

# Derivatives of Logical Formulas

Dmitriy Traytel

October 13, 2025

## Abstract

We formalize new decision procedures for WS1S, M2L(Str), and Presburger Arithmetics. Formulas of these logics denote regular languages. Unlike traditional decision procedures, we do *not* translate formulas into automata (nor into regular expressions), at least not explicitly. Instead we devise notions of derivatives (inspired by Brzozowski derivatives for regular expressions) that operate on formulas directly and compute a syntactic bisimulation using these derivatives. The treatment of Boolean connectives and quantifiers is uniform for all mentioned logics and is abstracted into a locale. This locale is then instantiated by different atomic formulas and their derivatives (which may differ even for the same logic under different encodings of interpretations as formal words).

The WS1S instance is described in the draft paper *A Coalgebraic Decision Procedure for WS1S*<sup>1</sup> by the author.

## Contents

<b>1</b>	<b>Equivalence Framework</b>	<b>1</b>
1.1	Abstract Deterministic Automaton . . . . .	4
1.2	The overall procedure . . . . .	6
1.3	Abstract Deterministic Finite Automaton . . . . .	7
<b>2</b>	<b>Derivatives of Abstract Formulas</b>	<b>9</b>
2.1	Preliminaries . . . . .	9
2.2	Abstract formulas . . . . .	10
2.3	Normalization . . . . .	23
2.4	Derivatives of Formulas . . . . .	25
2.5	Finiteness of Derivatives Modulo ACI . . . . .	27
2.6	Emptiness Check . . . . .	30
2.7	Restrictions . . . . .	33
<b>3</b>	<b>WS1S Interpretations</b>	<b>36</b>

---

<sup>1</sup>[http://www21.in.tum.de/~traytel/papers/ws1s\\_derivatives/index.html](http://www21.in.tum.de/~traytel/papers/ws1s_derivatives/index.html)

4	Concrete Atomic WS1S Formulas (Minimum Semantics for FO Variables)	46
5	Concrete Atomic WS1S Formulas (Singleton Semantics for FO Variables)	63
6	Concrete Atomic Presburger Formulas	71
7	Comparing WS1S Formulas with Presburger Formulas	86
8	Nameful WS1S Formulas	88

## 1 Equivalence Framework

**coinductive** *rel-language* **where**

$\llbracket \circ L = \circ K; \bigwedge a b. R a b \implies \text{rel-language } R (\mathfrak{d} L a) (\mathfrak{d} K b) \rrbracket \implies \text{rel-language } R L K$

**declare** *rel-language.coinduct*[*consumes 1, case-names Lang, coinduct pred*]

**lemma** *rel-language-alt*:

*rel-language*  $R L K = \text{rel-fun } (\text{list-all2 } R) (=) (\text{in-language } L) (\text{in-language } K)$

**unfolding** *rel-fun-def* **proof** (*rule iffI, safe del: iffI*)

**fix** *xs ys*

**assume** *list-all2 R xs ys rel-language R L K*

**then show** *in-language L xs = in-language K ys*

**by** (*induct xs ys arbitrary: L K (auto del: iffI elim: rel-language.cases)*)

**next**

**assume**  $\forall xs ys. \text{list-all2 } R xs ys \longrightarrow \text{in-language } L xs = \text{in-language } K ys$

**then show** *rel-language R L K* **by** (*coinduction arbitrary: L K (auto dest: spec2)*)

**qed**

**lemma** *rel-language-eq*: *rel-language* (=) = (=)

**unfolding** *rel-language-alt*[*abs-def*] *list.rel-eq fun.rel-eq*

**by** (*subst (2) fun-eq-iff*)<sup>+</sup>

(*auto intro: box-equals*[*OF - to-language-in-language to-language-in-language*])

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

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

**by** (*induct w arbitrary: L simp-all*)

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

**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**  $\{((r, s), (f a r, g b s)) \mid a b r s. a \in A \wedge b \in B \wedge R a b\}^{\widehat{*}} =$   
 $\{((r, s), (fold f w1 r, fold g w2 s)) \mid w1 w2 r s. w1 \in lists A \wedge w2 \in lists B$   
 $\wedge list-all2 R w1 w2\}$   
**(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
}
```

**hence**  $\bigwedge x. x \in ?L \implies x \in ?R$  **by** *force*

**moreover**

```
{ fix r s r' s'
  { fix w1 w2 assume list-all2 R w1 w2 w1  $\in$  lists A w2  $\in$  lists B
    then have ((r, s), fold f w1 r, fold g w2 s)  $\in$  ?L
    proof(induction w1 w2 arbitrary: r s)
      case Nil show ?case by simp
    next
      case Cons thus ?case by (force elim!: converse-rtrancl-into-rtrancl[rotated])
    qed
  }
  hence ((r, s), (r', s'))  $\in$  ?R  $\implies$  ((r, s), (r', s'))  $\in$  ?L by auto
}
```

**hence**  $\bigwedge x. x \in ?R \implies x \in ?L$  **by** *blast*

**ultimately show** *?thesis* **by** *blast*

**qed**

**lemma** *rtrancl-fold-product1*:

**shows**  $\{(r, s). \exists a \in A. s = f a r\}^{\widehat{*}} = \{(r, s). \exists a \in lists A. s = fold f a r\}$  **(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
}
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 R
  assumes
     $\bigwedge w. \text{in-language } K \ w \implies w \in \text{lists } A$ 
     $\bigwedge w. \text{in-language } L \ w \implies w \in \text{lists } B$ 
     $\bigwedge a \ b. R \ a \ b \implies a \in A \longleftrightarrow b \in B$ 
  defines  $\mathfrak{B} \equiv \{(\partial s \ w1 \ K, \partial s \ w2 \ L) \mid w1 \ w2. w1 \in \text{lists } A \wedge w2 \in \text{lists } B \wedge \text{list-all2 } R \ w1 \ w2\}$ 
  shows rel-language R K L  $\longleftrightarrow (\forall (K, L) \in \mathfrak{B}. \circ K \longleftrightarrow \circ L)$ 
proof
  assume  $\forall (K, L) \in \mathfrak{B}. \circ K = \circ L$ 
  then show rel-language R K L
  unfolding  $\mathfrak{B}$ -def using assms(1,2)
  proof (coinduction arbitrary: K L)
    case (Lang K L)
    then have CIH:  $\bigwedge K' \ L'. \exists w1 \ w2. K' = \partial s \ w1 \ K \wedge L' = \partial s \ w2 \ L \wedge w1 \in \text{lists } A \wedge w2 \in \text{lists } B \wedge \text{list-all2 } R \ w1 \ w2 \implies \circ K' = \circ L'$  and
    [dest]:  $\bigwedge w. \text{in-language } K \ w \implies w \in \text{lists } A \wedge \text{in-language } L \ w \implies w \in \text{lists } B$ 
    by blast+
    show ?case unfolding ex-simps simp-thms
    proof (safe del: iffI)
      show  $\circ K = \circ L$  by (intro CIH[OF exI[where x = []]]) simp
    next
      fix x y w1 w2 assume  $\forall x \in \text{set } w1. x \in A \wedge \forall x \in \text{set } w2. x \in B \wedge \text{list-all2 } R \ w1 \ w2 \wedge R \ x \ y$ 
      then show  $\circ (\partial s \ w1 \ (\partial K \ x)) = \circ (\partial s \ w2 \ (\partial L \ y))$ 
      proof (cases  $x \in A \wedge y \in B$ )
        assume  $\neg (x \in A \wedge y \in B)$ 
        with assms(3)[OF  $\langle R \ x \ y \rangle$ ] show ?thesis
        by (auto simp: in-language- $\partial s$  in-language.simps[symmetric] simp del: in-language.simps)
      qed (intro CIH exI[where x = x # w1] exI[where x = y # w2], auto)
      qed (auto simp add: in-language.simps[symmetric] simp del: in-language.simps)
    qed
  qed (auto simp:  $\mathfrak{B}$ -def rel-language-alt rel-fun-def  $\circ$ - $\partial s$ )

```

## 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 Language :: 's  $\Rightarrow$  'a language
fixes wf :: 't  $\Rightarrow$  bool
fixes Lang :: 't  $\Rightarrow$  'a language
assumes distinct-alphabet: distinct alphabet
assumes Language-init: wf t  $\Longrightarrow$  Language (init t) = Lang t
assumes wellformed-init: wf t  $\Longrightarrow$  wellformed (init t)
assumes Language-alphabet:  $\llbracket$ wellformed s; in-language (Language s) w $\rrbracket \Longrightarrow w \in$ 
lists (set alphabet)
assumes wellformed-delta:  $\llbracket$ wellformed s; a  $\in$  set alphabet $\rrbracket \Longrightarrow$  wellformed (delta
a s)
assumes Language-delta:  $\llbracket$ wellformed s; a  $\in$  set alphabet $\rrbracket \Longrightarrow$  Language (delta a
s) =  $\mathfrak{d}$  (Language s) a
assumes accept-Language: wellformed s  $\Longrightarrow$  accept s  $\longleftrightarrow$   $\mathfrak{o}$  (Language s)
begin

```

**lemma** this: DA alphabet init delta accept wellformed Language wf Lang **by** unfold-locales

**lemma** wellformed-deltas:

