φ)⟩ | xs. True} ⊆
{FAll k ψ | ψ . nf-ACI $\psi \wedge$ conjuncts $\psi \subseteq (\bigcup xs. \text{conjuncts } \langle \text{fold (deriv d0) xs } \varphi \rangle)$ }

proof –

{ fix xs
have $\exists \psi. \langle \text{fold (deriv d0) xs (FAll k } \varphi) \rangle = \text{FAll k } \psi \wedge$
 $\text{nf-ACI } \psi \wedge \text{conjuncts } \psi \subseteq (\bigcup xs. \text{conjuncts } \langle \text{fold (deriv d0) xs } \varphi \rangle)$
proof (*induct xs arbitrary: φ*)
case (*Cons x xs*)
let $?\varphi = \text{FAnd (deriv d0 (extend k True x) } \varphi) (\text{deriv d0 (extend k False x) } \varphi)$
from *Cons[of ? φ]* **obtain** ψ **where** $\langle \text{fold (deriv d0) xs (FAll k } ?\varphi) \rangle = \text{FAll k } \psi$
 $\text{nf-ACI } \psi$ **and** $\text{conjuncts } \psi \subseteq (\bigcup xs. \text{conjuncts } \langle \text{fold (deriv d0) xs } ?\varphi \rangle)$ **by**
blast+
then show *?case*
proof (*intro exI conjI*)
have $(\bigcup xs. \text{conjuncts } \langle \text{fold (deriv d0) xs } ?\varphi \rangle) \subseteq$
 $(\bigcup xs. \text{conjuncts } \langle \text{fold (Formula-Operations.deriv extend d0) xs } \varphi \rangle)$
by (*force simp: fold-deriv-FAnd finite-juncts nonempty-juncts nf-ACI-juncts*
nf-ACI-norm-ACI
 $\text{dest: notin-juncts set-mp[OF equalityD1[OF conjuncts-NFOR], rotated } -1]$
 $\text{intro: exI[of - extend k b x \# xs for b xs]}$
with $*$ **show** $\text{conjuncts } \psi \subseteq \dots$ **by** *blast*
qed *simp-all*
qed (*auto simp: nf-ACI-norm-ACI intro!: exI[of - []]*)
}
then show *?thesis* **by** *blast*
qed


```

      nf-ACI  $\psi$  and *: conjuncts  $\psi \subseteq (\bigcup xs. \text{conjuncts } \langle \text{fold } (\text{deriv } d0) \text{ } xs \text{ } ?\varphi \rangle)$ 
by blast+
  then show ?case
  proof (intro exI conjI)
    have  $(\bigcup xs. \text{conjuncts } \langle \text{fold } (\text{deriv } d0) \text{ } xs \text{ } ?\varphi \rangle) \subseteq$ 
       $(\bigcup xs. \text{conjuncts } \langle \text{fold } (\text{Formula-Operations.deriv extend } d0) \text{ } xs \text{ } \varphi \rangle)$ 
    by (force simp: fold-deriv-FAnd finite-juncts nonempty-juncts nf-ACI-juncts
nf-ACI-norm-ACI
      dest: notin-juncts set-mp[OF equalityD1[OF conjuncts-NFAND], rotated
-1]
      intro: exI[of - extend k b x # xs for b xs])
    with * show conjuncts  $\psi \subseteq \dots$  by blast
  qed simp-all
qed (auto simp: nf-ACI-norm-ACI intro!: exI[of - []])
}
then show ?thesis by blast
qed

```

```

lemma finite-norm-ACI-juncts:
  fixes  $f :: ('a, 'k) \text{aformula} \Rightarrow ('a, 'k) \text{aformula}$ 
  shows finite  $B \Rightarrow$  finite  $\{f \varphi \mid \varphi. \text{nf-ACI } \varphi \wedge \text{disjuncts } \varphi \subseteq B\}$ 
  finite  $B \Rightarrow$  finite  $\{f \varphi \mid \varphi. \text{nf-ACI } \varphi \wedge \text{conjuncts } \varphi \subseteq B\}$ 
  by (elim finite-surj[OF iffD2[OF finite-Pow-iff], of - - f o NFOR o image
norm-ACI]
    finite-surj[OF iffD2[OF finite-Pow-iff], of - - f o NFAND o image norm-ACI],
    force simp: Pow-def image-Collect intro: arg-cong[OF norm-ACI-NFOR] arg-cong[OF
norm-ACI-NFAND])+)

```

end

```

locale Formula = Formula-Operations
  where TYPEVARS = TYPEVARS
  for TYPEVARS :: 'a(*l*) :: linorder (* $\times$  'ar :: linorder*) $\times$  'i  $\times$  'k :: linorder
 $\times$  'n  $\times$  'x  $\times$  'v +

```

```

assumes SUC-SUC:  $SUC \ k \ (SUC \ k' \ idx) = SUC \ k' \ (SUC \ k \ idx)$ 
and LEQ-0:  $LEQ \ k \ 0 \ idx$ 
and LESS-SUC:  $LEQ \ k \ (Suc \ l) \ idx = LESS \ k \ l \ idx$ 
 $k \neq k' \Rightarrow LESS \ k \ l \ (SUC \ k' \ idx) = LESS \ k \ l \ idx$ 

```

```

and nvars-Extend:  $\#_V \ (Extend \ k \ i \ \mathfrak{A} \ P) = SUC \ k \ (\#_V \ \mathfrak{A})$ 
and Length-Extend:  $Length \ (Extend \ k \ i \ \mathfrak{A} \ P) = \max \ (Length \ \mathfrak{A}) \ (len \ P)$ 
and Length-0-inj:  $\llbracket Length \ \mathfrak{A} = 0; Length \ \mathfrak{B} = 0; \#_V \ \mathfrak{A} = \#_V \ \mathfrak{B} \rrbracket \Rightarrow \mathfrak{A} = \mathfrak{B}$ 
and ex-Length-0:  $\exists \mathfrak{A}. Length \ \mathfrak{A} = 0 \wedge \#_V \ \mathfrak{A} = idx$ 
and Extend-commute-safe:  $\llbracket j \leq i; LEQ \ k \ i \ (\#_V \ \mathfrak{A}) \rrbracket \Rightarrow$ 
   $Extend \ k \ j \ (Extend \ k \ i \ \mathfrak{A} \ P) \ Q = Extend \ k \ (Suc \ i) \ (Extend \ k \ j \ \mathfrak{A} \ Q) \ P$ 
and Extend-commute-unsafe:  $k \neq k' \Rightarrow$ 
   $Extend \ k \ j \ (Extend \ k' \ i \ \mathfrak{A} \ P) \ Q = Extend \ k' \ i \ (Extend \ k \ j \ \mathfrak{A} \ Q) \ P$ 

```

and assigns-Extend: $LEQ \text{ ord } i (\#_V \mathfrak{A}) \implies$
 $m^{Extend \text{ ord } i \mathfrak{A}} P_{ord'} = (\text{if } ord = ord' \text{ then } (\text{if } m = i \text{ then } P \text{ else } dec \ i \ m^{\mathfrak{A}}_{ord})$
 $\text{else } m^{\mathfrak{A}}_{ord'})$
and assigns-SNOC-zero: $LESS \text{ ord } m (\#_V \mathfrak{A}) \implies m^{SNOC \ (zero \ (\#_V \mathfrak{A})) \ \mathfrak{A}}_{ord}$
 $= m^{\mathfrak{A}}_{ord}$
and Length-CONS: $Length \ (CONS \ x \ \mathfrak{A}) = Length \ \mathfrak{A} + 1$
and Length-SNOC: $Length \ (SNOC \ x \ \mathfrak{A}) = Suc \ (Length \ \mathfrak{A})$
and nvars-CONS: $\#_V \ (CONS \ x \ \mathfrak{A}) = \#_V \ \mathfrak{A}$
and nvars-SNOC: $\#_V \ (SNOC \ x \ \mathfrak{A}) = \#_V \ \mathfrak{A}$
and Extend-CONS: $\#_V \ \mathfrak{A} = size \ x \implies Extend \ k \ 0 \ (CONS \ x \ \mathfrak{A}) \ P =$
 $CONS \ (\text{extend } k \ (\text{if eval } P \ 0 \text{ then True else False}) \ x) \ (Extend \ k \ 0 \ \mathfrak{A} \ (\text{downshift}$
 $P))$
and Extend-SNOC-cut: $\#_V \ \mathfrak{A} = size \ x \implies len \ P \leq Length \ (SNOC \ x \ \mathfrak{A}) \implies$
 $Extend \ ord \ 0 \ (SNOC \ x \ \mathfrak{A}) \ P =$
 $SNOC \ (\text{extend } ord \ (\text{if eval } P \ (Length \ \mathfrak{A}) \text{ then True else False}) \ x) \ (Extend \ ord$
 $0 \ \mathfrak{A} \ (\text{cut } (Length \ \mathfrak{A}) \ P))$

and size-zero: $size \ (zero \ idx) = idx$
and size-extend: $size \ (\text{extend } k \ b \ x) = SUC \ k \ (size \ x)$

and downshift-upshift: $downshift \ (upshift \ P) = P$
and downshift-add-zero: $downshift \ (\text{add } 0 \ P) = downshift \ P$
and eval-add: $eval \ (\text{add } n \ P) \ n$
and eval-upshift: $\neg \ eval \ (upshift \ P) \ 0$
and eval-ge-len: $p \geq len \ P \implies \neg \ eval \ P \ p$
and len-cut-le: $len \ (\text{cut } n \ P) \leq n$
and len-cut: $len \ P \leq n \implies \text{cut } n \ P = P$
and cut-add: $\text{cut } n \ (\text{add } m \ P) = (\text{if } m \geq n \text{ then } \text{cut } n \ P \text{ else } \text{add } m \ (\text{cut } n \ P))$
and len-add: $len \ (\text{add } m \ P) = \max \ (Suc \ m) \ (len \ P)$
and len-upshift: $len \ (upshift \ P) = (\text{case } len \ P \text{ of } 0 \Rightarrow 0 \mid n \Rightarrow Suc \ n)$
and len-downshift: $len \ (downshift \ P) = (\text{case } len \ P \text{ of } 0 \Rightarrow 0 \mid Suc \ n \Rightarrow n)$

and wf0-decr0: $\llbracket wf0 \ (SUC \ k \ idx) \ a; LESS \ k \ l \ (SUC \ k \ idx); \neg \ find0 \ k \ l \ a \rrbracket \implies$
 $wf0 \ idx \ (\text{decr0 } k \ l \ a)$
and left-formula0-decr0: $\text{left-formula0 } \varphi \implies \text{left-formula0 } (\text{decr0 } k \ l \ \varphi)$
and Extend-satisfies0: $\llbracket \neg \ find0 \ k \ i \ a; LESS \ k \ i \ (SUC \ k \ (\#_V \ \mathfrak{A})) \rrbracket \implies$
 $Extend \ k \ i \ \mathfrak{A} \ P \models_0 a \longleftrightarrow \mathfrak{A} \models_0 \text{decr0 } k \ i \ a$
and nullable0-satisfies0: $Length \ \mathfrak{A} = 0 \implies \text{nullable0 } a \longleftrightarrow \mathfrak{A} \models_0 a$
and satisfies0-eqI: $wf0 \ (\#_V \ \mathfrak{B}) \ a \implies \#_V \ \mathfrak{A} = \#_V \ \mathfrak{B} \implies \text{left-formula0 } a \implies$
 $(\bigwedge m \ k. LESS \ k \ m \ (\#_V \ \mathfrak{B}) \implies m^{\mathfrak{A}}_k = m^{\mathfrak{B}}_k) \implies \mathfrak{A} \models_0 a \longleftrightarrow \mathfrak{B} \models_0 a$
and wf-ldriv0: $wf0 \ idx \ a \implies wf \ idx \ (ldriv0 \ x \ a)$
and left-formula-ldriv0: $\text{left-formula0 } a \implies \text{left-formula } (ldriv0 \ x \ a)$
and wf-rderiv0: $wf0 \ idx \ a \implies wf \ idx \ (rderiv0 \ x \ a)$
and satisfies-ldriv0: $\llbracket wf0 \ (\#_V \ \mathfrak{A}) \ a; \#_V \ \mathfrak{A} = size \ x \rrbracket \implies \mathfrak{A} \models \text{ldriv0 } x \ a \longleftrightarrow$
 $CONS \ x \ \mathfrak{A} \models_0 a$

and *satisfies-bounded-lderiv0*: $\llbracket wf0 \ (\#_V \mathfrak{A}) \ a; \ \#_V \mathfrak{A} = size\ x \rrbracket \implies \mathfrak{A} \models_b lderiv0$
 $x \ a \longleftrightarrow CONS\ x \ \mathfrak{A} \models_0 a$
and *satisfies-bounded-rderiv0*: $\llbracket wf0 \ (\#_V \mathfrak{A}) \ a; \ \#_V \mathfrak{A} = size\ x \rrbracket \implies \mathfrak{A} \models_b rderiv0$
 $x \ a \longleftrightarrow SNOC\ x \ \mathfrak{A} \models_0 a$
and *find0-FV0*: $find0\ k\ l\ a \longleftrightarrow (k, l) \in set\ (FV0\ a)$
and *distinct-FV0*: $distinct\ (FV0\ a)$
and *wf0-FV0-LESS*: $\llbracket wf0\ idx\ a; \ (k, v) \in set\ (FV0\ a) \rrbracket \implies LESS\ k\ v\ idx$
and *restrict-Restrict*: $i^{\mathfrak{A}}k = P \implies restrict\ k\ P \longleftrightarrow \mathfrak{A} \models Restrict\ k\ i$
and *restrict-Restrict_b*: $i^{\mathfrak{A}}k = P \implies restrict\ k\ P \longleftrightarrow \mathfrak{A} \models_b Restrict\ k\ i$
and *wf-Restrict*: $LESS\ k\ i\ idx \implies wf\ idx\ (Restrict\ k\ i)$
and *left-formula-Restrict*: $left\text{-}formula\ (Restrict\ k\ i)$
and *finite-lderiv0*: $finite\ \{fold\ lderiv\ xs\ (FBase\ a) \mid xs.\ True\}$
and *finite-rderiv0*: $finite\ \{fold\ rderiv\ xs\ (FBase\ a) \mid xs.\ True\}$

locale *Word-Formula* = *Formula*
where *TYPEVARS* = *TYPEVARS*
for *TYPEVARS* :: 'a :: linorder \times 'i \times 'k :: linorder \times 'n \times 'x \times 'v +
fixes *enc* :: 'i \Rightarrow 'x list
and *alphabet* :: 'n \Rightarrow 'x list
and *ZERO* :: 'n

assumes *enc-inj*: $\#_V \mathfrak{A} = \#_V \mathfrak{B} \implies enc\ \mathfrak{A} = enc\ \mathfrak{B} \longleftrightarrow \mathfrak{A} = \mathfrak{B}$
and *length-enc*: $length\ (enc\ \mathfrak{A}) = Length\ \mathfrak{A}$
and *enc-CONS*: $\#_V \mathfrak{A} = size\ x \implies enc\ (CONS\ x\ \mathfrak{A}) = x \ \# \ enc\ \mathfrak{A}$
and *in-set-encD*: $x \in set\ (enc\ \mathfrak{A}) \implies size\ x = \#_V \mathfrak{A}$
and *alphabet-size*: $x \in set\ (alphabet\ idx) \longleftrightarrow size\ x = idx$

context *Formula*
begin

lemma *satisfies-eqI*:
 $\llbracket wf\ (\#_V \mathfrak{A})\ \varphi; \ \#_V \mathfrak{A} = \#_V \mathfrak{B}; \ \bigwedge m\ k.\ LESS\ k\ m\ (\#_V \mathfrak{A}) \implies m^{\mathfrak{A}}k = m^{\mathfrak{B}}k;$
 $left\text{-}formula\ \varphi \rrbracket \implies$
 $\mathfrak{A} \models \varphi \longleftrightarrow \mathfrak{B} \models \varphi$
proof (*induct* φ *arbitrary*: $\mathfrak{A}\ \mathfrak{B}$)
case (*FEx* $k\ \varphi$)
from *FEx.prem*s **have** $\bigwedge P.\ (Extend\ k\ 0\ \mathfrak{A}\ P \models \varphi) \longleftrightarrow (Extend\ k\ 0\ \mathfrak{B}\ P \models \varphi)$
by (*intro* *FEx.hyps*) (*auto simp: nvars-Extend assigns-Extend dec-def gr0-conv-Suc LEQ-0 LESS-SUC*)
then show ?*case* **by** *simp*
next
case (*FAll* $k\ \varphi$)
from *FAll.prem*s **have** $\bigwedge P.\ (Extend\ k\ 0\ \mathfrak{A}\ P \models \varphi) \longleftrightarrow (Extend\ k\ 0\ \mathfrak{B}\ P \models \varphi)$
by (*intro* *FAll.hyps*) (*auto simp: nvars-Extend assigns-Extend dec-def gr0-conv-Suc LEQ-0 LESS-SUC*)
then show ?*case* **by** *simp*
next
case (*FNot* φ)
from *FNot.prem*s **have** $(\mathfrak{A} \models \varphi) \longleftrightarrow (\mathfrak{B} \models \varphi)$ **by** (*intro* *FNot.hyps*) *simp-all*