```

 $\llbracket$ wellformed s; w  $\in$  lists (set alphabet) $\rrbracket \Longrightarrow$  wellformed (fold delta w s)
by (induction w arbitrary: s) (auto simp add: Language-delta wellformed-delta)

```

**lemma** Language-deltas:

```

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

```

Auxiliary functions:

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

```

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

```

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

```

automaton as s =
  snd (the
    (let start = (([s], {s}), {});
        test =  $\lambda$ ((ws, Z), A). ws  $\neq$  [];
        step =  $\lambda$ ((ws, Z), A).
          (let s = hd ws;
              new-edges = map ( $\lambda$ a. ((s, a), delta a s)) as;
              new = remdups (filter ( $\lambda$ ss. ss  $\notin$  Z) (map snd new-edges))
          in ((new @ tl ws, set new  $\cup$  Z), set new-edges  $\cup$  A))
    )

```

*in while-option test step start*))

**definition** *match* :: 's ⇒ 'a list ⇒ bool **where**  
*match* s w = accept (fold delta w s)

**lemma** *match-correct*:

**assumes** *wellformed* s w ∈ lists (set alphabet)

**shows** *match* s w ⇔ in-language (Language s) w

**unfolding** *match-def* accept-Language[OF *wellformed-deltas*[OF *assms*]] Language-deltas[OF *assms*] o-ds ..

**end**

**locale** *DAs* =

*M*: DA alphabet1 init1 delta1 accept1 wellformed1 Language1 wf1 Lang1 +

*N*: DA alphabet2 init2 delta2 accept2 wellformed2 Language2 wf2 Lang2

**for** *alphabet1* :: 'a1 list **and** *init1* :: 't1 ⇒ 's1 **and** *delta1* *accept1* *wellformed1* Language1 wf1 Lang1

**and** *alphabet2* :: 'a2 list **and** *init2* :: 't2 ⇒ 's2 **and** *delta2* *accept2* *wellformed2* Language2 wf2 Lang2 +

**fixes** *letter-eq* :: 'a1 ⇒ 'a2 ⇒ bool

**assumes** *letter-eq*: ∧ a b. letter-eq a b ⇒ a ∈ set alphabet1 ⇔ b ∈ set alphabet2

**begin**

**abbreviation** *step* **where**

*step* ≡ (λ(p, q). map (λ(a, b). (delta1 a p, delta2 b q))

(filter (case-prod letter-eq) (List.product alphabet1 alphabet2)))

**abbreviation** *closure* :: 's1 \* 's2 ⇒ (('s1 \* 's2) list \* ('s1 \* 's2) set) option **where**

*closure* ≡ rtrancl-while (λ(p, q). accept1 p = accept2 q) step

**theorem** *closure-sound-complete*:

**assumes** *wf*: wf1 r wf2 s

**and** *result*: closure (init1 r, init2 s) = Some (ws, R) (**is** closure (?r, ?s) = -)

**shows** ws = [] ⇔ rel-language letter-eq (Lang1 r) (Lang2 s)

**proof** –

**from** *wf* **have** *wellformed*: wellformed1 ?r wellformed2 ?s

**using** *M.wellformed-init* *N.wellformed-init* **by** blast+

**note** *Language-alphabets[simp]* =

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

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

**have** *bisim*: rel-language letter-eq (Language1 ?r) (Language2 ?s) =

(∀ a b. (∃ w1 w2. a = ds w1 (Language1 ?r) ∧ b = ds w2 (Language2 ?s) ∧

w1 ∈ lists (set alphabet1) ∧ w2 ∈ lists (set alphabet2) ∧ list-all2 letter-eq w1

w2) →

o a = o b)

by (subst lang-eq-ext-Nil-fold-Deriv) (auto dest: letter-eq)

**have** *leq*: rel-language letter-eq (Language1 ?r) (Language2 ?s) =  
 $(\forall (r', s') \in \{((r, s), (\text{delta1 } a \ r, \text{delta2 } b \ s)) \mid a \ b \ r \ s.$   
 $a \in \text{set alphabet1} \wedge b \in \text{set alphabet2} \wedge \text{letter-eq } a \ b\}^{\wedge*} \text{ “ } \{( ?r, ?s)\}.$   
*accept1* *r'* = *accept2* *s'*) **using** Language-deltass  
*M.accept-Language*[OF *M.wellformed-deltas*[OF *wellformed*(1)]]  
*N.accept-Language*[OF *N.wellformed-deltas*[OF *wellformed*(2)]]  
**unfolding** rtrancl-fold-product in-lists-conv-set bisim  
**by** (auto 10 0)

**have**  $\{(x,y). y \in \text{set } (\text{step } x)\} =$   
 $\{((r, s), (\text{delta1 } a \ r, \text{delta2 } b \ s)) \mid a \ b \ r \ s. a \in \text{set alphabet1} \wedge b \in \text{set alphabet2}$   
 $\wedge \text{letter-eq } a \ b\}$

**by** auto

**with** rtrancl-while-Some[OF result]

**have** (*ws* = []) = rel-language letter-eq (Language1 ?r) (Language2 ?s)  
**by** (auto simp add: leq Ball-def split: if-splits)

**then show** ?thesis **unfolding** *M.Language-init*[OF *wf*(1)] *N.Language-init*[OF  
*wf*(2)].

qed

## 1.2 The overall procedure

**definition** *check-eqv* :: 't1  $\Rightarrow$  't2  $\Rightarrow$  bool **where**  
*check-eqv* *r* *s* = (*wf1* *r*  $\wedge$  *wf2* *s*  $\wedge$  (case closure (*init1* *r*, *init2* *s*) of Some([], -)  $\Rightarrow$   
True | -  $\Rightarrow$  False))

**lemma** *soundness*:

**assumes** *check-eqv* *r* *s* **shows** rel-language letter-eq (Lang1 *r*) (Lang2 *s*)

**proof** –

**obtain** *R* **where** *wf1* *r* *wf2* *s* closure (*init1* *r*, *init2* *s*) = Some([], *R*)

**using** *assms* **by** (auto simp: *check-eqv-def* *Let-def* *split*: option.splits list.splits)

**from** closure-sound-complete[OF *this*] **show** rel-language letter-eq (Lang1 *r*)  
(Lang2 *s*) **by** *simp*

qed

end

## 1.3 Abstract Deterministic Finite Automaton

**locale** *DFA* = *DA* +

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

**begin**

**lemma** *finite-rtrancl-delta-Image1*:

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

**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* =  $\{\text{fold delta } w \ r \mid w. w \in \text{lists } (\text{set } as)\}$   
**and** *finite-reachable*: *finite (reachable as r)*  
**proof** –  
**obtain** *wsZ* **where** \*: *rtrancl-while*  $(\lambda-. \text{True}) (\lambda s. \text{map } (\lambda a. \text{delta } a \ s) \ as) \ r =$   
*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* =  $\{\text{fold delta } w \ r \mid w. w \in \text{lists } (\text{set } as)\}$   
**unfolding** *reachable-def* **by** (*cases wsZ*)  
*(auto dest!: rtrancl-while-Some split: if-splits simp: rtrancl-fold-product1 im-*  
*age-iff)*  
**then show** *finite (reachable as r)* **using** *assms* **by** (*force intro: finite-subset[OF*  
*- fin]*)  
**qed**  
**end**

**locale** *DFA*s =  
*M: DFA alphabet1 init1 delta1 accept1 wellformed1 Language1 wf1 Lang1 +*  
*N: DFA alphabet2 init2 delta2 accept2 wellformed2 Language2 wf2 Lang2*  
**for** *alphabet1* :: '*a1 list* **and** *init1* :: '*t1*  $\Rightarrow$  '*s1* **and** *delta1 accept1 wellformed1*  
*Language1 wf1 Lang1*  
**and** *alphabet2* :: '*a2 list* **and** *init2* :: '*t2*  $\Rightarrow$  '*s2* **and** *delta2 accept2 wellformed2*  
*Language2 wf2 Lang2 +*  
**fixes** *letter-eq* :: '*a1*  $\Rightarrow$  '*a2*  $\Rightarrow$  *bool*  
**assumes** *letter-eq*:  $\bigwedge a \ b. \text{letter-eq } a \ b \implies a \in \text{set } \text{alphabet1} \longleftrightarrow b \in \text{set } \text{alphabet2}$   
**begin**

**interpretation** *DAs* **by** *unfold-locales* (*auto dest: letter-eq*)

**lemma** *finite-rtrancl-delta-Image*:  
 $\llbracket \text{wellformed1 } r; \text{wellformed2 } s \rrbracket \implies$   
*finite*  $(\{(r, s), (\text{delta1 } a \ r, \text{delta2 } b \ s)\} \mid a \ b \ r \ s.$   
 $a \in \text{set } \text{alphabet1} \wedge b \in \text{set } \text{alphabet2} \wedge R \ a \ b\} \widehat{*} \text{ “ } \{(r, s)\}$ )  
**unfolding** *rtrancl-fold-product Image-singleton*  
**by** (*auto intro: finite-subset[OF - finite-cartesian-product[OF M.fin N.fin]]*)

**lemma** *termination*:  
**assumes** *wellformed1 r wellformed2 s*  
**shows**  $\exists st. \text{closure } (r, s) = \text{Some } st \ (\text{is } \exists -. \text{closure } ?i = -)$   
**proof** (*rule rtrancl-while-finite-Some*)  
**show** *finite*  $(\{(x, st). st \in \text{set } (\text{step } x)\} \widehat{*} \text{ “ } \{?i\})$   
**by** (*rule finite-subset[OF Image-mono[OF rtrancl-mono]*  
*finite-rtrancl-delta-Image[OF assms, of letter-eq]]*) *auto*  
**qed**



```

lemma completeness:
assumes wf1 r wf2 s rel-language letter-eq (Lang1 r) (Lang2 s) shows check-eqv
r s
proof –
  obtain ws R where 1: closure (init1 r, init2 s) = Some (ws, R) using termi-
nation
  M.wellformed-init N.wellformed-init assms by fastforce
  with closure-sound-complete[OF - - this] assms
  show check-eqv r s by (simp add: check-eqv-def)
qed

end

sublocale DA < DAs
  alphabet init delta accept wellformed Language wf Lang
  alphabet init delta accept wellformed Language wf Lang (=)
  by unfold-locales auto

sublocale DFA < DFAs
  alphabet init delta accept wellformed Language wf Lang
  alphabet init delta accept wellformed Language wf Lang (=)
  by unfold-locales auto

lemma (in DA) step-alt: step = (λ(p, q). map (λa. (delta a p, delta a q)) alphabet)
using distinct-alphabet
proof (induct alphabet)
  case (Cons x xs)
  moreover
  { fix x :: 'a and xs ys :: 'a list
    assume x ∉ set xs
    then have [(x, y) ← List.product xs (x # ys) . x = y] = [(x, y) ← List.product
xs ys . x = y]
    by (induct xs arbitrary: x) auto
  }
  moreover
  { fix x :: 'a and xs :: 'a list
    assume x ∉ set xs
    then have [(x, y) ← map (Pair x) xs . x = y] = []
    by (induct xs) auto
  }
  ultimately show ?case by (auto simp: fun-eq-iff)
qed simp

```

## 2 Derivatives of Abstract Formulas

### 2.1 Preliminaries

```

lemma pred-Diff-0[simp]: 0 ∉ A ⇒ i ∈ (λx. x - Suc 0) ‘ A ↔ Suc i ∈ 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 div-mult-mod-eq[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**

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

**definition** *dec k m* = (if  $m > k$  then  $m - \text{Suc } 0$  else  $m :: \text{nat}$ )

## 2.2 Abstract formulas

**datatype** (*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*  
**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* ( $FOr \varphi \psi$ ) = *disjuncts*  $\varphi \cup$  *disjuncts*  $\psi$   
| *disjuncts*  $\varphi = \{\varphi\}$

**fun** *conjuncts* **where**  
*conjuncts* ( $FAnd \varphi \psi$ ) = *conjuncts*  $\varphi \cup$  *conjuncts*  $\psi$   
| *conjuncts*  $\varphi = \{\varphi\}$

**fun** *disjuncts-list* **where**  
*disjuncts-list* ( $FOr \varphi \psi$ ) = *disjuncts-list*  $\varphi @$  *disjuncts-list*  $\psi$   
| *disjuncts-list*  $\varphi = [\varphi]$

**fun** *conjuncts-list* **where**  
*conjuncts-list* ( $FAnd \varphi \psi$ ) = *conjuncts-list*  $\varphi @$  *conjuncts-list*  $\psi$   
| *conjuncts-list*  $\varphi = [\varphi]$

**lemma** *finite-juncts[simp]*: *finite* (*disjuncts*  $\varphi$ ) *finite* (*conjuncts*  $\varphi$ )  
**and** *nonempty-juncts[simp]*: *disjuncts*  $\varphi \neq \{\}$  *conjuncts*  $\varphi \neq \{\}$   
**by** (*induct*  $\varphi$ ) *auto*

**lemma** *juncts-eq-set-juncts-list*:  
*disjuncts*  $\varphi = set$  (*disjuncts-list*  $\varphi$ )  
*conjuncts*  $\varphi = set$  (*conjuncts-list*  $\varphi$ )  
**by** (*induct*  $\varphi$ ) *auto*

**lemma** *notin-juncts*:  
 $[\psi \in \text{disjuncts } \varphi; \text{is-FOr } \psi] \implies \text{False}$   
 $[\psi \in \text{conjuncts } \varphi; \text{is-FAnd } \psi] \implies \text{False}$   
**by** (*induct*  $\varphi$ ) *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*  $\varphi$ ) *auto*

**lemma** *juncts-singleton*:  
 $\neg \text{is-FOr } \varphi \implies \text{disjuncts } \varphi = \{\varphi\}$   
 $\neg \text{is-FAnd } \varphi \implies \text{conjuncts } \varphi = \{\varphi\}$   
**by** (*induct*  $\varphi$ ) *auto*

**lemma** *nonempty-juncts-list*: *conjuncts-list*  $\varphi \neq []$  *disjuncts-list*  $\varphi \neq []$   
**using** *nonempty-juncts[of  $\varphi$ ]* **by** (*auto simp: Suc-le-eq juncts-eq-set-juncts-list*)

**primrec** *norm-ACI* ( $\langle\langle-\rangle\rangle$ ) **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  $\varphi s \wedge distinct \ \varphi s \wedge nf-ACI \ \psi1 \wedge nf-ACI \ \psi2$ ))  
 | *nf-ACI* (FAnd  $\psi1 \ \psi2$ ) = ( $\neg$  is-FAnd  $\psi1 \wedge$  (let  $\varphi s = \psi1 \ \# \ conjuncts-list \ \psi2$  in  
     sorted  $\varphi s \wedge distinct \ \varphi s \wedge nf-ACI \ \psi1 \wedge nf-ACI \ \psi2$ ))  
 | *nf-ACI* (FNot  $\varphi$ ) = *nf-ACI*  $\varphi$   
 | *nf-ACI* (FEx  $k \ \varphi$ ) = *nf-ACI*  $\varphi$   
 | *nf-ACI* (FAll  $k \ \varphi$ ) = *nf-ACI*  $\varphi$   
 | *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  $\varphi$ ) (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  $\varphi s$  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  $\varphi s$  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*:

$\llbracket sorted \ \varphi s; \ distinct \ \varphi s; \ list-all \ nf-ACI \ \varphi s; \ list-all \ (\lambda x. \neg is-FOr \ x) \ \varphi s \rrbracket \implies nf-ACI \ (nFOR \ \varphi s)$   
**by** (induct  $\varphi s$  rule: nFOR.induct)  
 (auto simp: juncts-list-singleton disjuncts-list-nFOR nf-ACI-D)

**lemma** *nf-ACI-nFAND*:

$\llbracket \text{sorted } \varphi s; \text{ distinct } \varphi s; \text{ list-all nf-ACI } \varphi s; \text{ list-all } (\lambda x. \neg \text{is-FAnd } x) \varphi s \rrbracket \implies$   
 $\text{nf-ACI } (\text{nFAND } \varphi s)$

**by** (*induct*  $\varphi s$  *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*  $\varphi$  *auto*)

**lemma** *nf-ACI-norm-ACI*:  $\text{nf-ACI } \langle \varphi \rangle$

**by** (*induct*  $\varphi$ )

(*force simp*: *NFOR-def NFAND-def list-all-iff*)

*intro!*: *nf-ACI-nFOR nf-ACI-nFAND elim*: *nf-ACI-juncts notin-juncts*)+

**lemma** *nFOR-Cons*:  $\text{nFOR } (x \# xs) = (\text{if } xs = [] \text{ then } x \text{ else } \text{FOR } x (\text{nFOR } xs))$