then show *?case by simp*
qed (*auto dest: satisfies0-eqI*)

lemma *wf-decr*:
 $\llbracket \text{wf } (SUC\ k\ idx)\ \varphi; LEQ\ k\ l\ idx; \neg\ \text{find}\ k\ l\ \varphi \rrbracket \implies \text{wf}\ idx\ (\text{decr}\ k\ l\ \varphi)$
by (*induct* φ *arbitrary: idx l*) (*auto simp: wf0-decr0 LESS-SUC SUC-SUC*)

lemma *left-formula-decr*:
 $\text{left-formula}\ \varphi \implies \text{left-formula}\ (\text{decr}\ k\ l\ \varphi)$
by (*induct* φ *arbitrary: l*) (*auto simp: left-formula0-decr0*)

lemma *Extend-satisfies-decr*:
 $\llbracket \neg\ \text{find}\ k\ i\ \varphi; LEQ\ k\ i\ (\#_V\ \mathfrak{A}) \rrbracket \implies \text{Extend}\ k\ i\ \mathfrak{A}\ P \models \varphi \longleftrightarrow \mathfrak{A} \models \text{decr}\ k\ i\ \varphi$
by (*induct* φ *arbitrary: i* \mathfrak{A})
(auto simp: Extend-commute-unsafe[of - k 0 - - P] Extend-commute-safe
Extend-satisfies0 nvars-Extend LESS-SUC SUC-SUC split: bool.splits)

lemma *LEQ-SUC*: $k \neq k' \implies LEQ\ k\ i\ (SUC\ k'\ idx) = LEQ\ k\ i\ idx$
by (*metis LESS-SUC(2) SUC-SUC*)

lemma *Extend-satisfies-bounded-decr*:
 $\llbracket \neg\ \text{find}\ k\ i\ \varphi; LEQ\ k\ i\ (\#_V\ \mathfrak{A}); \text{len}\ P \leq \text{Length}\ \mathfrak{A} \rrbracket \implies$
 $\text{Extend}\ k\ i\ \mathfrak{A}\ P \models_b \varphi \longleftrightarrow \mathfrak{A} \models_b \text{decr}\ k\ i\ \varphi$
proof (*induct* φ *arbitrary: i* $\mathfrak{A}\ P$)
case (*FEx* $k'\ \varphi$)
show *?case*
proof (*cases* $k = k'$)
case *True*
with *FEx(2,3,4)* **show** *?thesis*
using *FEx(1)[of Suc i Extend k' 0* $\mathfrak{A}\ Q\ P$ **for** $Q\ j$]
by (*auto simp: Extend-commute-safe LESS-SUC Length-Extend nvars-Extend*
max-def)
next
case *False*
with *FEx(2,3,4)* **show** *?thesis*
using *FEx(1)[of i Extend k' j* $\mathfrak{A}\ Q\ P$ **for** $Q\ j$]
by (*auto simp: Extend-commute-unsafe LEQ-SUC Length-Extend nvars-Extend*
max-def)
qed
next
case (*FAll* $k'\ \varphi$) **show** *?case*
proof (*cases* $k = k'$)
case *True*
with *FAll(2,3,4)* **show** *?thesis*
using *FAll(1)[of Suc i Extend k' 0* $\mathfrak{A}\ Q\ P$ **for** $Q\ j$]
by (*auto simp: Extend-commute-safe LESS-SUC Length-Extend nvars-Extend*
max-def)
next
case *False*

```

with FAll(2,3,4) show ?thesis
  using FAll(1)[of i Extend k' j  $\mathfrak{A}$  Q P for Q j]
  by (auto simp: Extend-commute-unsafe LEQ-SUC Length-Extend nvars-Extend
max-def)
  qed
qed (auto simp: Extend-satisfies0 split: bool.splits)

```

4 Normalization

lemma *wf-nFOr*:

```

wf idx (FOr  $\varphi$   $\psi$ )  $\implies$  wf idx (nFOr  $\varphi$   $\psi$ )
by (induct  $\varphi$   $\psi$  rule: nFOr.induct) (simp-all add: Let-def)

```

lemma *wf-nFAnd*:

```

wf idx (FAnd  $\varphi$   $\psi$ )  $\implies$  wf idx (nFAnd  $\varphi$   $\psi$ )
by (induct  $\varphi$   $\psi$  rule: nFAnd.induct) (simp-all add: Let-def)

```

lemma *wf-nFNot*:

```

wf idx (FNot  $\varphi$ )  $\implies$  wf idx (nFNot  $\varphi$ )
by (induct  $\varphi$  arbitrary: idx rule: nFNot.induct) (auto simp: wf-nFOr wf-nFAnd)

```

lemma *wf-nFEx*:

```

wf idx (FEx b  $\varphi$ )  $\implies$  wf idx (nFEx b  $\varphi$ )
by (induct  $\varphi$  arbitrary: idx rule: nFEx.induct)
  (auto simp: SUC-SUC LEQ-0 LESS-SUC(1) gr0-conv-Suc wf-nFOr intro:
wf0-decr0 wf-decr)

```

lemma *wf-nFAll*:

```

wf idx (FAll b  $\varphi$ )  $\implies$  wf idx (nFAll b  $\varphi$ )
by (induct  $\varphi$  arbitrary: idx rule: nFAll.induct)
  (auto simp: SUC-SUC LEQ-0 LESS-SUC(1) gr0-conv-Suc wf-nFAnd intro:
wf0-decr0 wf-decr)

```

lemma *wf-norm*: *wf idx $\varphi \implies$ wf idx (norm φ)*

```

by (induct  $\varphi$  arbitrary: idx) (simp-all add: wf-nFOr wf-nFAnd wf-nFNot wf-nFEx
wf-nFAll)

```

lemma *left-formula-nFOr*:

```

left-formula (FOr  $\varphi$   $\psi$ )  $\implies$  left-formula (nFOr  $\varphi$   $\psi$ )
by (induct  $\varphi$   $\psi$  rule: nFOr.induct) (simp-all add: Let-def)

```

lemma *left-formula-nFAnd*:

```

left-formula (FAnd  $\varphi$   $\psi$ )  $\implies$  left-formula (nFAnd  $\varphi$   $\psi$ )
by (induct  $\varphi$   $\psi$  rule: nFAnd.induct) (simp-all add: Let-def)

```

lemma *left-formula-nFNot*:

```

left-formula (FNot  $\varphi$ )  $\implies$  left-formula (nFNot  $\varphi$ )
by (induct  $\varphi$  rule: nFNot.induct) (auto simp: left-formula-nFOr left-formula-nFAnd)

```

lemma *left-formula-nFEx*:
 $\text{left-formula } (FEx\ b\ \varphi) \implies \text{left-formula } (nFEx\ b\ \varphi)$
by (*induct* φ *rule*: *nFEx.induct*)
(auto *simp*: *left-formula-nFOr left-formula0-decr0 left-formula-decr*)

lemma *left-formula-nFAll*:
 $\text{left-formula } (FAll\ b\ \varphi) \implies \text{left-formula } (nFAll\ b\ \varphi)$
by (*induct* φ *rule*: *nFAll.induct*)
(auto *simp*: *left-formula-nFAnd left-formula0-decr0 left-formula-decr*)

lemma *left-formula-norm*: $\text{left-formula } \varphi \implies \text{left-formula } (\text{norm } \varphi)$
by (*induct* φ) (*simp-all add*: *left-formula-nFOr left-formula-nFAnd left-formula-nFNot left-formula-nFEx left-formula-nFAll*)

lemma *satisfies-nFOr*:
 $\mathfrak{A} \models nFOr\ \varphi\ \psi \longleftrightarrow \mathfrak{A} \models FOr\ \varphi\ \psi$
by (*induct* $\varphi\ \psi$ *arbitrary*: \mathfrak{A} *rule*: *nFOr.induct*) *auto*

lemma *satisfies-nFAnd*:
 $\mathfrak{A} \models nFAnd\ \varphi\ \psi \longleftrightarrow \mathfrak{A} \models FAnd\ \varphi\ \psi$
by (*induct* $\varphi\ \psi$ *arbitrary*: \mathfrak{A} *rule*: *nFAnd.induct*) *auto*

lemma *satisfies-nFNot*:
 $\mathfrak{A} \models nFNot\ \varphi \longleftrightarrow \mathfrak{A} \models FNot\ \varphi$
by (*induct* φ *arbitrary*: \mathfrak{A})
(auto *simp*: *satisfies-nFOr satisfies-nFAnd*)

lemma *satisfies-nFEx*: $\mathfrak{A} \models nFEx\ b\ \varphi \longleftrightarrow \mathfrak{A} \models FEx\ b\ \varphi$
by (*induct* φ *rule*: *nFEx.induct*)
(auto *simp add*: *satisfies-nFOr Extend-satisfies-decr*
LEQ-0 LESS-SUC(1) nvars-Extend Extend-satisfies0 Extend-commute-safe
Extend-commute-unsafe)

lemma *satisfies-nFAll*: $\mathfrak{A} \models nFAll\ b\ \varphi \longleftrightarrow \mathfrak{A} \models FAll\ b\ \varphi$
by (*induct* φ *rule*: *nFAll.induct*)
(auto *simp add*: *satisfies-nFAnd Extend-satisfies-decr*
Extend-satisfies0 LEQ-0 LESS-SUC(1) nvars-Extend Extend-commute-safe
Extend-commute-unsafe)

lemma *satisfies-norm*: $\mathfrak{A} \models \text{norm } \varphi \longleftrightarrow \mathfrak{A} \models \varphi$
using *satisfies-nFOr satisfies-nFAnd satisfies-nFNot satisfies-nFEx satisfies-nFAll*
by (*induct* φ *arbitrary*: \mathfrak{A}) *simp-all*

lemma *satisfies-bounded-nFOr*:
 $\mathfrak{A} \models_b nFOr\ \varphi\ \psi \longleftrightarrow \mathfrak{A} \models_b FOr\ \varphi\ \psi$
by (*induct* $\varphi\ \psi$ *arbitrary*: \mathfrak{A} *rule*: *nFOr.induct*) *auto*

lemma *satisfies-bounded-nFAnd*:
 $\mathfrak{A} \models_b nFAnd\ \varphi\ \psi \longleftrightarrow \mathfrak{A} \models_b FAnd\ \varphi\ \psi$

by (*induct* φ *arbitrary*: \mathfrak{A} *rule*: *nFAnd.induct*) *auto*

lemma *satisfies-bounded-nFNot*:
 $\mathfrak{A} \models_b \text{nFNot } \varphi \longleftrightarrow \mathfrak{A} \models_b \text{FNot } \varphi$
by (*induct* φ *arbitrary*: \mathfrak{A})
(auto simp: satisfies-bounded-nFOr satisfies-bounded-nFAnd)

lemma *len-cut-0*: $\text{len } (\text{cut } 0 \ P) = 0$
by (*metis* *le-0-eq* *len-cut-le*)

lemma *satisfies-bounded-nFEx*: $\mathfrak{A} \models_b \text{nFEx } b \ \varphi \longleftrightarrow \mathfrak{A} \models_b \text{FEx } b \ \varphi$
by (*induct* φ *rule*: *nFEx.induct*)
(auto 4 4 simp add: satisfies-bounded-nFOr Extend-satisfies-bounded-decr
LEQ-0 LESS-SUC(1) nvars-Extend Length-Extend len-cut-0
Extend-satisfies0 Extend-commute-safe Extend-commute-unsafe cong: ex-cong
split: bool.splits
intro: exI[where $P = \lambda x. P \ x \wedge Q \ x$ for $P \ Q$, OF conjI[rotated]] exI[of -
cut 0 P for P])

lemma *satisfies-bounded-nFAll*: $\mathfrak{A} \models_b \text{nFAll } b \ \varphi \longleftrightarrow \mathfrak{A} \models_b \text{FAll } b \ \varphi$
by (*induct* φ *rule*: *nFAll.induct*)
(auto 4 4 simp add: satisfies-bounded-nFAnd Extend-satisfies-bounded-decr
LEQ-0 LESS-SUC(1) nvars-Extend Length-Extend len-cut-0
Extend-satisfies0 Extend-commute-safe Extend-commute-unsafe cong: split:
bool.splits
intro: exI[where $P = \lambda x. P \ x \wedge Q \ x$ for $P \ Q$, OF conjI[rotated]] dest: spec[of -
cut 0 P for P])

lemma *satisfies-bounded-norm*: $\mathfrak{A} \models_b \text{norm } \varphi \longleftrightarrow \mathfrak{A} \models_b \varphi$
by (*induct* φ *arbitrary*: \mathfrak{A})
(simp-all add: satisfies-bounded-nFOr satisfies-bounded-nFAnd
satisfies-bounded-nFNot satisfies-bounded-nFEx satisfies-bounded-nFAll)

5 Derivatives of Formulas

lemma *wf-ldderiv*:
 $\text{wf idx } \varphi \implies \text{wf idx } (\text{ldderiv } x \ \varphi)$
by (*induct* φ *arbitrary*: $x \ \text{idx}$) (*auto simp: wf-ldderiv0*)

lemma *left-formula-ldderiv*:
 $\text{left-formula } \varphi \implies \text{left-formula } (\text{ldderiv } x \ \varphi)$
by (*induct* φ *arbitrary*: x) (*auto simp: left-formula-ldderiv0*)

lemma *wf-rderiv*:
 $\text{wf idx } \varphi \implies \text{wf idx } (\text{rderiv } x \ \varphi)$
by (*induct* φ *arbitrary*: $x \ \text{idx}$) (*auto simp: wf-rderiv0*)

theorem *satisfies-ldderiv*: $\llbracket \text{wf } (\#_V \ \mathfrak{A}) \ \varphi; \#_V \ \mathfrak{A} = \text{size } x \rrbracket \implies \mathfrak{A} \models \text{ldderiv } x \ \varphi \longleftrightarrow \text{CONS } x \ \mathfrak{A} \models \varphi$

proof (*induct* φ *arbitrary*: $x \mathfrak{A}$)
case ($FEx\ k\ \varphi$)
from $FEx.prem\ FEx.hyps[of\ Extend\ k\ 0\ \mathfrak{A}\ P\ extend\ k\ b\ x\ for\ P\ b]$ **show** $?case$
by (*auto simp*: $nvars-Extend\ size-extend\ Extend-CONS$
 $downshift-upshift\ eval-add\ eval-upshift\ downshift-add-zero$
 $intro: exI[of - add\ 0\ (upshift\ P)\ for\ P]\ exI[of - upshift\ P\ for\ P]$)
next
case ($FAll\ k\ \varphi$)
from $FAll.prem\ FAll.hyps[of\ Extend\ k\ 0\ \mathfrak{A}\ P\ extend\ k\ b\ x\ for\ P\ b]$ **show** $?case$
by (*auto simp*: $nvars-Extend\ size-extend\ Extend-CONS$
 $downshift-upshift\ eval-add\ eval-upshift\ downshift-add-zero$
 $dest: spec[of - add\ 0\ (upshift\ P)\ for\ P]\ spec[of - upshift\ P\ for\ P]$)
qed (*simp-all add: satisfies-lderiv0 split: bool.splits*)

theorem *satisfies-bounded-lderiv*: $\llbracket wf\ (\#_V\ \mathfrak{A})\ \varphi;\ \#_V\ \mathfrak{A} = size\ x \rrbracket \implies \mathfrak{A} \models_b lderiv\ x\ \varphi \longleftrightarrow CONS\ x\ \mathfrak{A} \models_b \varphi$

proof (*induct* φ *arbitrary*: $x \mathfrak{A}$)
case ($FEx\ k\ \varphi$)
note [*simp*] = $nvars-Extend\ size-extend\ Extend-CONS\ Length-CONS$
 $downshift-upshift\ eval-add\ eval-upshift\ downshift-add-zero\ len-add\ len-upshift$
 $len-downshift$
from $FEx.prem\ FEx.hyps[of\ Extend\ k\ 0\ \mathfrak{A}\ P\ extend\ k\ b\ x\ for\ P\ b]$ **show** $?case$
by *auto* (*force intro*: $exI[of - add\ 0\ (upshift\ P)\ for\ P]\ exI[of - upshift\ P\ for\ P]$ *split*: $nat.splits$)
next
case ($FAll\ k\ \varphi$)
note [*simp*] = $nvars-Extend\ size-extend\ Extend-CONS\ Length-CONS$
 $downshift-upshift\ eval-add\ eval-upshift\ downshift-add-zero\ len-add\ len-upshift$
 $len-downshift$
from $FAll.prem\ FAll.hyps[of\ Extend\ k\ 0\ \mathfrak{A}\ P\ extend\ k\ b\ x\ for\ P\ b]$ **show** $?case$
by *auto* (*force dest*: $spec[of - add\ 0\ (upshift\ P)\ for\ P]\ spec[of - upshift\ P\ for\ P]$ *split*: $nat.splits$)
qed (*simp-all add: satisfies-bounded-lderiv0 split: bool.splits*)

theorem *satisfies-bounded-rderiv*:

$\llbracket wf\ (\#_V\ \mathfrak{A})\ \varphi;\ \#_V\ \mathfrak{A} = size\ x \rrbracket \implies \mathfrak{A} \models_b rderiv\ x\ \varphi \longleftrightarrow SNOC\ x\ \mathfrak{A} \models_b \varphi$

proof (*induct* φ *arbitrary*: $x \mathfrak{A}$)
case ($FEx\ k\ \varphi$)
from $FEx.prem\ FEx.hyps[of\ Extend\ k\ 0\ \mathfrak{A}\ P\ extend\ k\ b\ x\ for\ P\ b]$ **show** $?case$
by (*auto simp*: $nvars-Extend\ size-extend\ Extend-SNOC-cut\ len-cut-le\ eval-ge-len$
 $eval-add\ cut-add\ Length-SNOC\ len-add\ len-cut\ le-Suc-eq\ max-def$
 $intro: exI[of - cut\ (Length\ \mathfrak{A})\ P\ for\ P]\ exI[of - add\ (Length\ \mathfrak{A})\ P\ for\ P]$
 $split: if-splits$)
next
case ($FAll\ k\ \varphi$)
from $FAll.prem\ FAll.hyps[of\ Extend\ k\ 0\ \mathfrak{A}\ P\ extend\ k\ b\ x\ for\ P\ b]$ **show** $?case$
by (*auto simp*: $nvars-Extend\ size-extend\ Extend-SNOC-cut\ len-cut-le\ eval-ge-len$)

$eval-add$ $cut-add$ $Length-SNOC$ $len-add$ $len-cut$ $le-Suc-eq$ $max-def$
 $dest: spec[of - cut (Length \mathfrak{A}) P \text{ for } P] spec[of - add (Length \mathfrak{A}) P \text{ for } P]$
 $split: if-splits)$
 $qed (simp-all add: satisfies-bounded-rderiv0 split: bool.splits)$

lemma $wf-norm-rderivs: wf\ idx\ \varphi \implies wf\ idx\ (((norm \circ rderiv\ (zero\ idx))\)^{\wedge k})\ \varphi)$
by $(induct\ k)\ (auto\ simp: wf-norm\ wf-rderiv)$

6 Finiteness of Derivatives Modulo ACI

lemma $finite-fold-deriv:$
assumes $d0 = lderiv0 \vee d0 = rderiv0$
shows $finite\ \{\langle fold\ (deriv\ d0)\ xs\ \varphi \rangle \mid xs. True\}$
proof $(induct\ \varphi)$
case $(FBase\ a)$ **then show** $?case$ **using** $assms$
by $(auto\ intro:$
 $finite-subset[OF\ -\ finite-imageI[OF\ finite-lderv0]]$
 $finite-subset[OF\ -\ finite-imageI[OF\ finite-rderiv0]])$
next
case $(FNot\ \varphi)$
then show $?case$
by $(auto\ simp: fold-deriv-FNot\ intro: finite-surj[OF\ FNot])$
next
case $(FOr\ \varphi\ \psi)$
then show $?case$
by $(auto\ simp: fold-deriv-FOr\ intro!: finite-surj[OF\ finite-cartesian-product[OF\ FOr]])$
next
case $(FAnd\ \varphi\ \psi)$
then show $?case$
by $(auto\ simp: fold-deriv-FAnd\ intro!: finite-surj[OF\ finite-cartesian-product[OF\ FAnd]])$
next
case $(FEx\ k\ \varphi)$
then have $finite\ (\bigcup\ (disjuncts\ ' \{\langle fold\ (deriv\ d0)\ xs\ \varphi \rangle \mid xs. True\}))$
by $(auto\ simp: finite-juncts)$
then have $finite\ (\bigcup\ xs. disjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle)$
by $(rule\ finite-subset[rotated])\ auto$
then have $finite\ \{FEx\ k\ \psi \mid \psi. nf-ACI\ \psi \wedge disjuncts\ \psi \subseteq (\bigcup\ xs. disjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle)\}$
by $(rule\ finite-norm-ACI-juncts)$
then show $?case$
by $(rule\ finite-subset[OF\ fold-deriv-FEx])$
next
case $(FAll\ k\ \varphi)$
then have $finite\ (\bigcup\ (conjuncts\ ' \{\langle fold\ (deriv\ d0)\ xs\ \varphi \rangle \mid xs. True\}))$
by $(auto\ simp: finite-juncts)$
then have $finite\ (\bigcup\ xs. conjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle)$

by (rule finite-subset[rotated]) auto
 then have finite {Fall k ψ | ψ . nf-ACI $\psi \wedge$ conjuncts $\psi \subseteq (\bigcup xs. conjuncts$
 $\langle fold (deriv d0) xs \varphi \rangle)$ }
 by (rule finite-norm-ACI-juncts)
 then show ?case
 by (rule finite-subset[OF fold-deriv-Fall])
 qed (simp add: fold-deriv-FBool)

theorem

finite-fold-lderiv: finite { $\langle fold lderiv xs \varphi \rangle$ | $xs. True$ } and
 finite-fold-rderiv: finite { $\langle fold rderiv xs \varphi \rangle$ | $xs. True$ }
 by (blast intro: nf-ACI-norm-ACI finite-fold-deriv)+

lemma wf-nFOR: wf idx (nFOR φs) \longleftrightarrow ($\forall \varphi \in set \varphi s. wf idx \varphi$)
 by (induct rule: nFOR.induct) auto

lemma wf-nFAND: wf idx (nFAND φs) \longleftrightarrow ($\forall \varphi \in set \varphi s. wf idx \varphi$)
 by (induct rule: nFAND.induct) auto

lemma wf-NFOR: finite $\Phi \implies wf idx (NFOR \Phi) \longleftrightarrow$ ($\forall \varphi \in \Phi. wf idx \varphi$)
 unfolding NFOR-def o-apply by (auto simp: wf-nFOR)

lemma wf-NFAND: finite $\Phi \implies wf idx (NFAND \Phi) \longleftrightarrow$ ($\forall \varphi \in \Phi. wf idx \varphi$)
 unfolding NFAND-def o-apply by (auto simp: wf-nFAND)

lemma satisfies-bounded-nFOR: $\mathfrak{A} \models_b nFOR \varphi s \longleftrightarrow$ ($\exists \varphi \in set \varphi s. \mathfrak{A} \models_b \varphi$)
 by (induct rule: nFOR.induct) (auto simp: satisfies-bounded-nFOR)

lemma satisfies-bounded-nFAND: $\mathfrak{A} \models_b nFAND \varphi s \longleftrightarrow$ ($\forall \varphi \in set \varphi s. \mathfrak{A} \models_b \varphi$)
 by (induct rule: nFAND.induct) (auto simp: satisfies-bounded-nFAND)

lemma satisfies-bounded-NFOR: finite $\Phi \implies \mathfrak{A} \models_b NFOR \Phi \longleftrightarrow$ ($\exists \varphi \in \Phi. \mathfrak{A} \models_b \varphi$)
 unfolding NFOR-def o-apply by (auto simp: satisfies-bounded-nFOR)

lemma satisfies-bounded-NFAND: finite $\Phi \implies \mathfrak{A} \models_b NFAND \Phi \longleftrightarrow$ ($\forall \varphi \in \Phi. \mathfrak{A} \models_b \varphi$)
 unfolding NFAND-def o-apply by (auto simp: satisfies-bounded-nFAND)

lemma wf-juncts:

wf idx $\varphi \longleftrightarrow$ ($\forall \psi \in disjuncts \varphi. wf idx \psi$)
 wf idx $\varphi \longleftrightarrow$ ($\forall \psi \in conjuncts \varphi. wf idx \psi$)
 by (induct φ) auto

lemma wf-norm-ACI: wf idx $\langle \varphi \rangle = wf idx \varphi$
 by (induct φ arbitrary: idx)
 (auto simp: finite-juncts wf-NFOR wf-NFAND ball-Un wf-juncts[symmetric])

lemma satisfies-bounded-disjuncts:

$\mathfrak{A} \models_b \varphi \longleftrightarrow (\exists \psi \in \text{disjuncts } \varphi. \mathfrak{A} \models_b \psi)$
by (*induct* φ *arbitrary*: \mathfrak{A}) *auto*

lemma *satisfies-bounded-conjuncts*:
 $\mathfrak{A} \models_b \varphi \longleftrightarrow (\forall \psi \in \text{conjuncts } \varphi. \mathfrak{A} \models_b \psi)$
by (*induct* φ *arbitrary*: \mathfrak{A}) *auto*

lemma *satisfies-bounded-norm-ACI*: $\mathfrak{A} \models_b \langle \varphi \rangle \longleftrightarrow \mathfrak{A} \models_b \varphi$
by (*rule sym*, *induct* φ *arbitrary*: \mathfrak{A})
(auto simp: satisfies-bounded-NFOR satisfies-bounded-NFAND finite-juncts
intro: iffD2[OF satisfies-bounded-disjuncts] iffD2[OF satisfies-bounded-conjuncts]
dest: iffD1[OF satisfies-bounded-disjuncts] iffD1[OF satisfies-bounded-conjuncts])

lemma *nvars-SNOCs*: $\#_V ((\text{SNOC } x \hat{^k} \mathfrak{A}) = \#_V \mathfrak{A}$
by (*induct* k) (*auto simp: nvars-SNOC*)

lemma *wf-fold-rderiv*: $\text{wf idx } \varphi \implies \text{wf idx } (\text{fold rderiv } (\text{replicate } k \ x) \ \varphi)$
by (*induct* k *arbitrary*: φ) (*auto simp: wf-rderiv*)

lemma *satisfies-bounded-fold-rderiv*:
 $\llbracket \text{wf idx } \varphi; \#_V \mathfrak{A} = \text{idx}; \text{size } x = \text{idx} \rrbracket \implies$
 $\mathfrak{A} \models_b \text{fold rderiv } (\text{replicate } k \ x) \ \varphi \longleftrightarrow (\text{SNOC } x \hat{^k} \mathfrak{A}) \models_b \varphi$
by (*induct* k *arbitrary*: $\mathfrak{A} \ \varphi$) (*auto simp: satisfies-bounded-rderiv wf-rderiv nvars-SNOCs*)

7 Emptiness Check

context
fixes $b :: \text{bool}$
and $\text{idx} :: 'n$
and $\psi :: ('a, 'k) \text{ aformula}$
begin

abbreviation *fut-test* $\equiv \lambda(\varphi, \Phi). \varphi \notin \text{set } \Phi$
abbreviation *fut-step* $\equiv \lambda(\varphi, \Phi). (\text{norm } (\text{rderiv } (\text{zero idx}) \ \varphi), \varphi \# \Phi)$
definition *fut-derivs* $k \ \varphi \equiv ((\text{norm } o \ \text{rderiv } (\text{zero idx})) \hat{^k} \varphi$

lemma *fut-derivs-Suc[simp]*: $\text{norm } (\text{rderiv } (\text{zero idx}) \ (\text{fut-derivs } k \ \varphi)) = \text{fut-derivs } (\text{Suc } k) \ \varphi$
unfolding *fut-derivs-def* **by** *auto*

definition *fut-invariant* $=$
 $(\lambda(\varphi, \Phi). \text{wf idx } \varphi \wedge (\forall \varphi \in \text{set } \Phi. \text{wf idx } \varphi) \wedge$
 $(\exists k. \varphi = \text{fut-derivs } k \ \psi \wedge \Phi = \text{map } (\lambda i. \text{fut-derivs } i \ \psi) \ (\text{rev } [0 ..< k])))$
definition *fut-spec* $\varphi \Phi \equiv (\forall \varphi \in \text{set } (\text{snd } \varphi \Phi). \text{wf idx } \varphi) \wedge$
 $(\forall \mathfrak{A}. \#_V \mathfrak{A} = \text{idx} \longrightarrow$
 $(\text{if } b \text{ then } (\exists k. (\text{SNOC } (\text{zero idx}) \hat{^k} \mathfrak{A}) \models_b \psi) \longleftrightarrow (\exists \varphi \in \text{set } (\text{snd } \varphi \Phi). \mathfrak{A} \models_b \varphi)$
 $\text{else } (\forall k. (\text{SNOC } (\text{zero idx}) \hat{^k} \mathfrak{A}) \models_b \psi) \longleftrightarrow (\forall \varphi \in \text{set } (\text{snd } \varphi \Phi). \mathfrak{A} \models_b \varphi)))$

definition *fut-default* =
 (ψ , *sorted-list-of-set* { $\langle \text{fold rderiv (replicate } k \text{ (zero idx)) } \psi \rangle \mid k. \text{ True} \}$)

lemma *finite-fold-rderiv-zeros*: *finite* { $\langle \text{fold rderiv (replicate } k \text{ (zero idx)) } \psi \rangle \mid k. \text{ True} \}$
by (*rule finite-subset*[*OF* - *finite-fold-rderiv*[*of* ψ]]) *blast*

definition *fut* :: (*'a*, *'k*) *aformula* **where**
fut = (*if* *b* *then* *nFOR* *else* *nFAND*) (*snd* (*while-default fut-default fut-test fut-step* (ψ , [])))

context
assumes *wf*: *wf idx* ψ
begin

lemma *wf-fut-derivs*:
wf idx (fut-derivs k ψ)
by (*induct k*) (*auto simp: wf-norm wf-rderiv wf fut-derivs-def*)

lemma *satisfies-bounded-fut-derivs*:
 $\#_V \mathfrak{A} = \text{idx} \implies \mathfrak{A} \models_b \text{fut-derivs } k \ \psi \longleftrightarrow (\text{SNOC (zero idx)} \hat{\ }^k) \mathfrak{A} \models_b \psi$
by (*induct k arbitrary: \mathfrak{A}*) (*auto simp: fut-derivs-def satisfies-bounded-rderiv satisfies-bounded-norm*
wf-norm-rderiv size-zero nvars-SNOC funpow-swap1 [of SNOC x for x] wf)

lemma *fut-init*: *fut-invariant* (ψ , [])
unfolding *fut-invariant-def* **by** (*auto simp: fut-derivs-def wf*)

lemma *fut-spec-default*: *fut-spec fut-default*
using *satisfies-bounded-fold-rderiv*[*OF* *iffD2*[*OF* *wf-norm-ACI wf*] *sym size-zero*]

unfolding *fut-spec-def fut-default-def snd-conv*
conjunct1[*OF* *sorted-list-of-set*[*OF* *finite-fold-rderiv-zeros*]]
by (*auto simp: satisfies-bounded-norm-ACI wf-fold-rderiv wf wf-norm-ACI simp*
del: fold-replicate)

lemma *fut-invariant*: *fut-invariant* $\varphi\Phi \implies \text{fut-test } \varphi\Phi \implies \text{fut-invariant (fut-step } \varphi\Phi)$
by (*cases* $\varphi\Phi$) (*auto simp: fut-invariant-def wf-norm wf-rderiv split: if-splits*)

lemma *fut-terminate*: *fut-invariant* $\varphi\Phi \implies \neg \text{fut-test } \varphi\Phi \implies \text{fut-spec } \varphi\Phi$
proof (*induct* $\varphi\Phi$, *unfold prod.case not-not*)
fix $\varphi \Phi$ **assume** *fut-invariant* (φ , Φ) $\varphi \in \text{set } \Phi$
then obtain *i k* **where** $i < k$ **and** $\varphi\text{-def}$: $\varphi = \text{fut-derivs } i \ \psi$
and $\Phi\text{-def}$: $\Phi = \text{map } (\lambda i. \text{fut-derivs } i \ \psi) (\text{rev } [0..<k])$
and $*$: $\text{fut-derivs } k \ \psi = \text{fut-derivs } i \ \psi$ **unfolding** *fut-invariant-def* **by** *auto*
have $\text{set } \Phi = \{\text{fut-derivs } k \ \psi \mid k. \text{ True}\}$
unfolding $\Phi\text{-def}$ *set-map set-rev set-upt* **proof** *safe*

```

fix j
show fut-derivs j  $\psi \in (\lambda i. \text{fut-derivs } i \ \psi) \text{ ' } \{0..<k\}$ 
proof (cases j < k)
  case False
  with *  $\langle i < k \rangle$  have fut-derivs j  $\psi = \text{fut-derivs } ((j - i) \bmod (k - i) + i) \ \psi$ 
    unfolding fut-derivs-def by (auto intro: funpow-cycle-offset)
  then show ?thesis using  $\langle i < k \rangle \ \langle \neg j < k \rangle$ 
    by (metis image-eqI atLeastLessThan-iff le0 less-diff-conv mod-less-divisor
zero-less-diff)
  qed simp
qed (blast intro: *)
then show fut-spec ( $\varphi, \Phi$ )
  unfolding fut-spec-def using satisfies-bounded-fut-derivs by (auto simp: wf-fut-derivs)
qed