**by** (*cases*  $xs$ ) *simp-all*

**lemma** *nFAND-Cons*:  $\text{nFAND } (x \# xs) = (\text{if } xs = [] \text{ then } x \text{ else } \text{FAnd } x (\text{nFAND } xs))$

**by** (*cases*  $xs$ ) *simp-all*

**lemma** *nFOR-disjuncts*:  $\text{nf-ACI } \psi \implies \text{nFOR } (\text{disjuncts-list } \psi) = \psi$

**by** (*induct*  $\psi$ ) (*auto simp*: *juncts-list-singleton nFOR-Cons*)

**lemma** *nFAND-conjuncts*:  $\text{nf-ACI } \psi \implies \text{nFAND } (\text{conjuncts-list } \psi) = \psi$

**by** (*induct*  $\psi$ ) (*auto simp*: *juncts-list-singleton nFAND-Cons*)

**lemma** *NFOR-disjuncts*:  $\text{nf-ACI } \psi \implies \text{NFOR } (\text{disjuncts } \psi) = \psi$

**using** *nFOR-disjuncts[of  $\psi$ ]* **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*:  $\text{nf-ACI } \psi \implies \text{NFAND } (\text{conjuncts } \psi) = \psi$

**using** *nFAND-conjuncts[of  $\psi$ ]* **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*:  $\text{nf-ACI } \varphi \implies \langle \varphi \rangle = \varphi$

**by** (*induct*  $\varphi$ )

(*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-list-of-set-sort-remdups distinct-remdups-id sorted-sort-id insert-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*:

$\text{nf-ACI } \varphi \implies \text{norm-ACI } \langle \text{disjuncts } \varphi \rangle = \text{disjuncts } \varphi$

$\text{nf-ACI } \varphi \implies \text{norm-ACI } \langle \text{conjuncts } \varphi \rangle = \text{conjuncts } \varphi$

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

**locale** *Formula-Operations* =

**fixes** *TYPEVARS* :: 'a :: linorder × 'i × 'k :: {linorder, enum} × 'n × 'x × 'v

**and** *SUC* :: 'k ⇒ 'n ⇒ 'n

**and** *LESS* :: 'k ⇒ nat ⇒ 'n ⇒ bool

**and** *assigns* :: nat ⇒ 'i ⇒ 'k ⇒ 'v (⟨ $\cdot$ ⟩ [900, 999, 999] 999)

**and** *nvars* :: 'i ⇒ 'n (⟨ $\#_V$ ⟩ -> [1000] 900)

**and** *Extend* :: 'k ⇒ nat ⇒ 'i ⇒ 'v ⇒ 'i

**and** *CONS* :: 'x ⇒ 'i ⇒ 'i

**and** *SNOC* :: 'x ⇒ 'i ⇒ 'i

**and** *Length* :: 'i ⇒ nat

**and** *extend* :: 'k ⇒ bool ⇒ 'x ⇒ 'x

**and** *size* :: 'x ⇒ 'n

**and** *zero* :: 'n ⇒ 'x

**and** *alphabet* :: 'n ⇒ 'x list

**and** *eval* :: 'v ⇒ nat ⇒ bool

**and** *downshift* :: 'v ⇒ 'v

**and** *upshift* :: 'v ⇒ 'v

**and** *add* :: nat ⇒ 'v ⇒ 'v

**and** *cut* :: nat ⇒ 'v ⇒ 'v

**and** *len* :: 'v ⇒ nat

**and** *restrict* :: 'k ⇒ 'v ⇒ bool

**and** *Restrict* :: 'k ⇒ nat ⇒ ('a, 'k) aformula

**and** *lformula0* :: 'a ⇒ bool

**and** *FV0* :: 'k ⇒ 'a ⇒ nat set

**and** *find0* :: 'k ⇒ nat ⇒ 'a ⇒ bool

**and** *wf0* :: 'n ⇒ 'a ⇒ bool

**and** *decr0* :: 'k ⇒ nat ⇒ 'a ⇒ 'a

**and** *satisfies0* :: 'i ⇒ 'a ⇒ bool (**infix** ⟨ $\models_0$ ⟩ 50)

**and** *nullable0* :: 'a ⇒ bool

**and** *lderiv0* :: 'x ⇒ 'a ⇒ ('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**

$FV\ (FBool\ -)\ k = \{\}$   
 $| FV\ (FBase\ a)\ k = FV0\ k\ a$   
 $| FV\ (FNot\ \varphi)\ k = FV\ \varphi\ k$   
 $| FV\ (FOr\ \varphi\ \psi)\ k = FV\ \varphi\ k \cup FV\ \psi\ k$   
 $| FV\ (FAnd\ \varphi\ \psi)\ k = FV\ \varphi\ k \cup FV\ \psi\ k$   
 $| FV\ (FEx\ k'\ \varphi)\ k = (if\ k' = k\ then\ (\lambda x. x - 1) \text{ ' } (FV\ \varphi\ k - \{0\})\ else\ FV\ \varphi\ k)$   
 $| FV\ (FAll\ k'\ \varphi)\ k = (if\ k' = k\ then\ (\lambda x. x - 1) \text{ ' } (FV\ \varphi\ k - \{0\})\ else\ FV\ \varphi\ k)$

**primrec**  $find$  **where**

$find\ k\ l\ (FBool\ -) = False$   
 $| find\ k\ l\ (FBase\ a) = find0\ k\ l\ a$   
 $| find\ k\ l\ (FNot\ \varphi) = find\ k\ l\ \varphi$   
 $| find\ k\ l\ (FOr\ \varphi\ \psi) = (find\ k\ l\ \varphi \vee find\ k\ l\ \psi)$   
 $| find\ k\ l\ (FAnd\ \varphi\ \psi) = (find\ k\ l\ \varphi \wedge find\ k\ l\ \psi)$   
 $| find\ k\ l\ (FEx\ k'\ \varphi) = find\ k\ (if\ k = k'\ then\ Suc\ l\ else\ l)\ \varphi$   
 $| find\ k\ l\ (FAll\ k'\ \varphi) = find\ k\ (if\ k = k'\ then\ Suc\ l\ else\ l)\ \varphi$

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

$wf\ n\ (FBool\ -) = True$   
 $| wf\ n\ (FBase\ a) = wf0\ n\ a$   
 $| wf\ n\ (FNot\ \varphi) = wf\ n\ \varphi$   
 $| wf\ n\ (FOr\ \varphi\ \psi) = (wf\ n\ \varphi \wedge wf\ n\ \psi)$   
 $| wf\ n\ (FAnd\ \varphi\ \psi) = (wf\ n\ \varphi \wedge wf\ n\ \psi)$   
 $| wf\ n\ (FEx\ k\ \varphi) = wf\ (SUC\ k\ n)\ \varphi$   
 $| wf\ n\ (FAll\ k\ \varphi) = wf\ (SUC\ k\ n)\ \varphi$

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

$lformula\ (FBool\ -) = True$   
 $| lformula\ (FBase\ a) = lformula0\ a$   
 $| lformula\ (FNot\ \varphi) = lformula\ \varphi$   
 $| lformula\ (FOr\ \varphi\ \psi) = (lformula\ \varphi \wedge lformula\ \psi)$   
 $| lformula\ (FAnd\ \varphi\ \psi) = (lformula\ \varphi \wedge lformula\ \psi)$   
 $| lformula\ (FEx\ k\ \varphi) = lformula\ \varphi$   
 $| lformula\ (FAll\ k\ \varphi) = lformula\ \varphi$

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

$decr\ k\ l\ (FBool\ b) = FBool\ b$   
 $| decr\ k\ l\ (FBase\ a) = FBase\ (decr0\ k\ l\ a)$   
 $| decr\ k\ l\ (FNot\ \varphi) = FNot\ (decr\ k\ l\ \varphi)$   
 $| decr\ k\ l\ (FOr\ \varphi\ \psi) = FOr\ (decr\ k\ l\ \varphi)\ (decr\ k\ l\ \psi)$   
 $| decr\ k\ l\ (FAnd\ \varphi\ \psi) = FAnd\ (decr\ k\ l\ \varphi)\ (decr\ k\ l\ \psi)$   
 $| decr\ k\ l\ (FEx\ k'\ \varphi) = FEx\ k'\ (decr\ k\ (if\ k = k'\ then\ Suc\ l\ else\ l)\ \varphi)$   
 $| decr\ k\ l\ (FAll\ k'\ \varphi) = FAll\ k'\ (decr\ k\ (if\ k = k'\ then\ Suc\ l\ else\ l)\ \varphi)$