```

```

lemma fut-spec-while-default:
  fut-spec (while-default fut-default fut-test fut-step ( $\psi, []$ ))
using fut-invariant fut-terminate fut-init fut-spec-default by (rule while-default-rule)

```

```

lemma wf-fut: wf idx fut
using fut-spec-while-default unfolding fut-def fut-spec-def by (auto simp: wf-nFOR
wf-nFAND)

```

```

lemma satisfies-bounded-fut:
  assumes  $\#_V \mathcal{A} = \text{id}_x$ 
  shows  $\mathcal{A} \models_b \text{fut} \longleftrightarrow$ 
    (if b then  $(\exists k. (\text{SNOC } (\text{zero } \text{id}_x) \ \hat{\wedge} \ k) \ \mathcal{A} \models_b \psi)$  else  $(\forall k. (\text{SNOC } (\text{zero } \text{id}_x) \ \hat{\wedge} \ k) \ \mathcal{A} \models_b \psi)$ )
  using fut-spec-while-default assms unfolding fut-def fut-spec-def
  by (auto simp: satisfies-bounded-nFOR satisfies-bounded-nFAND)

```

end

end

```

fun finalize :: 'n  $\Rightarrow$  ('a, 'k) aformula  $\Rightarrow$  ('a, 'k) aformula where
  finalize idx (FEx k  $\varphi$ ) = fut True idx (nFEx k (finalize (SUC k idx)  $\varphi$ ))
| finalize idx (FALL k  $\varphi$ ) = fut False idx (nFALL k (finalize (SUC k idx)  $\varphi$ ))
| finalize idx (FOr  $\varphi \ \psi$ ) = FOr (finalize idx  $\varphi$ ) (finalize idx  $\psi$ )
| finalize idx (FAnd  $\varphi \ \psi$ ) = FAnd (finalize idx  $\varphi$ ) (finalize idx  $\psi$ )
| finalize idx (FNot  $\varphi$ ) = FNot (finalize idx  $\varphi$ )
| finalize idx  $\varphi$  =  $\varphi$ 

```

```

definition final :: 'n  $\Rightarrow$  ('a, 'k) aformula  $\Rightarrow$  bool where
  final idx = nullable o finalize idx

```

```

lemma wf-finalize: wf idx  $\varphi \implies$  wf idx (finalize idx  $\varphi$ )
by (induct  $\varphi$  arbitrary: idx) (auto simp: wf-fut wf-nFEx wf-nFALL)

```

lemma *Length-SNOCs*: $\text{Length } ((\text{SNOC } x \text{ } \wedge \wedge k) \mathfrak{A}) = \text{Length } \mathfrak{A} + k$
by (*induct* k *arbitrary*: \mathfrak{A}) (*auto simp*: *Length-SNOC*)

lemma *assigns-SNOCs-zero*:

$\llbracket \text{LESS } \text{ord } m \text{ } (\#_V \mathfrak{A}); \#_V \mathfrak{A} = \text{idx} \rrbracket \implies m(\text{SNOC } (\text{zero } \text{idx}) \text{ } \wedge \wedge k) \mathfrak{A}_{\text{ord}} = m \mathfrak{A}_{\text{ord}}$
by (*induct* k *arbitrary*: \mathfrak{A}) (*auto simp*: *assigns-SNOC-zero nvars-SNOC funpow-swap1*)

lemma *Extend-SNOCs-zero-satisfies*: $\llbracket \text{wf } (\text{SUC } \text{ord } \text{idx}) \varphi; \#_V \mathfrak{A} = \text{idx}; \text{left-formula } \varphi \rrbracket \implies$
 $\text{Extend } \text{ord } 0 ((\text{SNOC } (\text{zero } (\#_V \mathfrak{A})) \text{ } \wedge \wedge k) \mathfrak{A}) P \models \varphi \longleftrightarrow \text{Extend } \text{ord } 0 \mathfrak{A} P \models$
 φ
by (*rule satisfies-eqI*)
(auto simp: nvars-Extend nvars-SNOCs assigns-Extend assigns-SNOCs-zero LEQ-0 LESS-SUC
dec-def gr0-conv-Suc)

lemma *finalize-satisfies*: $\llbracket \text{wf } \text{idx } \varphi; \#_V \mathfrak{A} = \text{idx}; \text{left-formula } \varphi \rrbracket \implies \mathfrak{A} \models_b \text{finalize}$
 $\text{idx } \varphi \longleftrightarrow \mathfrak{A} \models \varphi$
by (*induct* φ *arbitrary*: $\text{idx } \mathfrak{A}$)
(force simp add: wf-nFEx wf-nFAll wf-finalize Length-SNOCs nvars-Extend
nvars-SNOCs
satisfies-bounded-fut satisfies-bounded-nFEx satisfies-bounded-nFAll Extend-SNOCs-zero-satisfies
intro: le-add2)+

lemma *Extend-empty-satisfies0*:

$\llbracket \text{Length } \mathfrak{A} = 0; \text{len } P = 0 \rrbracket \implies \text{Extend } k \ i \ \mathfrak{A} \ P \models_0 a \longleftrightarrow \mathfrak{A} \models_0 a$
by (*intro box-equals[OF - nullable0-satisfies0 nullable0-satisfies0]*)
(auto simp: nvars-Extend Length-Extend)

lemma *Extend-empty-satisfies-bounded*:

$\llbracket \text{Length } \mathfrak{A} = 0; \text{len } P = 0 \rrbracket \implies \text{Extend } k \ 0 \ \mathfrak{A} \ P \models_b \varphi \longleftrightarrow \mathfrak{A} \models_b \varphi$
by (*induct* φ *arbitrary*: $k \ \mathfrak{A} \ P$)
(auto simp: Extend-empty-satisfies0 Length-Extend split: bool.splits)

lemma *nullable-satisfies-bounded*: $\text{Length } \mathfrak{A} = 0 \implies \text{nullable } \varphi \longleftrightarrow \mathfrak{A} \models_b \varphi$

by (*induct* φ) (*auto simp: nullable0-satisfies0 Extend-empty-satisfies-bounded*
len-cut-0
intro: exI[of - cut 0 P for P])

lemma *final-satisfies*:

$\llbracket \text{wf } \text{idx } \varphi \wedge \text{left-formula } \varphi; \text{Length } \mathfrak{A} = 0; \#_V \mathfrak{A} = \text{idx} \rrbracket \implies \text{final } \text{idx } \varphi = (\mathfrak{A} \models \varphi)$
by (*simp only: final-def o-apply nullable-satisfies-bounded finalize-satisfies*)

8 Restrictions

lemma *satisfies-gen-restrict-RESTR*:

$\text{satisfies-gen } (\lambda k \ P \ . \ \text{restrict } k \ P) \ \mathfrak{A} \ \varphi \longleftrightarrow \mathfrak{A} \models \text{RESTR } \varphi$
by (*induct* φ *arbitrary*: \mathfrak{A}) (*auto simp: restrict-Restrict[symmetric] assigns-Extend*)

LEQ-0)

lemma *satisfies-gen-restrict_b-RESTR*:

satisfies-gen ($\lambda k P n. \text{restrict } k P \wedge \text{len } P \leq n$) $\mathfrak{A} \varphi \longleftrightarrow \mathfrak{A} \models_b \text{RESTR } \varphi$
by (*induct* φ *arbitrary*: \mathfrak{A}) (*auto simp*: *restrict-Restrict_b[symmetric]* *assigns-Extend* *LEQ-0*)

lemma *distinct-FV*: *distinct* (*FV* φ)

by (*induct* φ) (*auto simp*: *distinct-FV0* *distinct-map inj-on-def split: if-splits*)

lemma *sat-vars-RESTRIC-T-VARS*: *sat-vars-gen* ($\lambda - - . \text{True}$) *vs* $\mathfrak{A} \varphi \longleftrightarrow \mathfrak{A} \models$
RESTRIC-T-VARS *vs* φ

by (*induct* *vs* *arbitrary*: φ) (*force simp*: *satisfies-gen-restrict-RESTR restrict-Restrict*) +

lemma *sat-vars_b-RESTRIC-T-VARS*: *sat-vars-gen* ($\lambda - P n. \text{len } P \leq n$) *vs* $\mathfrak{A} \varphi$
 $\longleftrightarrow \mathfrak{A} \models_b \text{RESTRIC-T-VARS}$ *vs* φ

by (*induct* *vs* *arbitrary*: φ) (*auto simp*: *satisfies-gen-restrict_b-RESTR restrict-Restrict_b*) +

lemma *sat-RESTRIC-T*: *sat* $\mathfrak{A} \varphi \longleftrightarrow \mathfrak{A} \models \text{RESTRIC-T } \varphi$

unfolding *sat-def RESTRIC-T-def* **by** (*rule* *sat-vars-RESTRIC-T-VARS*)

lemma *sat_b-RESTRIC-T*: *sat_b* $\mathfrak{A} \varphi \longleftrightarrow \mathfrak{A} \models_b \text{RESTRIC-T } \varphi$

unfolding *sat_b-def RESTRIC-T-def* **by** (*rule* *sat-vars_b-RESTRIC-T-VARS*)

end

context *Word-Formula*

begin

lemma *enc-Nil*: *Length* $\mathfrak{A} = 0 \implies \text{enc } \mathfrak{A} = []$

by (*metis* *length-0-conv length-enc*)

definition *decode* *idx* = *the-inv-into* $\{\mathfrak{B}. \#_V \mathfrak{B} = \text{idx}\}$ *enc*

lemma *inj-on-enc*: *inj-on* *enc* $\{\mathfrak{B}. \#_V \mathfrak{B} = \text{idx}\}$

unfolding *inj-on-def* **by** (*auto simp*: *enc-inj*)

lemma *surj-enc*: $\forall x \in \text{set } xs. \text{size } x = \text{idx} \implies x \in \text{enc } \{\mathfrak{B}. \#_V \mathfrak{B} = \text{idx}\}$

proof (*induct* *xs*)

case *Nil* **then show** *?case* **using** *ex-Length-0[of idx]*

by (*auto dest*: *enc-Nil[symmetric]*)

next

case (*Cons* *x xs*) **then show** *?case*

by (*auto simp*: *image-iff enc-CONS nvars-CONS intro: exI[of - CONS x \mathfrak{A} for \mathfrak{A}]*)

qed

lemma *enc-decode*: $\forall x \in \text{set } xs. \text{size } x = \text{idx} \implies \text{enc } (\text{decode } \text{idx } x) = x$

unfolding *decode-def* **by** (*rule* *f-the-inv-into-f[OF inj-on-enc surj-enc]*)

definition $TL\ \mathfrak{A} = decode\ (\#_V\ \mathfrak{A})\ (tl\ (enc\ \mathfrak{A}))$

lemma $in-set-tlD$: $x \in set\ (tl\ xs) \implies x \in set\ xs$
by $(cases\ xs)\ auto$

lemma $enc-TL$: $enc\ (TL\ \mathfrak{A}) = tl\ (enc\ \mathfrak{A})$
unfolding $TL-def$ **by** $(subst\ enc-decode)\ (auto\ dest!:\ in-set-encD\ in-set-tlD)$

lemma $nvars-TL$: $\#_V\ (TL\ \mathfrak{A}) = \#_V\ \mathfrak{A}$
unfolding $TL-def\ decode-def$
using $the-inv-into-into[OF\ inj-on-enc\ surj-enc\ subset-refl,\ of\ tl\ (enc\ \mathfrak{A})]$
by $(auto\ dest!:\ in-set-encD\ in-set-tlD)$

definition $lang\ idx\ \varphi = \{enc\ \mathfrak{A} \mid \mathfrak{A}. \mathfrak{A} \models \varphi \wedge \#_V\ \mathfrak{A} = idx\}$
definition $lang_b\ idx\ \varphi = \{enc\ \mathfrak{A} \mid \mathfrak{A}. \mathfrak{A} \models_b \varphi \wedge \#_V\ \mathfrak{A} = idx\}$

lemma $lang-eq-iff$: $lang\ idx\ \varphi = lang\ idx\ \psi \longleftrightarrow (\forall \mathfrak{A}. \#_V\ \mathfrak{A} = idx \longrightarrow \mathfrak{A} \models \varphi \longleftrightarrow \mathfrak{A} \models \psi)$
unfolding $lang-def\ set-eq-iff$ **by** $auto\ (metis\ enc-inj)+$

lemma $lang_b-eq-iff$: $lang_b\ idx\ \varphi = lang_b\ idx\ \psi \longleftrightarrow (\forall \mathfrak{A}. \#_V\ \mathfrak{A} = idx \longrightarrow \mathfrak{A} \models_b \varphi \longleftrightarrow \mathfrak{A} \models_b \psi)$
unfolding $lang_b-def\ set-eq-iff$ **by** $auto\ (metis\ enc-inj)+$

lemma $final-iff-Nil$: $wf\ idx\ \varphi \wedge left-formula\ \varphi \implies final\ idx\ \varphi \longleftrightarrow (\square \in lang\ idx\ \varphi)$
using $ex-Length-0[of\ idx]\ Length-0-inj$
 $enc-Nil[symmetric]\ enc-inj\ final-satisfies[of\ idx\ \varphi]$
unfolding $lang-def$ **by** $clarsimp\ metis$

lemma $nullable-iff-Nil$: $wf\ idx\ \varphi \wedge left-formula\ \varphi \implies nullable\ \varphi \longleftrightarrow (\square \in lang_b\ idx\ \varphi)$
using $ex-Length-0[of\ idx]\ Length-0-inj$
 $enc-Nil[symmetric]\ enc-inj\ nullable-satisfies-bounded$
unfolding $lang_b-def$ **by** $clarsimp\ metis$

lemma $lQuot-enc$:
assumes $\bigwedge \mathfrak{A}. P\ \mathfrak{A} \implies \#_V\ \mathfrak{A} = size\ a$
shows $\{w. a \# w \in \{enc\ \mathfrak{A} \mid \mathfrak{A}. P\ \mathfrak{A}\}\} = \{enc\ \mathfrak{A} \mid \mathfrak{A}. P\ (CONS\ a\ \mathfrak{A})\}$
proof $safe$
fix w **and** $\mathfrak{A} :: 'i$ **assume** $a \# w = enc\ \mathfrak{A}\ P\ \mathfrak{A}$
with $assms$ **have** $CONS\ a\ (TL\ \mathfrak{A}) = \mathfrak{A}\ enc\ \mathfrak{A} = a \# w$
by $(auto\ simp:\ enc-inj[symmetric]\ enc-CONS\ enc-TL\ nvars-CONS\ nvars-TL\ dest:\ sym)$
with $\langle P\ \mathfrak{A} \rangle$ **show** $\exists \mathfrak{A}. w = enc\ \mathfrak{A} \wedge P\ (CONS\ a\ \mathfrak{A})$ **by** $(auto\ simp:\ enc-TL\ intro!:\ exI[of\ -\ TL\ \mathfrak{A}])$
next
fix $\mathfrak{A} :: 'i$ **assume** $P\ (CONS\ a\ \mathfrak{A})$

with *assms*[*of CONS a A*] **show** $\exists A'. a \# \text{enc } A = \text{enc } A' \wedge P A'$
by (*auto simp: enc-CONS nvars-CONS intro: exI[of - CONS a A]*)
qed

lemma *lang-lderiv*:

$\llbracket \text{wf idx } \varphi; \text{idx} = \text{size } x \rrbracket \implies \text{lang idx (lderiv } x \varphi) = \{w. x \# w \in \text{lang idx } \varphi\}$
unfolding *lang-def* **by** (*subst lQuot-enc*) (*auto simp: satisfies-lderiv nvars-CONS*)

lemma *lang_b-lderiv*:

$\llbracket \text{wf idx } \varphi; \text{idx} = \text{size } x \rrbracket \implies \text{lang}_b \text{idx (lderiv } x \varphi) = \{w. x \# w \in \text{lang}_b \text{idx } \varphi\}$
unfolding *lang_b-def* **by** (*subst lQuot-enc*) (*auto simp: satisfies-bounded-lderiv nvars-CONS*)

lemma *lang-norm*: $\text{lang idx (norm } \varphi) = \text{lang idx } \varphi$

unfolding *lang-eq-iff* **by** (*simp add: satisfies-norm*)

lemma *lang_b-norm*: $\text{lang}_b \text{idx (norm } \varphi) = \text{lang}_b \text{idx } \varphi$

unfolding *lang_b-eq-iff* **by** (*simp add: satisfies-bounded-norm*)

lemma *lang-size*: $\llbracket w \in \text{lang idx } \varphi; x \in \text{set } w \rrbracket \implies \text{size } x = \text{idx}$

unfolding *lang-def* **by** (*auto elim: in-set-encD*)

lemma *lang_b-size*: $\llbracket w \in \text{lang}_b \text{idx } \varphi; x \in \text{set } w \rrbracket \implies \text{size } x = \text{idx}$

unfolding *lang_b-def* **by** (*auto elim: in-set-encD*)

definition *language idx* $\varphi = \{\text{enc } A \mid A. \text{sat } A \varphi \wedge \#_V A = \text{idx}\}$

definition *language_b idx* $\varphi = \{\text{enc } A \mid A. \text{sat}_b A \varphi \wedge \#_V A = \text{idx}\}$

lemma *language-lang-RESTRICT*: $\text{language idx } \varphi = \text{lang idx (RESTRICT } \varphi)$

unfolding *language-def lang-def* **by** (*auto simp: sat-RESTRICT*)

lemma *language_b-lang_b-RESTRICT*: $\text{language}_b \text{idx } \varphi = \text{lang}_b \text{idx (RESTRICT } \varphi)$

unfolding *language_b-def lang_b-def* **by** (*auto simp: sat_b-RESTRICT*)

lemma *wf-RESTR*: $\text{wf idx } \varphi \implies \text{wf idx (RESTR } \varphi)$

by (*induct* φ *arbitrary: idx*) (*auto simp: wf-Restrict LESS-SUC LEQ-0*)

lemma *wf-RESTRICT-VARS*: $\llbracket \text{wf idx } \varphi; \text{list-all } (\lambda(k, v). \text{LESS } k \ v \ \text{idx}) \ \text{vs} \rrbracket \implies$
 $\text{wf idx (RESTRICT-VARS } \text{vs } \varphi)$

by (*induct* *vs*) (*auto simp: wf-RESTR wf-Restrict*)

lemma *wf-FV-LESS*: $\llbracket \text{wf idx } \varphi; (k, v) \in \text{set (FV } \varphi) \rrbracket \implies \text{LESS } k \ v \ \text{idx}$

by (*induct* φ *arbitrary: idx v*)

(*force simp: distinct-FV wf0-FV0-LESS LESS-SUC diff-Suc split: if-splits nat.splits*) $+$

lemma *wf-RESTRICT*: $\text{wf idx } \varphi \implies \text{wf idx (RESTRICT } \varphi)$

unfolding *RESTRICT-def* **by** (*rule wf-RESTRICT-VARS*) (*auto simp: list-all-iff wf-FV-LESS*)

lemma *left-formula-RESTR*: $\text{left-formula } \varphi \implies \text{left-formula } (\text{RESTR } \varphi)$
by (*induct* φ) (*auto simp*: *left-formula-Restrict*)

lemma *left-formula-RESTRIC-VEARS*: $\text{left-formula } \varphi \implies \text{left-formula } (\text{RESTRIC-VEARS } \varphi)$
by (*induct* φ) (*auto simp*: *left-formula-RESTR left-formula-Restrict*)

lemma *left-formula-RESTRIC*: $\text{left-formula } \varphi \implies \text{left-formula } (\text{RESTRIC } \varphi)$
unfolding *RESTRIC-def* **by** (*rule left-formula-RESTRIC-VEARS*)

end

sublocale *Word-Formula* <
bounded!: *DA alphabet idx* $\lambda \varphi. \text{norm } (\text{RESTRIC } \varphi) \lambda a \varphi. \text{norm } (\text{lderv } a \varphi)$
nullable
 $\lambda \varphi. \text{wf idx } \varphi \wedge \text{left-formula } \varphi \lambda \varphi. \text{wf idx } \varphi \wedge \text{left-formula } \varphi$
 $\lambda \varphi. \text{to-language } (\text{lang}_b \text{ idx } \varphi) \lambda \varphi. \text{to-language } (\text{language}_b \text{ idx } \varphi)$ **for** *idx*
by *unfold-locales*
(*auto simp*: *nullable-iff-Nil lang_b-norm lang_b-lderv wf-norm wf-lderv*
left-formula-norm left-formula-lderv alphabet-size lang_b-size
language_b-lang_b-RESTRIC wf-RESTRIC left-formula-RESTRIC)

sublocale *Word-Formula* <
DA alphabet idx $\lambda \varphi. \text{norm } (\text{RESTRIC } \varphi) \lambda a \varphi. \text{norm } (\text{lderv } a \varphi)$ *final idx*
 $\lambda \varphi. \text{wf idx } \varphi \wedge \text{left-formula } \varphi \lambda \varphi. \text{wf idx } \varphi \wedge \text{left-formula } \varphi$
 $\lambda \varphi. \text{to-language } (\text{lang } \text{idx } \varphi) \lambda \varphi. \text{to-language } (\text{language } \text{idx } \varphi)$ **for** *idx*
by *unfold-locales*
(*auto simp*: *final-iff-Nil lang-norm lang-lderv wf-norm wf-lderv*
left-formula-norm left-formula-lderv alphabet-size lang-size
language-lang-RESTRIC wf-RESTRIC left-formula-RESTRIC)

lemma (**in** *Word-Formula*) *check-equiv-soundness*:
 $\llbracket \#_V \mathfrak{A} = \text{idx}; \text{check-equiv idx } \varphi \psi \rrbracket \implies \text{sat } \mathfrak{A} \varphi \longleftrightarrow \text{sat } \mathfrak{A} \psi$
by (*drule soundness, drule injD[OF bij-is-inj[OF to-language-bij]]*)
(*force simp*: *language-def set-eq-iff enc-inj*)

lemma (**in** *Word-Formula*) *bounded-check-equiv-soundness*:
 $\llbracket \#_V \mathfrak{A} = \text{idx}; \text{bounded.check-equiv idx } \varphi \psi \rrbracket \implies \text{sat}_b \mathfrak{A} \varphi \longleftrightarrow \text{sat}_b \mathfrak{A} \psi$
by (*drule bounded.soundness, drule injD[OF bij-is-inj[OF to-language-bij]]*)
(*force simp*: *language_b-def set-eq-iff enc-inj*)

end

9 Concrete Atomic WS1S Formulas

definition *eval* $P \ x = (x \mid \mid P)$

definition *downshift* $P = (\lambda x. x - \text{Suc } 0) \mid \mid (P \mid \mid \{|0|\})$

definition *upshift* $P = \text{Suc } \mid \mid P$

definition $\text{lift } bs \ i \ P = (\text{if } bs \ ! \ i \ \text{then } \text{fininsert } 0 \ (\text{upshift } P) \ \text{else } \text{upshift } P)$

definition $\text{snoc } n \ bs \ i \ P = (\text{if } bs \ ! \ i \ \text{then } \text{fininsert } n \ P \ \text{else } P)$

definition $\text{cut } n \ P = \text{ffilter } (\lambda i. \ i < n) \ P$

definition $\text{len } P = (\text{if } P = \{\|\} \ \text{then } 0 \ \text{else } \text{Suc } (\text{fMax } P))$

datatype-new $\text{order} = FO \mid SO$

datatype-compat order

derive $\text{linorder } \text{order}$

typedef $\text{idx} = \text{UNIV} :: (\text{nat} \times \text{nat}) \ \text{set} \ \text{by} \ (\text{rule } \text{UNIV-witness})$

setup-lifting $\text{type-definition-idx}$

lift-definition $\text{ZERO} :: \text{idx} \ \text{is} \ (0, 0) \ .$

lift-definition $\text{SUC} :: \text{order} \Rightarrow \text{idx} \Rightarrow \text{idx} \ \text{is}$

$\lambda \text{ord } (m, n). \ \text{case } \text{ord} \ \text{of } FO \Rightarrow (\text{Suc } m, n) \mid SO \Rightarrow (m, \text{Suc } n) \ .$

lift-definition $\text{LESS} :: \text{order} \Rightarrow \text{nat} \Rightarrow \text{idx} \Rightarrow \text{bool} \ \text{is}$

$\lambda \text{ord } l \ (m, n). \ \text{case } \text{ord} \ \text{of } FO \Rightarrow l < m \mid SO \Rightarrow l < n \ .$

abbreviation $\text{LEQ } \text{ord } l \ \text{idx} \equiv \text{LESS } \text{ord } l \ (\text{SUC } \text{ord } \text{idx})$

definition $\text{MSB } Is \equiv$

$\text{if } \forall P \in \text{set } Is. \ P = \{\|\} \ \text{then } 0 \ \text{else } \text{Suc } (\text{Max } (\bigcup P \in \text{set } Is. \ \text{fset } P))$

lemma $\text{MSB-Nil}[\text{simp}]: \text{MSB } \|\ = 0$

unfolding MSB-def **by** simp

lemma $\text{MSB-Cons}[\text{simp}]: \text{MSB } (I \# Is) = \text{max } (\text{if } I = \{\|\} \ \text{then } 0 \ \text{else } \text{Suc } (\text{fMax } I)) \ (\text{MSB } Is)$

unfolding MSB-def **including** fset.lifting

by $\text{transfer } (\text{auto } \text{simp}: \text{Max-Un list-all-iff Sup-bot-conv}(2)[\text{symmetric}] \ \text{simp del}: \text{Sup-bot-conv}(2))$

lemma $\text{MSB-append}[\text{simp}]: \text{MSB } (I1 \ @ \ I2) = \text{max } (\text{MSB } I1) \ (\text{MSB } I2)$

by $(\text{induct } I1) \ \text{auto}$

lemma $\text{MSB-insert-nth}[\text{simp}]:$

$\text{MSB } (\text{insert-nth } n \ P \ Is) = \text{max } (\text{if } P = \{\|\} \ \text{then } 0 \ \text{else } \text{Suc } (\text{fMax } P)) \ (\text{MSB } Is)$

by $(\text{subst } (2) \ \text{append-take-drop-id}[\text{of } n \ Is, \ \text{symmetric}])$

$(\text{simp only}: \text{insert-nth-take-drop MSB-append MSB-Cons MSB-Nil})$

lemma $\text{MSB-greater}:$

$\llbracket i < \text{length } Is; \ p \ |\in| \ Is \ ! \ i \rrbracket \Longrightarrow p < \text{MSB } Is$

unfolding MSB-def **by** $(\text{fastforce } \text{simp}: \text{Bex-def in-set-conv-nth less-Suc-eq-le intro}: \text{Max-ge})$

lemma $\text{MSB-mono}: \text{set } I1 \subseteq \text{set } I2 \Longrightarrow \text{MSB } I1 \leq \text{MSB } I2$

unfolding MSB-def **including** fset.lifting

by $\text{transfer } (\text{auto } \text{simp}: \text{list-all-iff intro!}: \text{Max-ge})$

lemma *MSB-map-index'-CONS*[simp]:
 $MSB \ (map-index' \ i \ (lift \ bs) \ Is) =$
 $(if \ MSB \ Is = 0 \wedge (\forall i \in \{i \dots i + length \ Is\}. \neg bs \ ! \ i) \ then \ 0 \ else \ Suc \ (MSB \ Is))$
by (*induct Is arbitrary: i*)
 $(auto \ split: if-splits \ simp: mono-fMax-commute[where \ f = Suc, \ symmetric]$
mono-def
 $lift-def \ upshift-def,$
 $metis \ atLeastLessThan-iff \ le-antisym \ not-less-eq-eq)$

lemma *MSB-map-index'-SNOC*[simp]:
 $MSB \ Is \leq n \implies MSB \ (map-index' \ i \ (snoc \ n \ bs) \ Is) =$
 $(if \ (\forall i \in \{i \dots i + length \ Is\}. \neg bs \ ! \ i) \ then \ MSB \ Is \ else \ Suc \ n)$
by (*induct Is arbitrary: i*)
 $(auto \ split: if-splits \ simp: mono-fMax-commute[where \ f = Suc, \ symmetric]$
mono-def
 $snoc-def, \ (metis \ atLeastLessThan-iff \ le-antisym \ not-less-eq-eq)+)$

lemma *MSB-replicate*[simp]: $MSB \ (replicate \ n \ P) = (if \ P = \{\|\} \vee n = 0 \ then \ 0$
 $else \ Suc \ (fMax \ P))$
by (*induct n*) *auto*

typedef *interp* =
 $\{(n :: nat, \ I1 :: nat \ fset \ list, \ I2 :: nat \ fset \ list) \mid n \ I1 \ I2. \ MSB \ (I1 \ @ \ I2) \leq n\}$
by *auto*

setup-lifting *type-definition-interp*

lift-definition *assigns* :: $nat \Rightarrow interp \Rightarrow order \Rightarrow nat \ fset \ (- - [900, 999, 999]$
 $999)$
is $\lambda n \ (-, \ I1, \ I2) \ ord. \ case \ ord \ of \ FO \Rightarrow \ if \ n < length \ I1 \ then \ I1 \ ! \ n \ else \ \{\|\}$
 $\mid \ SO \Rightarrow \ if \ n < length \ I2 \ then \ I2 \ ! \ n \ else \ \{\|\} \ .$

lift-definition *nvars* :: $interp \Rightarrow idx \ (\#_V - [1000] \ 900)$
is $\lambda(-, \ I1, \ I2). \ (length \ I1, \ length \ I2) \ .$

lift-definition *Length* :: $interp \Rightarrow nat$
is $\lambda(n, -, -). \ n \ .$

lift-definition *Extend* :: $order \Rightarrow nat \Rightarrow interp \Rightarrow nat \ fset \Rightarrow interp$
is $\lambda ord \ i \ (n, \ I1, \ I2) \ P. \ case \ ord \ of$
 $FO \Rightarrow (max \ n \ (if \ P = \{\|\} \ then \ 0 \ else \ Suc \ (fMax \ P)), \ insert-nth \ i \ P \ I1, \ I2)$
 $\mid \ SO \Rightarrow (max \ n \ (if \ P = \{\|\} \ then \ 0 \ else \ Suc \ (fMax \ P)), \ I1, \ insert-nth \ i \ P \ I2)$
using *MSB-mono* **by** (*auto simp del: insert-nth-take-drop split: order.splits*)

lift-definition *CONS* :: $(bool \ list \times bool \ list) \Rightarrow interp \Rightarrow interp$
is $\lambda(bs1, \ bs2) \ (n, \ I1, \ I2).$
 $(Suc \ n, \ map-index \ (lift \ bs1) \ I1, \ map-index \ (lift \ bs2) \ I2)$
by *auto*

lift-definition *SNOC* :: $(bool \ list \times bool \ list) \Rightarrow interp \Rightarrow interp$

is $\lambda(bs1, bs2) (n, I1, I2).$
 $(Suc\ n, map-index\ (snoc\ n\ bs1)\ I1, map-index\ (snoc\ n\ bs2)\ I2)$
by $(auto\ simp: Let-def)$

abbreviation $enc-atom-bool\ I\ n \equiv map\ (\lambda P. n \in P)\ I$

abbreviation $enc-atom\ I1\ I2\ n \equiv (enc-atom-bool\ I1\ n, enc-atom-bool\ I2\ n)$