**primrec** *satisfies-gen* :: ('k ⇒ 'v ⇒ nat ⇒ bool) ⇒ 'i ⇒ ('a, 'k) aformula ⇒ bool  
**where**

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

**abbreviation** *satisfies* (infix <|=> 50) **where**

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

**abbreviation** *satisfies-bounded* (infix <|=b> 50) **where**

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

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

*sat-vars-gen* r K  $\mathfrak{A}$   $\varphi \equiv$   
*satisfies-gen* ( $\lambda$ k P n. restrict k P  $\wedge$  r k P n)  $\mathfrak{A}$   $\varphi \wedge$  ( $\forall k \in K. \forall x \in FV \varphi k.$   
restrict k ( $x^{\mathfrak{A}}k$ ))

**definition** *sat* **where**

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

**definition** *sat<sub>b</sub>* **where**

*sat<sub>b</sub>*  $\mathfrak{A}$   $\varphi \equiv$  *sat-vars-gen* ( $\lambda$ - P n. len P ≤ n) UNIV  $\mathfrak{A}$   $\varphi$

**fun** *RESTR* **where**

*RESTR* (FOr  $\varphi$   $\psi$ ) = FOr (*RESTR*  $\varphi$ ) (*RESTR*  $\psi$ )  
| *RESTR* (FAnd  $\varphi$   $\psi$ ) = FAnd (*RESTR*  $\varphi$ ) (*RESTR*  $\psi$ )  
| *RESTR* (FNot  $\varphi$ ) = FNot (*RESTR*  $\varphi$ )  
| *RESTR* (FEx k  $\varphi$ ) = FEx k (FAnd (Restrict k 0) (*RESTR*  $\varphi$ ))  
| *RESTR* (FAll k  $\varphi$ ) = FAll k (FOr (FNot (Restrict k 0)) (*RESTR*  $\varphi$ ))  
| *RESTR*  $\varphi$  =  $\varphi$

**abbreviation** *RESTRICT-VARS* **where**

*RESTRICT-VARS* ks V  $\varphi \equiv$   
foldr (%k  $\varphi$ . foldr ( $\lambda$ x  $\varphi$ . FAnd (Restrict k x)  $\varphi$ ) (V k)  $\varphi$ ) ks (*RESTR*  $\varphi$ )

**definition** *RESTRICT* **where**

*RESTRICT*  $\varphi \equiv$  *RESTRICT-VARS* Enum.enum (sorted-list-of-set o FV  $\varphi$ )  $\varphi$

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

*nullable* (FBool b) = b  
| *nullable* (FBase a) = *nullable0* a  
| *nullable* (FNot  $\varphi$ ) = ( $\neg$  *nullable*  $\varphi$ )



|  $\text{nullable } (FOr \varphi \psi) = (\text{nullable } \varphi \vee \text{nullable } \psi)$   
|  $\text{nullable } (FAnd \varphi \psi) = (\text{nullable } \varphi \wedge \text{nullable } \psi)$   
|  $\text{nullable } (FEx k \varphi) = \text{nullable } \varphi$   
|  $\text{nullable } (FAll k \varphi) = \text{nullable } \varphi$

**fun**  $nFOr :: ('a, 'k) aformula \Rightarrow ('a, 'k) aformula \Rightarrow ('a, 'k) aformula$  **where**  
 $nFOr (FBool b1) (FBool b2) = FBool (b1 \vee b2)$   
|  $nFOr (FBool b) \psi = (\text{if } b \text{ then } FBool \text{ True } \text{ else } \psi)$   
|  $nFOr \varphi (FBool b) = (\text{if } b \text{ then } FBool \text{ True } \text{ else } \varphi)$   
|  $nFOr (FOr \varphi1 \varphi2) \psi = nFOr \varphi1 (nFOr \varphi2 \psi)$   
|  $nFOr \varphi (FOr \psi1 \psi2) =$   
 $(\text{if } \varphi = \psi1 \text{ then } FOr \psi1 \psi2$   
 $\text{ else if } \varphi < \psi1 \text{ then } FOr \varphi (FOr \psi1 \psi2)$   
 $\text{ else } FOr \psi1 (nFOr \varphi \psi2))$   
|  $nFOr \varphi \psi =$   
 $(\text{if } \varphi = \psi \text{ then } \varphi$   
 $\text{ else if } \varphi < \psi \text{ then } FOr \varphi \psi$   
 $\text{ else } FOr \psi \varphi)$

**fun**  $nFAnd :: ('a, 'k) aformula \Rightarrow ('a, 'k) aformula \Rightarrow ('a, 'k) aformula$  **where**  
 $nFAnd (FBool b1) (FBool b2) = FBool (b1 \wedge b2)$   
|  $nFAnd (FBool b) \psi = (\text{if } b \text{ then } \psi \text{ else } FBool \text{ False})$   
|  $nFAnd \varphi (FBool b) = (\text{if } b \text{ then } \varphi \text{ else } FBool \text{ False})$   
|  $nFAnd (FAnd \varphi1 \varphi2) \psi = nFAnd \varphi1 (nFAnd \varphi2 \psi)$   
|  $nFAnd \varphi (FAnd \psi1 \psi2) =$   
 $(\text{if } \varphi = \psi1 \text{ then } FAnd \psi1 \psi2$   
 $\text{ else if } \varphi < \psi1 \text{ then } FAnd \varphi (FAnd \psi1 \psi2)$   
 $\text{ else } FAnd \psi1 (nFAnd \varphi \psi2))$   
|  $nFAnd \varphi \psi =$   
 $(\text{if } \varphi = \psi \text{ then } \varphi$   
 $\text{ else if } \varphi < \psi \text{ then } FAnd \varphi \psi$   
 $\text{ else } FAnd \psi \varphi)$

**fun**  $nFEx :: 'k \Rightarrow ('a, 'k) aformula \Rightarrow ('a, 'k) aformula$  **where**  
 $nFEx k (FOr \varphi \psi) = nFOr (nFEx k \varphi) (nFEx k \psi)$   
|  $nFEx k \varphi = (\text{if find } k \ 0 \ \varphi \text{ then } FEx k \ \varphi \text{ else } \text{decr } k \ 0 \ \varphi)$

**fun**  $nFAll$  **where**  
 $nFAll k (FAnd \varphi \psi) = nFAnd (nFAll k \varphi) (nFAll k \psi)$   
|  $nFAll k \varphi = (\text{if find } k \ 0 \ \varphi \text{ then } FAll k \ \varphi \text{ else } \text{decr } k \ 0 \ \varphi)$

**fun**  $nFNot :: ('a, 'k) aformula \Rightarrow ('a, 'k) aformula$  **where**  
 $nFNot (FNot \varphi) = \varphi$   
|  $nFNot (FOr \varphi \psi) = nFAnd (nFNot \varphi) (nFNot \psi)$   
|  $nFNot (FAnd \varphi \psi) = nFOr (nFNot \varphi) (nFNot \psi)$   
|  $nFNot (FEx b \varphi) = nFAll b (nFNot \varphi)$   
|  $nFNot (FAll b \varphi) = nFEx b (nFNot \varphi)$   
|  $nFNot (FBool b) = FBool (\neg b)$   
|  $nFNot \varphi = FNot \varphi$

**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* *lderv0*

**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*  $\varphi$   $\psi$ ) = *FAnd* (*fold* (*deriv* *d0*) *xs*  $\varphi$ ) (*fold* (*deriv* *d0*) *xs*  $\psi$ )

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

**lemma** *fold-deriv-FEx*:

$\{\text{fold } (\text{deriv } d0) \text{ } xs \text{ } (FEx \ k \ \varphi) \mid xs. \ \text{True}\} \subseteq$

$\{FEx \ k \ \psi \mid \psi. \ \text{nf-ACI } \psi \wedge \text{disjuncts } \psi \subseteq (\bigcup xs. \text{disjuncts } \langle \text{fold } (\text{deriv } d0) \ \rangle xs)$

```

φ)))}
proof –
  { fix xs
    have  $\exists \psi. \langle \text{fold } (deriv\ d0) \ xs \ (FEx\ k\ \varphi) \rangle = FEx\ k\ \psi \wedge$ 
       $nf\text{-}ACI\ \psi \wedge \text{disjuncts } \psi \subseteq (\bigcup xs. \text{disjuncts } \langle \text{fold } (deriv\ d0) \ xs\ \varphi \rangle)$ 
    proof (induct xs arbitrary: φ)
      case (Cons x xs)
      let  $?\varphi = FOr\ (deriv\ d0\ (extend\ k\ True\ x)\ \varphi)\ (deriv\ d0\ (extend\ k\ False\ x)\ \varphi)$ 
      from Cons[of ?φ] obtain  $\psi$  where  $\langle \text{fold } (deriv\ d0) \ xs \ (FEx\ k\ ?\varphi) \rangle = FEx\ k$ 
 $\psi$ 
       $nf\text{-}ACI\ \psi$  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  $(\bigcup xs. \text{disjuncts } \langle \text{fold } (deriv\ d0) \ xs\ ?\varphi \rangle) \subseteq$ 
           $(\bigcup xs. \text{disjuncts } \langle \text{fold } (Formula\text{-}Operations.deriv\ extend\ d0) \ xs\ \varphi \rangle)$ 
        by (force simp: fold-deriv-FOr nf-ACI-juncts nf-ACI-norm-ACI
          dest: notin-juncts subsetD[OF equalityD1[OF disjuncts-NFOR], rotated
 $-1]$ 
          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:*