type-synonym $atom = bool\ list \times bool\ list$

lift-definition $enc :: interp \Rightarrow atom\ list$
is $\lambda(n, I1, I2). let\ m = MSB\ (I1\ @\ I2)\ in$
 $map\ (enc-atom\ I1\ I2)\ [0 ..< m]\ @$
 $replicate\ (n - m)\ (replicate\ (length\ I1)\ False, replicate\ (length\ I2)\ False) .$

lift-definition $zero :: idx \Rightarrow atom$
is $\lambda(m, n). (replicate\ m\ False, replicate\ n\ False) .$

definition $extend\ ord\ b \equiv$
 $\lambda(bs1, bs2). case\ ord\ of\ FO \Rightarrow (b \# bs1, bs2) \mid SO \Rightarrow (bs1, b \# bs2)$

lift-definition $size-atom :: bool\ list \times bool\ list \Rightarrow idx$
is $\lambda(bs1, bs2). (length\ bs1, length\ bs2) .$

type-synonym $fo = nat$
type-synonym $so = nat$

datatype-new $ws1s =$
 $Q\ fo \mid$
 $Less\ fo\ fo \mid LessF\ fo\ fo \mid LessT\ fo\ fo \mid$
 $In\ fo\ so \mid InT\ fo\ so$

datatype-compact $ws1s$
derive $linorder\ option$
derive $linorder\ ws1s$
type-synonym $formula = (ws1s, order)\ aformula$

primrec $wf0\ where$
 $wf0\ idx\ (Q\ m) = LESS\ FO\ m\ idx$
 $\mid wf0\ idx\ (Less\ m1\ m2) = (LESS\ FO\ m1\ idx \wedge LESS\ FO\ m2\ idx)$
 $\mid wf0\ idx\ (LessF\ m1\ m2) = (LESS\ FO\ m1\ idx \wedge LESS\ FO\ m2\ idx)$
 $\mid wf0\ idx\ (LessT\ m1\ m2) = (LESS\ FO\ m1\ idx \wedge LESS\ FO\ m2\ idx)$
 $\mid wf0\ idx\ (In\ m\ M) = (LESS\ FO\ m\ idx \wedge LESS\ SO\ M\ idx)$
 $\mid wf0\ idx\ (InT\ m\ M) = (LESS\ FO\ m\ idx \wedge LESS\ SO\ M\ idx)$

fun $left-formula0\ where$
 $left-formula0\ x = True$

fun $FV0\ where$
 $FV0\ (Q\ m) = [(FO, m)]$
 $\mid FV0\ (Less\ m1\ m2) = List.insert\ (FO, m1)\ [(FO, m2)]$

```

| FV0 (LessF m1 m2) = List.insert (FO, m1) [(FO, m2)]
| FV0 (LessT m1 m2) = List.insert (FO, m1) [(FO, m2)]
| FV0 (In m M) = [(FO, m), (SO, M)]
| FV0 (InT m M) = [(FO, m), (SO, M)]

```

fun find0 where

```

  find0 FO i (Q m) = (i = m)
| find0 FO i (Less m1 m2) = (i = m1 ∨ i = m2)
| find0 FO i (LessF m1 m2) = (i = m1 ∨ i = m2)
| find0 FO i (LessT m1 m2) = (i = m1 ∨ i = m2)
| find0 FO i (In m -) = (i = m)
| find0 SO i (In - M) = (i = M)
| find0 FO i (InT m -) = (i = m)
| find0 SO i (InT - M) = (i = M)
| find0 - - = False

```

primrec decr0 where

```

  decr0 ord k (Q m) = Q (case-order (dec k) id ord m)
| decr0 ord k (Less m1 m2) = Less (case-order (dec k) id ord m1) (case-order (dec
k) id ord m2)
| decr0 ord k (LessF m1 m2) = LessF (case-order (dec k) id ord m1) (case-order
(dec k) id ord m2)
| decr0 ord k (LessT m1 m2) = LessT (case-order (dec k) id ord m1) (case-order
(dec k) id ord m2)
| decr0 ord k (In m M) = In (case-order (dec k) id ord m) (case-order id (dec k)
ord M)
| decr0 ord k (InT m M) = InT (case-order (dec k) id ord m) (case-order id (dec
k) ord M)

```

primrec satisfies0 where

```

  satisfies0 A (Q m) = (mAFO ≠ {||})
| satisfies0 A (Less m1 m2) =
  (let P1 = m1AFO; P2 = m2AFO in if P1 = {||} ∨ P2 = {||} then False else
fMin P1 < fMin P2)
| satisfies0 A (LessF m1 m2) =
  (let P1 = m1AFO; P2 = m2AFO in
  if P1 = {||} then False else if P2 = {||} then True else fMin P1 < fMin P2)
| satisfies0 A (LessT m1 m2) =
  (let P1 = m1AFO; P2 = m2AFO in
  if P2 = {||} then True else if P1 = {||} then False else fMin P1 < fMin P2)
| satisfies0 A (In m M) =
  (let P = mAFO in if P = {||} then False else fMin P |∈ MASO)
| satisfies0 A (InT m M) =
  (let P = mAFO in if P = {||} then True else fMin P |∈ MASO)

```

fun lderiv0 where

```

  lderiv0 (bs1, bs2) (Q m) = (if bs1 ! m then FBool True else FBase (Q m))
| lderiv0 (bs1, bs2) (Less m1 m2) = (case (bs1 ! m1, bs1 ! m2) of
  (False, False) ⇒ FBase (Less m1 m2)

```

```

| (True, False) ⇒ FBase (Q m2)
| - ⇒ FBool False)
| lderiv0 (bs1, bs2) (LessF m1 m2) = (case (bs1 ! m1, bs1 ! m2) of
  (False, False) ⇒ FBase (LessF m1 m2)
| (True, False) ⇒ FBool True
| - ⇒ FBool False)
| lderiv0 (bs1, bs2) (LessT m1 m2) = (case (bs1 ! m1, bs1 ! m2) of
  (False, False) ⇒ FBase (LessT m1 m2)
| (True, False) ⇒ FBool True
| - ⇒ FBool False)
| lderiv0 (bs1, bs2) (In m M) = (case (bs1 ! m, bs2 ! M) of
  (False, -) ⇒ FBase (In m M)
| (True, True) ⇒ FBool True
| - ⇒ FBool False)
| lderiv0 (bs1, bs2) (InT m M) = (case (bs1 ! m, bs2 ! M) of
  (False, -) ⇒ FBase (InT m M)
| (True, True) ⇒ FBool True
| - ⇒ FBool False)

```

fun rderiv0 **where**

```

  rderiv0 (bs1, bs2) (Q m) = (if bs1 ! m then FBool True else FBase (Q m))
| rderiv0 (bs1, bs2) (Less m1 m2) = (case bs1 ! m2 of
  False ⇒ FBase (Less m1 m2)
| True ⇒ FBase (LessF m1 m2))
| rderiv0 (bs1, bs2) (LessF m1 m2) = (case (bs1 ! m1, bs1 ! m2) of
  (True, False) ⇒ FBase (LessT m1 m2)
| - ⇒ FBase (LessF m1 m2))
| rderiv0 (bs1, bs2) (LessT m1 m2) = (case bs1 ! m2 of
  False ⇒ FBase (LessT m1 m2)
| True ⇒ FBase (LessF m1 m2))
| rderiv0 (bs1, bs2) (In m M) = (case (bs1 ! m, bs2 ! M) of
  (True, True) ⇒ FBase (InT m M)
| - ⇒ FBase (In m M))
| rderiv0 (bs1, bs2) (InT m M) = (case (bs1 ! m, bs2 ! M) of
  (True, False) ⇒ FBase (In m M)
| - ⇒ FBase (InT m M))

```

fun nullable0 **where**

```

  nullable0 (Q m) = False
| nullable0 (Less m1 m2) = False
| nullable0 (LessF m1 m2) = False
| nullable0 (LessT m1 m2) = True
| nullable0 (In m M) = False
| nullable0 (InT m M) = True

```

lift-definition $\sigma :: \text{idx} \Rightarrow \text{atom list}$

is $(\lambda(n, N). \text{map } (\lambda bs. (\text{take } n \text{ } bs, \text{drop } n \text{ } bs)) (\text{List.n-lists } (n + N) [\text{True}, \text{False}])))$.

10 Interpretation

lemma *fMin-fimage-Suc[simp]*: $x \in A \implies fMin (Suc \mid A) = Suc (fMin A)$
by (rule *fMin-eqI*) (auto intro: *fMin-in*)

lemma *fMin-eq-0[simp]*: $0 \in A \implies fMin A = (0 :: nat)$
by (rule *fMin-eqI*) auto

lemma *insert-nth-Cons[simp]*:
 $insert_nth\ i\ x\ (y \# xs) = (case\ i\ of\ 0 \Rightarrow x \# y \# xs \mid Suc\ i \Rightarrow y \# insert_nth\ i\ x\ xs)$
by (cases *i*) *simp-all*

lemma *insert-nth-commute[simp]*:
assumes $j \leq i \leq length\ xs$
shows $insert_nth\ j\ y\ (insert_nth\ i\ x\ xs) = insert_nth\ (Suc\ i)\ x\ (insert_nth\ j\ y\ xs)$
using *assms* **by** (induct *xs* arbitrary: *i j*) (auto *simp* del: *insert-nth-take-drop* *split: nat.splits*)

lemma *SUC-SUC[simp]*: $SUC\ ord\ (SUC\ ord'\ idx) = SUC\ ord'\ (SUC\ ord\ idx)$
by *transfer* (auto *split: order.splits*)

lemma *LESS-SUC[simp]*:
 $LESS\ ord\ 0\ (SUC\ ord\ idx)$
 $LESS\ ord\ (Suc\ l)\ (SUC\ ord\ idx) = LESS\ ord\ l\ idx$
 $ord \neq ord' \implies LESS\ ord\ l\ (SUC\ ord'\ idx) = LESS\ ord\ l\ idx$
 $LESS\ ord\ l\ idx \implies LESS\ ord\ l\ (SUC\ ord'\ idx)$
by (*transfer*, *force split: order.splits*)+

lemma *nvars-Extend[simp]*:
 $\#_V\ (Extend\ ord\ i\ \mathfrak{A}\ P) = SUC\ ord\ (\#_V\ \mathfrak{A})$
by (*transfer*, *force split: order.splits*)

lemma *Length-Extend[simp]*:
 $Length\ (Extend\ k\ i\ \mathfrak{A}\ P) = max\ (Length\ \mathfrak{A})\ (if\ P = \{\mid\} \text{ then } 0 \text{ else } Suc\ (fMax\ P))$
unfolding *max-def* **by** (*split if-splits*, *transfer*) (*force split: order.splits*)

lemma *assigns-Extend[simp]*:
 $LEQ\ ord\ i\ (\#_V\ \mathfrak{A}) \implies m^{Extend\ ord\ i\ \mathfrak{A}\ P\ ord} = (if\ m = i \text{ then } P \text{ else } dec\ i\ m^{\mathfrak{A}\ ord})$
 $ord \neq ord' \implies m^{Extend\ ord\ i\ \mathfrak{A}\ P\ ord'} = m^{\mathfrak{A}\ ord'}$
by (*transfer*, *force simp: dec-def min-def nth-append split: order.splits*)+

lemma *Extend-commute-safe[simp]*:
 $\llbracket j \leq i; LEQ\ ord\ i\ (\#_V\ \mathfrak{A}) \rrbracket \implies$
 $Extend\ ord\ j\ (Extend\ ord\ i\ \mathfrak{A}\ P1)\ P2 = Extend\ ord\ (Suc\ i)\ (Extend\ ord\ j\ \mathfrak{A}\ P2)\ P1$
by (*transfer*,
force simp del: insert-nth-take-drop simp: replicate-add[symmetric] split: or-

der.splits)

lemma *Extend-commute-unsafe*:

ord \neq *ord'* \implies *Extend* *ord* *j* (*Extend* *ord'* *i* \mathfrak{A} *P1*) *P2* = *Extend* *ord'* *i* (*Extend* *ord* *j* \mathfrak{A} *P2*) *P1*

by (*transfer*, *force simp: replicate-add[symmetric]* *split: order.splits*)

lemma *Length-CONS[simp]*:

Length (*CONS* *x* \mathfrak{A}) = *Suc* (*Length* \mathfrak{A})

by (*transfer*, *force split: order.splits*)

lemma *Length-SNOC[simp]*:

Length (*SNOC* *x* \mathfrak{A}) = *Suc* (*Length* \mathfrak{A})

by (*transfer*, *force simp: Let-def split: order.splits*)

lemma *nvars-CONS[simp]*:

$\#_V$ (*CONS* *x* \mathfrak{A}) = $\#_V$ \mathfrak{A}

by (*transfer*, *force*)

lemma *nvars-SNOC[simp]*:

$\#_V$ (*SNOC* *x* \mathfrak{A}) = $\#_V$ \mathfrak{A}

by (*transfer*, *force simp: Let-def*)

lemma *assigns-CONS[simp]*:

assumes $\#_V$ \mathfrak{A} = *size-atom* *bs1-bs2*

shows *LESS* *ord* *x* ($\#_V$ \mathfrak{A}) \implies $x^{CONS\ bs1-bs2\ \mathfrak{A}\ ord} =$

(*if split case-order* *bs1-bs2* *ord* ! *x* *then* *fininsert* 0 (*upshift* ($x^{\mathfrak{A}\ ord}$)) *else* *upshift* ($x^{\mathfrak{A}\ ord}$))

by (*insert assms, transfer*) (*auto simp: lift-def split: order.splits*)

lemma *assigns-SNOC[simp]*:

assumes $\#_V$ \mathfrak{A} = *size-atom* *bs1-bs2*

shows *LESS* *ord* *x* ($\#_V$ \mathfrak{A}) \implies $x^{SNOC\ bs1-bs2\ \mathfrak{A}\ ord} =$

(*if split case-order* *bs1-bs2* *ord* ! *x* *then* *fininsert* (*Length* \mathfrak{A}) ($x^{\mathfrak{A}\ ord}$) *else* $x^{\mathfrak{A}\ ord}$)

by (*insert assms, transfer*) (*force simp: snoc-def Let-def split: order.splits*)

lemma *map-index'-eq-conv[simp]*:

map-index' *i* *f* *xs* = *map-index'* *j* *g* *xs* = ($\forall k < \text{length } xs. f\ (i + k)\ (xs\ !\ k) = g\ (j + k)\ (xs\ !\ k)$)

proof (*induct xs arbitrary: i j*)

case *Cons* **from** *Cons*(1)[*of Suc i Suc j*] **show** ?*case* **by** (*auto simp: nth-Cons split: nat.splits*)

qed *simp*

lemma *fMax-Diff-0[simp]*: *Suc* *m* \in *P* \implies *fMax* (*P* $\mid-$ $\{|0|\}$) = *fMax* *P*

by (*rule fMax-eqI*) (*auto intro: fMax-in dest: fMax-ge*)

lemma *Suc-fMax-pred-fimage[simp]*:

assumes *Suc* *p* \in *P* 0 \notin *P*

shows $Suc (fMax ((\lambda x. x - Suc\ 0) \mid' \mid P)) = fMax\ P$
using *assms* **by** (*subst mono-fMax-commute*[of *Suc*, *unfolded mono-def*, *simplified*]) (*auto simp: o-def*)

lemma *Extend-CONS*[*simp*]: $\#_V \mathfrak{A} = size\text{-}atom\ x \implies Extend\ ord\ 0\ (CONS\ x\ \mathfrak{A})$
 $P =$
 $CONS\ (extend\ ord\ (eval\ P\ 0)\ x)\ (Extend\ ord\ 0\ \mathfrak{A}\ (downshift\ P))$
by *transfer* (*auto simp: extend-def o-def gr0-conv-Suc*
mono-fMax-commute[of *Suc*, *symmetric*, *unfolded mono-def*, *simplified*]
lift-def upshift-def downshift-def eval-def
dest!: fsubset-fsingletonD split: order.splits)

lemma *size-atom-extend*[*simp*]:
 $size\text{-}atom\ (extend\ ord\ b\ x) = SUC\ ord\ (size\text{-}atom\ x)$
unfolding *extend-def* **by** *transfer* (*simp split: prod.splits order.splits*)

lemma *size-atom-zero*[*simp*]:
 $size\text{-}atom\ (zero\ idx) = idx$
unfolding *extend-def* **by** *transfer* (*simp split: prod.splits order.splits*)