```

{  $\langle \text{fold } (deriv\ d0) \ xs \ (FAll\ k\ \varphi) \rangle \mid xs. True \} \subseteq$ 
 $\{ FAll\ k\ \psi \mid \psi. nf\text{-}ACI\ \psi \wedge \text{conjuncts } \psi \subseteq (\bigcup xs. \text{conjuncts } \langle \text{fold } (deriv\ d0) \ xs$ 
 $\varphi \rangle) \}$ 
proof –
  { fix xs
    have  $\exists \psi. \langle \text{fold } (deriv\ d0) \ xs \ (FAll\ k\ \varphi) \rangle = FAll\ k\ \psi \wedge$ 
       $nf\text{-}ACI\ \psi \wedge \text{conjuncts } \psi \subseteq (\bigcup xs. \text{conjuncts } \langle \text{fold } (deriv\ d0) \ xs\ \varphi \rangle)$ 
    proof (induct xs arbitrary: φ)
      case (Cons x xs)
      let  $?\varphi = FAnd\ (deriv\ d0\ (extend\ k\ True\ x)\ \varphi)\ (deriv\ d0\ (extend\ k\ False\ x)\ \varphi)$ 
      from Cons[of ?φ] obtain  $\psi$  where  $\langle \text{fold } (deriv\ d0) \ xs \ (FAll\ k\ ?\varphi) \rangle = FAll\ k$ 
 $\psi$ 
       $nf\text{-}ACI\ \psi$  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  $(\bigcup xs. \text{conjuncts } \langle \text{fold } (deriv\ d0) \ xs\ ?\varphi \rangle) \subseteq$ 
           $(\bigcup xs. \text{conjuncts } \langle \text{fold } (Formula\text{-}Operations.deriv\ extend\ d0) \ xs\ \varphi \rangle)$ 
        by (force simp: fold-deriv-FAnd nf-ACI-juncts nf-ACI-norm-ACI
          dest: notin-juncts subsetD[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) aformula  $\Rightarrow$  ('a, 'k) aformula
  shows finite B  $\Rightarrow$  finite {f  $\varphi$  |  $\varphi$ . nf-ACI  $\varphi \wedge$  disjuncts  $\varphi \subseteq B$ }
    finite B  $\Rightarrow$  finite {f  $\varphi$  |  $\varphi$ . nf-ACI  $\varphi \wedge$  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 :: linorder  $\times$  'i  $\times$  'k :: {linorder, enum}  $\times$  '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  $\mathfrak{A} = 0 \wedge \#_V$   $\mathfrak{A} = \text{idx}$ 
  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 k i ( $\#_V$   $\mathfrak{A}$ )  $\Rightarrow$ 
     $m^{\text{Extend k i } \mathfrak{A} P}_{k'} = (\text{if } k = k' \text{ then } (\text{if } m = i \text{ then } P \text{ else } \text{dec } i \ m^{\mathfrak{A}} k) \text{ else } m^{\mathfrak{A}} k')$ 
  and assigns-SNOC-zero: LESS k m ( $\#_V$   $\mathfrak{A}$ )  $\Rightarrow m^{\text{SNOC (zero } (\#_V \mathfrak{A})) \mathfrak{A}}_k = m^{\mathfrak{A}}_k$ 
  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:  $\#_V$   $\mathfrak{A} = \text{size } x \Rightarrow \text{Extend k 0 (CONS x } \mathfrak{A}) P =$ 
    CONS (extend k (if eval P 0 then True else False) x) (Extend k 0  $\mathfrak{A}$  (downshift
P))
  and Extend-SNOC-cut:  $\#_V$   $\mathfrak{A} = \text{size } x \Rightarrow \text{len } P \leq \text{Length (SNOC x } \mathfrak{A}) \Rightarrow$ 

```

*Extend k 0 (SNOC x  $\mathfrak{A}$ ) P =*  
*SNOC (extend k (if eval P (Length  $\mathfrak{A}$ ) then True else False) x) (Extend k 0  $\mathfrak{A}$*   
*(cut (Length  $\mathfrak{A}$ ) P))*  
**and** *CONS-inj: size x =  $\#_V \mathfrak{A} \implies$  size y =  $\#_V \mathfrak{A} \implies \#_V \mathfrak{A} = \#_V \mathfrak{B} \implies$*   
*CONS x  $\mathfrak{A} =$  CONS y  $\mathfrak{B} \iff (x = y \wedge \mathfrak{A} = \mathfrak{B})$*   
**and** *CONS-surj: Length  $\mathfrak{A} \neq 0 \implies \#_V \mathfrak{A} = \text{idx} \implies$*   
 *$\exists x \mathfrak{B}. \mathfrak{A} = \text{CONS } x \mathfrak{B} \wedge \#_V \mathfrak{B} = \text{idx} \wedge \text{size } x = \text{idx}$*   
  
**and** *size-zero: size (zero idx) = idx*  
**and** *size-extend: size (extend k b x) = SUC k (size x)*  
**and** *distinct-alphabet: distinct (alphabet idx)*  
**and** *alphabet-size:  $x \in \text{set (alphabet idx)} \iff \text{size } x = \text{idx}$*   
  
**and** *downshift-upshift: downshift (upshift P) = P*  
**and** *downshift-add-zero: downshift (add 0 P) = downshift P*  
**and** *eval-add: eval (add n P) n*  
**and** *eval-upshift:  $\neg \text{eval (upshift P) 0}$*   
**and** *eval-ge-len:  $p \geq \text{len } P \implies \neg \text{eval } P p$*   
**and** *len-cut-le: len (cut n P)  $\leq n$*   
**and** *len-cut: len P  $\leq n \implies \text{cut } n P = P$*   
**and** *cut-add: cut n (add m P) = (if  $m \geq n$  then cut n P else add m (cut n P))*  
**and** *len-add: len (add m P) = max (Suc m) (len P)*  
**and** *len-upshift: len (upshift P) = (case len P of 0  $\Rightarrow$  0 | n  $\Rightarrow$  Suc n)*  
**and** *len-downshift: len (downshift P) = (case len P of 0  $\Rightarrow$  0 | Suc n  $\Rightarrow$  n)*  
  
**and** *wf0-decr0:  $\llbracket \text{wf0 (SUC k idx) } a; \text{LESS } k l (\text{SUC } k \text{ idx}); \neg \text{find0 } k l a \rrbracket \implies$*   
*wf0 idx (decr0 k l a)*  
**and** *lformula0-decr0: lformula0  $\varphi \implies \text{lformula0 (decr0 k l } \varphi)$*   
**and** *Extend-satisfies0:  $\llbracket \neg \text{find0 } k i a; \text{LESS } k i (\text{SUC } k (\#_V \mathfrak{A})); \text{lformula0 } a \vee$*   
*len P  $\leq$  Length  $\mathfrak{A} \rrbracket \implies$*   
*Extend k i  $\mathfrak{A}$  P  $\models_0 a \iff \mathfrak{A} \models_0 \text{decr0 } k i a$*   
**and** *nullable0-satisfies0: Length  $\mathfrak{A} = 0 \implies \text{nullable0 } a \iff \mathfrak{A} \models_0 a$*   
**and** *satisfies0-eqI: wf0 ( $\#_V \mathfrak{B}$ ) a  $\implies \#_V \mathfrak{A} = \#_V \mathfrak{B} \implies \text{lformula0 } a \implies$*   
*( $\bigwedge m k. \text{LESS } k m (\#_V \mathfrak{B}) \implies m^{\mathfrak{A}} k = m^{\mathfrak{B}} k$ )  $\implies \mathfrak{A} \models_0 a \iff \mathfrak{B} \models_0 a$*   
**and** *wf-lderiv0:  $\llbracket \text{wf0 idx } a; \text{lformula0 } a \rrbracket \implies \text{wf idx (lderiv0 } x a)$*   
**and** *lformula-lderiv0: lformula0 a  $\implies \text{lformula (lderiv0 } x a)$*   
**and** *wf-rderiv0: wf0 idx a  $\implies \text{wf idx (rderiv0 } x a)$*   
**and** *satisfies-lderiv0:*  
 *$\llbracket \text{wf0 } (\#_V \mathfrak{A}) a; \#_V \mathfrak{A} = \text{size } x; \text{lformula0 } a \rrbracket \implies \mathfrak{A} \models \text{lderiv0 } x a \iff \text{CONS}$*   
*x  $\mathfrak{A} \models_0 a$*   
**and** *satisfies-bounded-lderiv0:*  
 *$\llbracket \text{wf0 } (\#_V \mathfrak{A}) a; \#_V \mathfrak{A} = \text{size } x; \text{lformula0 } a \rrbracket \implies \mathfrak{A} \models_b \text{lderiv0 } x a \iff \text{CONS}$*   
*x  $\mathfrak{A} \models_0 a$*   
**and** *satisfies-bounded-rderiv0:*  
 *$\llbracket \text{wf0 } (\#_V \mathfrak{A}) a; \#_V \mathfrak{A} = \text{size } x \rrbracket \implies \mathfrak{A} \models_b \text{rderiv0 } x a \iff \text{SNOC } x \mathfrak{A} \models_0 a$*   