lemma *interp-eqI*:
 $\llbracket Length\ \mathfrak{A} = Length\ \mathfrak{B}; \#_V \mathfrak{A} = \#_V \mathfrak{B}; \bigwedge m\ k. LESS\ k\ m\ (\#_V \mathfrak{A}) \implies m^{\mathfrak{A}}k = m^{\mathfrak{B}}k \rrbracket \implies \mathfrak{A} = \mathfrak{B}$
by *transfer* (*force split: order.splits intro!: nth-equalityI*)

lemma *Extend-SNOC-cut*[*unfolded eval-def cut-def Length-SNOC, simp*]:
 $\llbracket len\ P \leq Length\ (SNOC\ x\ \mathfrak{A}); \#_V \mathfrak{A} = size\text{-}atom\ x \rrbracket \implies$
 $Extend\ ord\ 0\ (SNOC\ x\ \mathfrak{A})\ P =$
 $SNOC\ (extend\ ord\ (if\ eval\ P\ (Length\ \mathfrak{A})\ then\ True\ else\ False)\ x)\ (Extend\ ord\ 0\ \mathfrak{A}\ (cut\ (Length\ \mathfrak{A})\ P))$
by *transfer* (*fastforce simp: extend-def len-def cut-def ffilter-eq-fempty-iff snoc-def eval-def*
split: if-splits order.splits dest: fMax-ge fMax-boundedD intro: fMax-in)

lemma *nth-replicate-simp*: $replicate\ m\ x\ !\ i = (if\ i < m\ then\ x\ else\ []\ !\ (i - m))$
by (*induct m arbitrary: i*) (*auto simp: nth-Cons'*)

lemma *MSB-eq-SucD*: $MSB\ Is = Suc\ x \implies \exists P \in set\ Is. x \in P$
using *Max-in*[of $\bigcup x \in set\ Is. fset\ x$]
unfolding *MSB-def* **by** (*force simp: fmember-def split: if-splits*)

lemma *last-enc-atom-MSB*:
fixes *I1 I2*
defines $xs \equiv map\ (enc\text{-}atom\ I1\ I2)\ [0..<MSB\ (I1\ @\ I2)]$
assumes $m = length\ I1 = length\ I2\ xs \neq []$
shows $last\ xs \neq (replicate\ m\ False, replicate\ n\ False)$
proof (*rule ccontr, unfold not-not*)
assume $last\ xs = (replicate\ m\ False, replicate\ n\ False)$
moreover from $\langle xs \neq [] \rangle$ **obtain** *i* **where** $i: MSB\ (I1\ @\ I2) = Suc\ i$

unfolding $xs\text{-def}$ **by** (auto simp: gr0-conv-Suc)
ultimately have $enc\text{-atom } I1 \ I2 \ i = (\text{replicate } m \ \text{False}, \text{replicate } n \ \text{False})$
unfolding $xs\text{-def}$ **by** auto
with i **show** False
by (auto simp: list-eq-iff-nth-eq max-def in-set-conv-nth dest!: MSB-eq-SucD
split: if-splits)
qed

lemma $append\text{-replicate-inj}$:

assumes $xs \neq [] \implies last \ xs \neq x$ **and** $ys \neq [] \implies last \ ys \neq x$
shows $xs @ replicate \ m \ x = ys @ replicate \ n \ x \longleftrightarrow (xs = ys \wedge m = n)$
proof safe
from $assms$ **have** $assms'$: $xs \neq [] \implies rev \ xs \ ! \ 0 \neq x$ $ys \neq [] \implies rev \ ys \ ! \ 0 \neq x$
by (auto simp: hd-rev hd-conv-nth[symmetric])
assume *: $xs @ replicate \ m \ x = ys @ replicate \ n \ x$
then have $rev \ (xs @ replicate \ m \ x) = rev \ (ys @ replicate \ n \ x)$..
then have $replicate \ m \ x @ rev \ xs = replicate \ n \ x @ rev \ ys$ **by** simp
then have $take \ (max \ m \ n) \ (replicate \ m \ x @ rev \ xs) = take \ (max \ m \ n) \ (replicate \ n \ x @ rev \ ys)$ **by** simp
then have $replicate \ m \ x @ take \ (max \ m \ n - m) \ (rev \ xs) =$
 $replicate \ n \ x @ take \ (max \ m \ n - n) \ (rev \ ys)$ **by** (auto simp: min-def max-def
split: if-splits)
then have $(replicate \ m \ x @ take \ (max \ m \ n - m) \ (rev \ xs)) \ ! \ min \ m \ n =$
 $(replicate \ n \ x @ take \ (max \ m \ n - n) \ (rev \ ys)) \ ! \ min \ m \ n$ **by** simp
with $arg\text{-cong}[OF \ *, \ of \ length, \ simplified]$ $assms'$ **show** $m = n$
by (cases $xs = [] \ ys = []$ rule: bool.exhaust[case-product bool.exhaust])
(auto simp: min-def nth-append split: if-splits)
with * **show** $xs = ys$ **by** auto
qed

lemma $enc\text{-inj}[simp]$: $\#_V \ \mathfrak{A} = \#_V \ \mathfrak{B} \implies enc \ \mathfrak{A} = enc \ \mathfrak{B} \implies \mathfrak{A} = \mathfrak{B}$

using MSB-greater
by transfer (elim exE conjE, hypsubst, unfold prod.case Pair-eq Let-def, elim
conjE, simp only;
subst (asm) append-replicate-inj,
erule (2) last-enc-atom-MSB[OF sym sym], erule last-enc-atom-MSB[OF refl
refl],
auto simp: list-eq-iff-nth-eq, (metis less-max-iff-disj)+)

lemma $length\text{-enc}[simp]$: $length \ (enc \ \mathfrak{A}) = Length \ \mathfrak{A}$

by transfer (auto simp: Let-def)

lemma $in\text{-set-encD}[simp]$:

$x \in set \ (enc \ \mathfrak{A}) \implies \#_V \ \mathfrak{A} = size\text{-atom } x$
by transfer (auto simp: Let-def split: if-splits)

lemma $fin\text{-lift}[simp]$: $m \in | \ lift \ bs \ i \ (I \ ! \ i) \longleftrightarrow (case \ m \ of \ 0 \Rightarrow bs \ ! \ i \mid Suc \ m \Rightarrow m \in | \ I \ ! \ i)$

unfolding lift-def upshift-def **by** (auto split: nat.splits)

lemma *enc-CONS*[simp]:
 assumes $\#_V \mathfrak{A} = \text{size-atom } x$
 shows $\text{enc } (\text{CONS } x \ \mathfrak{A}) = x \ \# \ \text{enc } \mathfrak{A}$
 using *assms* by transfer (fastforce simp add: Let-def nth-append nth-Cons
 simp del: upt-conv-Nil diff-is-0-eq' atLeastLessThan-empty
 split: prod.splits nat.splits intro!: nth-equalityI)

lemma *ex-Length-0*[simp]:
 $\exists \mathfrak{A}. \text{Length } \mathfrak{A} = 0 \wedge \#_V \mathfrak{A} = \text{id}x$
 by transfer (auto intro!: exI[of - replicate m {||}] for m)

lemma *is-empty-inj*[simp]: $[\text{Length } \mathfrak{A} = 0; \text{Length } \mathfrak{B} = 0; \#_V \mathfrak{A} = \#_V \mathfrak{B}] \implies \mathfrak{A} = \mathfrak{B}$
 by transfer (simp add: list-eq-iff-nth-eq split: prod.splits,
 metis MSB-greater fMax-in less-nat-zero-code)

lemma *set-σ-length-atom*[simp]: $(x \in \text{set } (\sigma \text{id}x)) \longleftrightarrow \text{id}x = \text{size-atom } x$
 by transfer (auto simp: set-n-lists enum-UNIV image-iff intro!: exI[of - I1 @ I2
 for I1 I2])

lemma *fMin-less-Length*[simp]: $x \in | \ m1^{\mathfrak{A}}k \implies \text{fMin } (m1^{\mathfrak{A}}k) < \text{Length } \mathfrak{A}$
 by transfer
 (force elim: order.strict-trans2[OF MSB-greater, rotated -1] intro: fMin-in
 split: order.splits)

lemma *min-Length-fMin*[simp]: $x \in | \ m1^{\mathfrak{A}}k \implies \min (\text{Length } \mathfrak{A}) (\text{fMin } (m1^{\mathfrak{A}}k))$
 $= \text{fMin } (m1^{\mathfrak{A}}k)$
 using *fMin-less-Length*[of $x \ m1 \ \mathfrak{A} \ k$] unfolding *fMin-def* by auto

lemma *assigns-less-Length*[simp]: $x \in | \ m1^{\mathfrak{A}}k \implies x < \text{Length } \mathfrak{A}$
 by transfer (force dest: MSB-greater split: order.splits if-splits)

lemma *Length-notin-assigns*[simp]: $\text{Length } \mathfrak{A} \notin | \ m^{\mathfrak{A}}k$
 by (metis assigns-less-Length less-not-refl)

lemma *nth-zero*[simp]: $\text{LESS } \text{ord } m (\#_V \mathfrak{A}) \implies \neg \text{split case-order } (\text{zero } (\#_V \mathfrak{A}))$
 $\text{ord } ! \ m$
 by transfer (auto split: order.splits)

lemma *in-fimage-Suc*[simp]: $x \in | \ \text{Suc } |' \ A \longleftrightarrow (\exists y. y \in | \ A \wedge x = \text{Suc } y)$
 by blast

lemma *fimage-Suc-inj*[simp]: $\text{Suc } |' \ A = \text{Suc } |' \ B \longleftrightarrow A = B$
 by blast

lemma *MSB-eq0-D*: $\text{MSB } I = 0 \implies x < \text{length } I \implies I ! x = \{||\}$
 unfolding *MSB-def* by (auto split: if-splits)

lemma *Suc-in-fimage-Suc*: $Suc\ x \mid\in\ Suc\ \mid'\ X \longleftrightarrow x \mid\in\ X$
by *auto*

lemma *Suc-in-fimage-Suc-o-Suc[simp]*: $Suc\ x \mid\in\ (Suc\ o\ Suc)\ \mid'\ X \longleftrightarrow x \mid\in\ Suc\ \mid'\ X$
by *auto*

lemma *finsert-same-eq-iff[simp]*: $finsert\ k\ X = finsert\ k\ Y \longleftrightarrow X \mid-\mid\ \{|k|\} = Y \mid-\mid\ \{|k|\}$
by *auto*

lemma *fimage-Suc-o-Suc-eq-fimage-Suc-iff[simp]*:
 $((Suc\ o\ Suc)\ \mid'\ X = Suc\ \mid'\ Y) \longleftrightarrow (Suc\ \mid'\ X = Y)$
by $(metis\ fimage-Suc-inj\ fset.map-comp)$

lemma *fMax-image-Suc[simp]*: $x \mid\in\ P \implies fMax\ (Suc\ \mid'\ P) = Suc\ (fMax\ P)$
by $(rule\ fMax-eqI)\ (metis\ Suc-le-mono\ fMax-ge\ fimageE,\ metis\ fimageI\ fempty-iff\ fMax-in)$

lemma *len-downshift-helper*:

$x \mid\in\ P \implies Suc\ (fMax\ ((\lambda x. x - Suc\ 0)\ \mid'\ (P \mid-\mid\ \{|0|\}))) \neq fMax\ P \implies xa \mid\in\ P \implies xa = 0$

proof –

assume *a1*: $xa \mid\in\ P$

assume *a2*: $Suc\ (fMax\ ((\lambda x. x - Suc\ 0)\ \mid'\ (P \mid-\mid\ \{|0|\}))) \neq fMax\ P$

have $xa \mid\in\ \{|0|\} \longrightarrow xa = 0$ **by** *fastforce*

moreover

$\{ \text{assume } xa \mid\notin\ \{|0|\}$

hence $0 \mid\notin\ P \mid-\mid\ \{|0|\} \wedge xa \mid\notin\ \{|0|\}$ **by** *blast*

then obtain $esk1_1 :: nat \Rightarrow nat$ **where** $xa = 0$ **using** *a1 a2* **by** $(metis\ Suc-fMax-pred-fimage\ fMax-Diff-0\ fminus-iff\ not0-implies-Suc)$

ultimately show $xa = 0$ **by** *blast*

qed

declare $[[goals-limit = 50]]$

definition *restrict ord* $P = (case\ ord\ of\ FO \Rightarrow P \neq \{||\} \mid SO \Rightarrow True)$

definition *Restrict ord* $i = (case\ ord\ of\ FO \Rightarrow FBase\ (Q\ i) \mid SO \Rightarrow FBool\ True)$

permanent-interpretation *WS1S*: *Word-Formula SUC LESS assigns nvars Extend CONS SNOC Length*

extend size-atom zero eval downshift upshift finsert cut len restrict Restrict

left-formula0 FV0 find0 wf0 decr0 satisfies0 nullable0 lderiv0 rderiv0 undefined enc σ ZERO

defining $norm = Formula-Operations.norm\ find0\ decr0$

and $nFOr = Formula-Operations.nFOr :: formula \Rightarrow -$

and $nFAnd = Formula-Operations.nFAnd :: formula \Rightarrow -$

and $nFNot = Formula-Operations.nFNot :: formula \Rightarrow -$

```

and nFEx = Formula-Operations.nFEx find0 decr0
and nFAll = Formula-Operations.nFAll find0 decr0
and decr = Formula-Operations.decr decr0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula  $\Rightarrow$  -
and find = Formula-Operations.find find0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula  $\Rightarrow$  -
and FV = Formula-Operations.FV FV0
and RESTR = Formula-Operations.RESTR Restrict
and RESTRICT = Formula-Operations.RESTRICK Restrict FV0
and deriv =  $\lambda d0$  (a :: atom) ( $\varphi$  :: formula). Formula-Operations.deriv extend d0
a  $\varphi$ 
and nullable =  $\lambda \varphi$  :: formula. Formula-Operations.nullable nullable0  $\varphi$ 
and fut-default = Formula.fut-default extend zero rderiv0
and fut = Formula.fut extend zero find0 decr0 rderiv0
and finalize = Formula.finalize SUC extend zero find0 decr0 rderiv0
and final = Formula.final SUC extend zero find0 decr0
      nullable0 rderiv0 :: idx  $\Rightarrow$  formula  $\Rightarrow$  -
and ws1s-wf = Formula-Operations.wf SUC (wf0 :: idx  $\Rightarrow$  ws1s  $\Rightarrow$  -)
and ws1s-left-formula = Formula-Operations.left-formula left-formula0 :: formula
 $\Rightarrow$  -
and check-equiv =  $\lambda idx$ . DA.check-equiv ( $\sigma$  idx)
      ( $\lambda \varphi$ . norm (RESTRICT  $\varphi$ ) :: (ws1s, order) aformula)
      ( $\lambda a$   $\varphi$ . norm (deriv (lderiv0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula) (a :: atom)  $\varphi$ ))
      (final idx) ( $\lambda \varphi$  :: formula. ws1s-wf idx  $\varphi \wedge$  ws1s-left-formula  $\varphi$ )
and bounded-check-equiv =  $\lambda idx$ . DA.check-equiv ( $\sigma$  idx)
      ( $\lambda \varphi$ . norm (RESTRICT  $\varphi$ ) :: (ws1s, order) aformula)
      ( $\lambda a$   $\varphi$ . norm (deriv (lderiv0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula) (a :: atom)  $\varphi$ ))
      nullable ( $\lambda \varphi$  :: formula. ws1s-wf idx  $\varphi \wedge$  ws1s-left-formula  $\varphi$ )
and automaton = DA.automaton
      ( $\lambda a$   $\varphi$ . norm (deriv lderiv0 (a :: atom)  $\varphi$  :: formula))
proof
  fix k idx and a :: ws1s and l assume wf0 (SUC k idx) a LESS k l (SUC k idx)
   $\neg$  find0 k l a
  then show wf0 idx (decr0 k l a)
    by (induct a) (unfold wf0.simps decr0.simps find0.simps,
      (transfer, force simp: dec-def split: if-splits order.splits)+)
next
  fix k and a :: ws1s and l assume left-formula0 a
  then show left-formula0 (decr0 k l a) by (induct a) simp-all
next
  fix i k and a :: ws1s and  $\mathfrak{A}$  :: interp and P assume  $\neg$  find0 k i a LESS k i
  (SUC k ( $\#_V$   $\mathfrak{A}$ ))
  then show satisfies0 (Extend k i  $\mathfrak{A}$  P) a = satisfies0  $\mathfrak{A}$  (decr0 k i a)
    by (induct a) (auto split: if-splits order.splits)
next
  fix idx and a :: ws1s and x assume wf0 idx a
  then show Formula-Operations.wf SUC wf0 idx (lderiv0 x a)
    by (induct rule: lderiv0.induct)
    (auto simp: Formula-Operations.wf.simps Let-def split: bool.splits order.splits)
next
  fix a :: ws1s and x assume left-formula0 a