**and** *find0-FV0:  $\llbracket \text{wf0 idx } a; \text{LESS } k l \text{ idx} \rrbracket \implies \text{find0 } k l a \iff l \in \text{FV0 } k a$*

**and** *finite-FV0*: *finite (FV0 k a)*  
**and** *wf0-FV0-LESS*:  $\llbracket \text{wf0 } idx \ a; \ v \in \text{FV0 } k \ a \rrbracket \implies \text{LESS } k \ v \ idx$   
**and** *restrict-Restrict*:  $i^{\mathfrak{A}}k = P \implies \text{restrict } k \ P \longleftrightarrow \text{satisfies-gen } r \ \mathfrak{A} \ (\text{Restrict } k \ i)$   
**and** *wf-Restrict*:  $\text{LESS } k \ i \ idx \implies \text{wf } idx \ (\text{Restrict } k \ i)$   
**and** *lformula-Restrict*: *lformula (Restrict k i)*  
**and** *finite-lderiv0*: *lformula0 a  $\implies$  finite {fold lderiv xs (FBase a) | xs. True}*  
**and** *finite-rderiv0*: *finite {fold rderiv xs (FBase a) | xs. True}*

**context** *Formula*  
**begin**

**lemma** *satisfies-eqI*:

$\llbracket \text{wf } (\#_V \ \mathfrak{A}) \ \varphi; \ \#_V \ \mathfrak{A} = \#_V \ \mathfrak{B}; \ \bigwedge m \ k. \ \text{LESS } k \ m \ (\#_V \ \mathfrak{A}) \implies m^{\mathfrak{A}}k = m^{\mathfrak{B}}k; \ \text{lformula } \varphi \rrbracket \implies$

$\mathfrak{A} \models \varphi \longleftrightarrow \mathfrak{B} \models \varphi$

**proof** (*induct  $\varphi$  arbitrary:  $\mathfrak{A} \ \mathfrak{B}$* )

**case** (*FEx k  $\varphi$* )

**from** *FEx.prem*s **have**  $\bigwedge P. (\text{Extend } k \ 0 \ \mathfrak{A} \ P \models \varphi) \longleftrightarrow (\text{Extend } k \ 0 \ \mathfrak{B} \ P \models \varphi)$

**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**  $\bigwedge P. (\text{Extend } k \ 0 \ \mathfrak{A} \ P \models \varphi) \longleftrightarrow (\text{Extend } k \ 0 \ \mathfrak{B} \ P \models \varphi)$

**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  $\varphi$* )

**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*:

$\llbracket \text{wf } (\text{SUC } k \ idx) \ \varphi; \ \text{LEQ } k \ l \ idx; \ \neg \text{find } k \ l \ \varphi \rrbracket \implies \text{wf } idx \ (\text{decr } k \ l \ \varphi)$

**by** (*induct  $\varphi$  arbitrary:  $idx \ l$* ) (*auto simp: wf0-decr0 LESS-SUC SUC-SUC*)

**lemma** *lformula-decr*:

*lformula  $\varphi \implies$  lformula (decr k l  $\varphi$ )*

**by** (*induct  $\varphi$  arbitrary:  $l$* ) (*auto simp: lformula0-decr0*)

**lemma** *Extend-satisfies-decr*:

$\llbracket \neg \text{find } k \ i \ \varphi; \ \text{LEQ } k \ i \ (\#_V \ \mathfrak{A}); \ \text{lformula } \varphi \rrbracket \implies \text{Extend } k \ i \ \mathfrak{A} \ P \models \varphi \longleftrightarrow \mathfrak{A} \models \text{decr } k \ i \ \varphi$

**by** (*induct  $\varphi$  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*:  $k \neq k' \implies \text{LEQ } k \ i \ (\text{SUC } k' \ \text{idx}) = \text{LEQ } k \ i \ \text{idx}$   
**by** (*metis LESS-SUC(2) SUC-SUC*)

**lemma** *Extend-satisfies-bounded-decr*:

$\llbracket \neg \text{find } k \ i \ \varphi; \text{LEQ } k \ i \ (\#_V \ \mathfrak{A}); \text{len } P \leq \text{Length } \mathfrak{A} \rrbracket \implies$   
 $\text{Extend } k \ i \ \mathfrak{A} \ P \models_b \varphi \longleftrightarrow \mathfrak{A} \models_b \text{decr } k \ i \ \varphi$

**proof** (*induct*  $\varphi$  *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*)

## 2.3 Normalization

**lemma** *wf-nFOr*:

$wf \ \text{idx} \ (\text{FOr } \varphi \ \psi) \implies wf \ \text{idx} \ (\text{nFOr } \varphi \ \psi)$

**by** (*induct*  $\varphi \ \psi$  *rule: nFOr.induct*) (*simp-all add: Let-def*)

**lemma** *wf-nFAnd*:

$wf \ \text{idx} \ (\text{FAnd } \varphi \ \psi) \implies wf \ \text{idx} \ (\text{nFAnd } \varphi \ \psi)$

**by** (*induct*  $\varphi \ \psi$  *rule: nFAnd.induct*) (*simp-all add: Let-def*)

**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-nFNot*:

*wf idx (FNot  $\varphi$ )  $\implies$  wf idx (nFNot  $\varphi$ )*

**by** (*induct  $\varphi$  arbitrary: idx rule: nFNot.induct*) (*auto simp: wf-nFOr wf-nFAnd wf-nFEx wf-nFAll*)

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

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

**lemma** *lformula-nFOr*:

*lformula (FOr  $\varphi \psi$ )  $\implies$  lformula (nFOr  $\varphi \psi$ )*

**by** (*induct  $\varphi \psi$  rule: nFOr.induct*) (*simp-all add: Let-def*)

**lemma** *lformula-nFAnd*:

*lformula (FAnd  $\varphi \psi$ )  $\implies$  lformula (nFAnd  $\varphi \psi$ )*

**by** (*induct  $\varphi \psi$  rule: nFAnd.induct*) (*simp-all add: Let-def*)

**lemma** *lformula-nFEx*:

*lformula (FEx b  $\varphi$ )  $\implies$  lformula (nFEx b  $\varphi$ )*

**by** (*induct  $\varphi$  rule: nFEx.induct*)

(*auto simp: lformula-nFOr lformula0-decr0 lformula-decr*)

**lemma** *lformula-nFAll*:

*lformula (FAll b  $\varphi$ )  $\implies$  lformula (nFAll b  $\varphi$ )*

**by** (*induct  $\varphi$  rule: nFAll.induct*)

(*auto simp: lformula-nFAnd lformula0-decr0 lformula-decr*)

**lemma** *lformula-nFNot*:

*lformula (FNot  $\varphi$ )  $\implies$  lformula (nFNot  $\varphi$ )*

**by** (*induct  $\varphi$  rule: nFNot.induct*) (*auto simp: lformula-nFOr lformula-nFAnd lformula-nFEx lformula-nFAll*)

**lemma** *lformula-norm*: *lformula  $\varphi \implies$  lformula (norm  $\varphi$ )*

**by** (*induct  $\varphi$* ) (*simp-all add: lformula-nFOr lformula-nFAnd lformula-nFNot lformula-nFEx lformula-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-nFEx*: *lformula  $\varphi \implies \mathfrak{A} \models nFEx b \varphi \longleftrightarrow \mathfrak{A} \models FEx b \varphi$*

**by** (*induct  $\varphi$  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*: *lformula  $\varphi \implies \mathfrak{A} \models nFAll b \varphi \longleftrightarrow \mathfrak{A} \models FAll b \varphi$*

**by** (*induct  $\varphi$  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-nFNot*:

*lformula  $\varphi \implies \mathfrak{A} \models nFNot \varphi \longleftrightarrow \mathfrak{A} \models FNot \varphi$*   
**by** (*induct  $\varphi$  arbitrary:  $\mathfrak{A}$* )  
*(simp-all add: satisfies-nFOR satisfies-nFAnd satisfies-nFEx satisfies-nFAll*  
*lformula-nFNot)*