```

```

then show Formula-Operations.left-formula left-formula0 (lderiv0 x a)
  by (induct a rule: lderiv0.induct)
    (auto simp: Formula-Operations.left-formula.simps split: bool.splits)
next
  fix idx and a :: ws1s and x assume wf0 idx a
  then show Formula-Operations.wf SUC wf0 idx (rderiv0 x a)
    by (induct rule: lderiv0.induct)
      (auto simp: Formula-Operations.wf.simps Let-def split: bool.splits order.splits)
next
  fix  $\mathfrak{A}$  :: interp and a :: ws1s
  assume Length  $\mathfrak{A}$  = 0
  then show nullable0 a = satisfies0  $\mathfrak{A}$  a
    by (induct a, unfold wf0.simps nullable0.simps satisfies0.simps Let-def)
      (transfer, (auto 0 2 dest: MSB-greater split: prod.splits if-splits) [])+
next
  note Formula-Operations.satisfies-gen.simps[simp] Let-def[simp] upshift-def[simp]
  fix x :: atom and a :: ws1s and  $\mathfrak{A}$  :: interp
  assume wf0 ( $\#_V \mathfrak{A}$ ) a  $\#_V \mathfrak{A}$  = size-atom x
  then show Formula-Operations.satisfies Extend Length satisfies0  $\mathfrak{A}$  (lderiv0 x
a) = satisfies0 (CONS x  $\mathfrak{A}$ ) a
    proof (induct a)
      qed (auto split: prod.splits bool.splits)
next
  note Formula-Operations.satisfies-gen.simps[simp] Let-def[simp] upshift-def[simp]
  fix x :: atom and a :: ws1s and  $\mathfrak{A}$  :: interp
  assume wf0 ( $\#_V \mathfrak{A}$ ) a  $\#_V \mathfrak{A}$  = size-atom x
  then show Formula-Operations.satisfies-bounded Extend Length len satisfies0  $\mathfrak{A}$ 
(lderiv0 x a) = satisfies0 (CONS x  $\mathfrak{A}$ ) a
    proof (induct a)
      qed (auto split: prod.splits bool.splits)
next
  note Formula-Operations.satisfies-gen.simps[simp] Let-def[simp]
  fix x :: atom and a :: ws1s and  $\mathfrak{A}$  :: interp
  assume wf0 ( $\#_V \mathfrak{A}$ ) a  $\#_V \mathfrak{A}$  = size-atom x
  then show Formula-Operations.satisfies-bounded Extend Length len satisfies0
 $\mathfrak{A}$  (rderiv0 x a) = satisfies0 (SNOC x  $\mathfrak{A}$ ) a
    proof (induct a)
      case Less then show ?case by (auto split: prod.splits) (metis fMin-less-Length
less-not-sym)
    next
      case LessT then show ?case by (auto split: prod.splits) (metis fMin-less-Length
less-not-sym)
    next
      case LessF then show ?case by (auto split: prod.splits) (metis fMin-less-Length
less-not-sym)
    next
      case In then show ?case by (auto split: prod.splits) (metis fMin-less-Length
less-not-sym)+
    next

```

```

    case InT then show ?case by (auto split: prod.splits) (metis fMin-less-Length
less-not-sym)+
    qed (auto split: prod.splits)
next
  fix a :: ws1s and  $\mathfrak{A} \ \mathfrak{B} :: \text{interp}$ 
  assume wf0 ( $\#_V \ \mathfrak{B}$ ) a  $\#_V \ \mathfrak{A} = \#_V \ \mathfrak{B} \ (\bigwedge m \ k. \text{LESS } k \ m \ (\#_V \ \mathfrak{B}) \implies m^{\mathfrak{A}}_k = m^{\mathfrak{B}}_k)$ 
  left-formula0 a
  then show satisfies0  $\mathfrak{A} \ a \longleftrightarrow \text{satisfies0 } \mathfrak{B} \ a$  by (induct a) simp-all
next
  fix a :: ws1s
  let ?d = Formula-Operations.deriv extend lderiv0
  def  $\Phi \equiv \lambda a. (\text{case } a \text{ of}$ 
    |  $Q \ i \Rightarrow \{FBase \ (Q \ i), FBool \ True\}$ 
    |  $Less \ i \ j \Rightarrow \{FBase \ (Less \ i \ j), FBase \ (Q \ j), FBool \ True, FBool \ False\}$ 
    |  $LessT \ i \ j \Rightarrow \{FBase \ (LessT \ i \ j), FBool \ True, FBool \ False\}$ 
    |  $LessF \ i \ j \Rightarrow \{FBase \ (LessF \ i \ j), FBool \ True, FBool \ False\}$ 
    |  $In \ i \ I \Rightarrow \{FBase \ (In \ i \ I), FBool \ True, FBool \ False\}$ 
    |  $InT \ i \ I \Rightarrow \{FBase \ (InT \ i \ I), FBool \ True, FBool \ False\} :: (ws1s, order)$ 
  aformula set
  { fix xs
    note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
   $\Phi$ -def[simp]
    have  $\forall a. \text{fold } ?d \ xs \ (FBase \ a) \in \Phi \ a$ 
    by (induct xs) (auto split: ws1s.splits bool.splits if-splits, metis+)
  }
  moreover have finite ( $\Phi \ a$ ) unfolding  $\Phi$ -def by (auto split: ws1s.splits)
  ultimately show finite {fold ?d xs (FBase a) | xs. True}
  by (blast intro: finite-subset)
next
  fix a :: ws1s
  let ?d = Formula-Operations.deriv extend rderiv0
  def  $\Phi \equiv \lambda a. (\text{case } a \text{ of}$ 
    |  $Q \ i \Rightarrow \{FBase \ (Q \ i), FBool \ True\}$ 
    |  $Less \ i \ j \Rightarrow \{FBase \ (Less \ i \ j), FBase \ (LessF \ i \ j), FBase \ (LessT \ i \ j)\}$ 
    |  $LessT \ i \ j \Rightarrow \{FBase \ (LessT \ i \ j), FBase \ (LessF \ i \ j)\}$ 
    |  $LessF \ i \ j \Rightarrow \{FBase \ (LessF \ i \ j), FBase \ (LessT \ i \ j)\}$ 
    |  $In \ i \ I \Rightarrow \{FBase \ (In \ i \ I), FBase \ (InT \ i \ I)\}$ 
    |  $InT \ i \ I \Rightarrow \{FBase \ (InT \ i \ I), FBase \ (In \ i \ I)\} :: (ws1s, order)$ 
  aformula set
  { fix x xs
    note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
   $\Phi$ -def[simp]
    have  $\forall a. \text{fold } ?d \ xs \ (FBase \ a) \in \Phi \ a$ 
    by (induct xs) (auto split: ws1s.splits bool.splits if-splits, metis+)
  }
  moreover have finite ( $\Phi \ a$ ) unfolding  $\Phi$ -def by (auto split: ws1s.splits)
  ultimately show finite {fold ?d xs (FBase a) | xs. True}
  by (blast intro: finite-subset)
next

```



```

    fix k l and a :: ws1s
    show find0 k l a  $\longleftrightarrow$  (k, l)  $\in$  set (FV0 a) by (induct a rule: find0.induct) auto
next
    fix a :: ws1s
    show distinct (FV0 a) by (induct a) auto
next
    fix idx a k v
    assume wf0 idx a (k, v)  $\in$  set (FV0 a)
    then show LESS k v idx by (induct a) auto
next
    fix idx k i
    assume LESS k i idx
    then show Formula-Operations.wf SUC wf0 idx (Restrict k i)
        unfolding Restrict-def by (cases k) (auto simp: Formula-Operations.wf.simps)
next
    fix k i
    show Formula-Operations.left-formula left-formula0 (Restrict k i)
        unfolding Restrict-def by (cases k) (auto simp: Formula-Operations.left-formula.simps)
next
    fix i  $\mathfrak{A}$  k P
    assume  $i^{\mathfrak{A}}_k = P$ 
    then show restrict k P  $\longleftrightarrow$ 
        Formula-Operations.satisfies-gen Extend Length satisfies0 ( $\lambda$ - - . True)  $\mathfrak{A}$ 
        (Restrict k i)
        unfolding restrict-def Restrict-def
        by (cases k) (auto simp: Formula-Operations.satisfies-gen.simps)
next
    fix i  $\mathfrak{A}$  k P
    assume  $i^{\mathfrak{A}}_k = P$ 
    then show restrict k P  $\longleftrightarrow$ 
        Formula-Operations.satisfies-gen Extend Length satisfies0 ( $\lambda$ - P n. len P  $\leq$  n)
 $\mathfrak{A}$  (Restrict k i)
        unfolding restrict-def Restrict-def
        by (cases k) (auto simp: Formula-Operations.satisfies-gen.simps)
qed (auto simp: Extend-commute-unsafe downshift-def upshift-def fimage-iff Suc-le-eq
len-def
eval-def cut-def len-downshift-helper
dest: fMax-ge fMax-ffilter-less fMax-boundedD fsubset-fsingletonD
split: order.splits if-splits)

```

lemma [code]: check-equiv idx r s =
 $((ws1s\text{-}wf\ idx\ r \wedge ws1s\text{-}left\text{-}formula\ r) \wedge (ws1s\text{-}wf\ idx\ s \wedge ws1s\text{-}left\text{-}formula\ s) \wedge$
 $(case\ rtranc1\text{-}while\ (\lambda(p, q). final\ idx\ p = final\ idx\ q)$
 $(\lambda(p, q). map\ (\lambda a. (norm\ (deriv\ lderiv0\ a\ p), norm\ (deriv\ lderiv0\ a\ q))))\ (\sigma$
 $idx))$
 $(norm\ (RESTRICT\ r), norm\ (RESTRICT\ s))\ of$
 $None \Rightarrow False$
 $| Some\ ([], x) \Rightarrow True$

| *Some* (*a* # *list*, *x*) \Rightarrow *False*)
unfolding *check-equiv-def* *WS1S.check-equiv-def* ..

definition *while where* [*code del*, *code-abbrev*]: *while idx* φ = *while-default* (*while-default* *idx* φ)
declare *while-default-code*[*of* *while-default idx* φ **for** *idx* φ , *folded while-def*, *code*]

export-code *check-equiv* **in** *SML module-name* *WS1S-Generated*

lemma *check-equiv-sound*:
 $\llbracket \#_V \mathfrak{A} = \text{idx}; \text{check-equiv idx } \varphi \ \psi \rrbracket \Rightarrow (WS1S.sat \ \mathfrak{A} \ \varphi \longleftrightarrow WS1S.sat \ \mathfrak{A} \ \psi)$
unfolding *check-equiv-def* **by** (*rule* *WS1S.check-equiv-soundness*)

lemma *bounded-check-equiv-sound*:
 $\llbracket \#_V \mathfrak{A} = \text{idx}; \text{bounded-check-equiv idx } \varphi \ \psi \rrbracket \Rightarrow (WS1S.sat_b \ \mathfrak{A} \ \varphi \longleftrightarrow WS1S.sat_b \ \mathfrak{A} \ \psi)$
unfolding *bounded-check-equiv-def* **by** (*rule* *WS1S.bounded-check-equiv-soundness*)

end

11 Examples

abbreviation (*input*) *FImp* **where** *FImp* $\varphi \ \psi \equiv FOr \ (FNot \ \varphi) \ \psi$
abbreviation *FIff* **where** *FIff* $\varphi \ \psi \equiv FAnd \ (FImp \ \varphi \ \psi) \ (FImp \ \psi \ \varphi)$
abbreviation *FBall* **where** *FBall* $X \ \varphi \equiv FAll \ FO \ (FImp \ (FBase \ (In \ 0 \ X)) \ \varphi)$

abbreviation *SUBSET* **where** *SUBSET* $X \ Y \equiv FBall \ X \ (FBase \ (In \ 0 \ Y))$
abbreviation *EQ* **where** *EQ* $X \ Y \equiv FAnd \ (SUBSET \ X \ Y) \ (SUBSET \ Y \ X)$
abbreviation *First* **where** *First* $x \equiv FNot \ (FEx \ FO \ (FBase \ (Less \ 0 \ (x+1))))$
abbreviation *Last* **where** *Last* $x \equiv FNot \ (FEx \ FO \ (FBase \ (Less \ (x+1) \ 0)))$
abbreviation *Succ* **where** *Succ* $sucx \ x \equiv FAnd \ (FBase \ (Less \ x \ sucx)) \ (FNot \ (FEx \ FO \ (FAnd \ (FBase \ (Less \ (x+1) \ 0)) \ (FBase \ (Less \ 0 \ (sucx+1))))))$

definition *Thm* *idx* φ = *check-equiv idx* $\varphi \ (FBool \ True)$

export-code *Thm* **in** *SML module-name* *Thm*

abbreviation *FTrue* (\top) **where** *FTrue* $\equiv FBool \ True$
abbreviation *FFalse* (\perp) **where** *FFalse* $\equiv FBool \ False$

notation *FImp* (**infixr** $-->$ 25)
notation (**output**) *FO* (1)
notation (**output**) *SO* (2)
notation *FEx* (\exists - [10] 10)
notation *FAll* (\forall - [10] 10)
notation *FNot* (\neg - [40] 40)
notation *FOr* (**infixr** \vee 30)
notation *FAnd* (**infixr** \wedge 35)
abbreviation *FLess* (x < x [65, 66] 65) **where** *FLess* $m1 \ m2 \equiv FBase \ (Less$

$m1\ m2)$

abbreviation $FIn\ (x_- \in X_- [65, 66]\ 65)$ **where** $FIn\ m\ M \equiv FBase\ (In\ m\ M)$

abbreviation $FQ\ ([x_-]\ [66]\ 65)$ **where** $FQ\ m \equiv FBase\ (Q\ m)$

definition $M2L = (FEx\ SO\ (FAll\ FO\ (FBase\ (In\ 0\ 0))) :: formula)$

definition $\Phi = (FAll\ FO\ (FEx\ FO\ (FBase\ (Less\ 1\ 0))) :: formula)$

lemma $Thm\ (Abs-idx\ (0, 1))\ (FNot\ M2L)$
by *eval*

lemma $Thm\ (Abs-idx\ (0, 0))\ \Phi$
by *eval*

abbreviation $Globally\ (\Box\ [40]\ 40)$ **where** $Globally\ P == \%n.\ FAll\ FO\ (FImp\ (FNot\ (FBase\ (Less\ (n+1)\ 0)))\ (P\ 0))$

abbreviation $Future\ (\Diamond\ [40]\ 40)$ **where** $Future\ P == \%n.\ FEx\ FO\ (FAnd\ (FNot\ (FBase\ (Less\ (n+1)\ 0)))\ (P\ 0))$

abbreviation $IMP\ (infixr\ \rightarrow\ 50)$ **where** $IMP\ P1\ P2 == \%n.\ FImp\ (P1\ n)\ (P2\ n)$

definition $\Psi :: nat \Rightarrow formula$ **where**

$\Psi\ n = FAll\ FO\ (((\Box(foldr\ (\lambda i\ \varphi.\ (\lambda m.\ FBase\ (In\ m\ i)) \rightarrow \varphi)\ [0..<n]\ (\lambda m.\ FBase\ (In\ m\ n)))) \rightarrow$
 $foldr\ (\lambda i\ \varphi.\ (\Box(\lambda m.\ FBase\ (In\ m\ i)) \rightarrow \varphi)\ [0..<n]\ (\Box(\lambda m.\ FBase\ (In\ m\ n))))\ 0)$

lemma $Thm\ (Abs-idx\ (0, 1))\ (\Psi\ 0)$ **by** *eval*

lemma $Thm\ (Abs-idx\ (0, 2))\ (\Psi\ 1)$ **by** *eval*

lemma $Thm\ (Abs-idx\ (0, 3))\ (\Psi\ 2)$ **by** *eval*

lemma $Thm\ (Abs-idx\ (0, 4))\ (\Psi\ 3)$ **by** *eval*

lemma $Thm\ (Abs-idx\ (0, 5))\ (\Psi\ 4)$ **by** *eval*

lemma $Thm\ (Abs-idx\ (0, 6))\ (\Psi\ 5)$ **by** *eval*

lemma $Thm\ (Abs-idx\ (0, 11))\ (\Psi\ 10)$ **by** *eval*

lemma $Thm\ (Abs-idx\ (0, 16))\ (\Psi\ 15)$ **by** *eval*