**lemma** *satisfies-norm*: *lformula  $\varphi \implies \mathfrak{A} \models norm \varphi \longleftrightarrow \mathfrak{A} \models \varphi$*

**using** *satisfies-nFOR satisfies-nFAnd satisfies-nFNot satisfies-nFEx satisfies-nFAll*  
**by** (*induct  $\varphi$  arbitrary:  $\mathfrak{A}$* ) (*simp-all add: lformula-norm*)

**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  $\varphi \psi$  arbitrary:  $\mathfrak{A}$  rule:  $nFAnd.induct$* ) *auto*

**lemma** *len-cut-0*: *len (cut 0 P) = 0*

**by** (*metis le-0-eq len-cut-le*)

**lemma** *satisfies-bounded-nFEx*:  *$\mathfrak{A} \models_b nFEx b \varphi \longleftrightarrow \mathfrak{A} \models_b FEx b \varphi$*

**by** (*induct  $\varphi$  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*:  $\mathfrak{A} \models_b nFAll\ b\ \varphi \longleftrightarrow \mathfrak{A} \models_b FAll\ b\ \varphi$   
**by** (*induct*  $\varphi$  *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-nFNot*:  
 $\mathfrak{A} \models_b nFNot\ \varphi \longleftrightarrow \mathfrak{A} \models_b FNot\ \varphi$   
**by** (*induct*  $\varphi$  *arbitrary*:  $\mathfrak{A}$ )  
*(auto simp: satisfies-bounded-nFOr satisfies-bounded-nFAnd satisfies-bounded-nFEx*  
*satisfies-bounded-nFAll)*

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

## 2.4 Derivatives of Formulas

**lemma** *wf-ldderiv*:  
 $\llbracket wf\ idx\ \varphi; lformula\ \varphi \rrbracket \Longrightarrow wf\ idx\ (lderiv\ x\ \varphi)$   
**by** (*induct*  $\varphi$  *arbitrary*:  $x\ idx$ ) (*auto simp: wf-ldderiv0*)

**lemma** *lformula-ldderiv*:  
 $lformula\ \varphi \Longrightarrow lformula\ (lderiv\ x\ \varphi)$   
**by** (*induct*  $\varphi$  *arbitrary*:  $x$ ) (*auto simp: lformula-ldderiv0*)

**lemma** *wf-rderiv*:  
 $wf\ idx\ \varphi \Longrightarrow wf\ idx\ (rderiv\ x\ \varphi)$   
**by** (*induct*  $\varphi$  *arbitrary*:  $x\ idx$ ) (*auto simp: wf-rderiv0*)

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

**proof** (*induct*  $\varphi$  *arbitrary*:  $x\ \mathfrak{A}$ )

**case** (*FEx*  $k\ \varphi$ )

**from** *FEx.premis 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.premis 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*:

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

**proof** (*induct*  $\varphi$  *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*s *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*s *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*  $\varphi$  *arbitrary: x*  $\mathfrak{A}$ )

**case** (*FEx*  $k\ \varphi$ )

**from** *FEx.prem*s *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*s *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 for P*] *spec*[*of - add (Length*  $\mathfrak{A}$ ) *P for P*] *split: if-splits*)

**qed** (*simp-all add: satisfies-bounded-rderiv0 split: bool.splits*)

**lemma** *wf-norm-rderivs*:  $wf\ idx\ \varphi \implies wf\ idx\ (((norm\ \circ\ rderiv\ (zero\ idx))\ \sim\sim\ k)\ \varphi)$

**by** (*induct k*) (*auto simp: wf-norm wf-rderiv*)

## 2.5 Finiteness of Derivatives Modulo ACI

**lemma** *finite-fold-deriv*:

**assumes**  $(d0 = lderiv0 \wedge lformula\ \varphi) \vee d0 = rderiv0$

**shows**  $finite\ \{\langle fold\ (deriv\ d0)\ xs\ \varphi \rangle \mid xs.\ True\}$

**using** *assms* **proof** (*induct*  $\varphi$ )

**case** (*FBase*  $a$ ) **then show** *?case*

**by** (*auto intro*:

*finite-subset*[*OF* - *finite-imageI*[*OF* *finite-ldderiv0*]]

*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(1)*])

**next**

**case** (*FOr*  $\varphi\ \psi$ )

**then show** *?case*

**by** (*auto simp: fold-deriv-FOr intro!*: *finite-surj*[*OF* *finite-cartesian-product*[*OF* *FOr(1,2)*]])

**next**

**case** (*FAnd*  $\varphi\ \psi$ )

**then show** *?case*

**by** (*auto simp: fold-deriv-FAnd intro!*: *finite-surj*[*OF* *finite-cartesian-product*[*OF* *FAnd(1,2)*]])

**next**

**case** (*FEx*  $k\ \varphi$ )

**then have**  $finite\ (\bigcup\ (disjuncts\ \langle \{ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle \mid xs.\ True \} \rangle))$  **by** *auto*

**then have**  $finite\ (\bigcup\ xs.\ disjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle)$  **by** (*rule* *finite-subset*[*rotated*])

*auto*

**then have**  $finite\ \{ FEx\ k\ \psi \mid \psi.\ nf\text{-}ACI\ \psi \wedge disjuncts\ \psi \subseteq (\bigcup\ xs.\ disjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle) \}$

**by** (*rule* *finite-norm-ACI-juncts*)

**then show** *?case* **by** (*rule* *finite-subset*[*OF* *fold-deriv-FEx*])

**next**

**case** (*FAll*  $k\ \varphi$ )

**then have**  $finite\ (\bigcup\ (conjuncts\ \langle \{ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle \mid xs.\ True \} \rangle))$  **by** *auto*

**then have**  $finite\ (\bigcup\ xs.\ conjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle)$  **by** (*rule* *finite-subset*[*rotated*])

*auto*

**then have**  $finite\ \{ FAll\ k\ \psi \mid \psi.\ nf\text{-}ACI\ \psi \wedge conjuncts\ \psi \subseteq (\bigcup\ xs.\ conjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi \rangle) \}$

**by** (*rule* *finite-norm-ACI-juncts*)

**then show** *?case* **by** (*rule* *finite-subset*[*OF* *fold-deriv-FAll*])

**qed** (*simp add: fold-deriv-FBool*)

**lemma** *lformula-nFOR*:  $lformula\ (nFOR\ \varphi s) = (\forall\ \varphi \in set\ \varphi s.\ lformula\ \varphi)$

**by** (*induct*  $\varphi s$  *rule: nFOR.induct*) *auto*

**lemma** *lformula-nFAND*:  $lformula\ (nFAND\ \varphi s) = (\forall\ \varphi \in set\ \varphi s.\ lformula\ \varphi)$

**by** (*induct*  $\varphi s$  *rule: nFAND.induct*) *auto*

**lemma** *lformula-NFOR*:  $\text{finite } \Phi \implies \text{lformula } (\text{NFOR } \Phi) = (\forall \varphi \in \Phi. \text{lformula } \varphi)$

**unfolding** *NFOR-def o-apply lformula-nFOR by simp*

**lemma** *lformula-NFAND*:  $\text{finite } \Phi \implies \text{lformula } (\text{NFAND } \Phi) = (\forall \varphi \in \Phi. \text{lformula } \varphi)$

**unfolding** *NFAND-def o-apply lformula-nFAND by simp*

**lemma** *lformula-disjuncts*:  $(\forall \psi \in \text{disjuncts } \varphi. \text{lformula } \psi) = \text{lformula } \varphi$   
**by** (*induct*  $\varphi$  rule: *disjuncts.induct*) *fastforce+*

**lemma** *lformula-conjuncts*:  $(\forall \psi \in \text{conjuncts } \varphi. \text{lformula } \psi) = \text{lformula } \varphi$   
**by** (*induct*  $\varphi$  rule: *conjuncts.induct*) *fastforce+*

**lemma** *lformula-norm-ACI*:  $\text{lformula } \langle \varphi \rangle = \text{lformula } \varphi$

**by** (*induct*  $\varphi$ ) (*simp-all add: ball-Un*)

*lformula-NFOR lformula-disjuncts lformula-NFAND lformula-conjuncts*)

**theorem**

*finite-fold-lderiv*:  $\text{lformula } \varphi \implies \text{finite } \{\langle \text{fold lderiv } xs \ \langle \varphi \rangle \rangle \mid xs. \text{True}\}$  **and**

*finite-fold-rderiv*:  $\text{finite } \{\langle \text{fold rderiv } xs \ \langle \varphi \rangle \rangle \mid xs. \text{True}\}$

**by** (*subst (asm) lformula-norm-ACI[symmetric]*) (*blast intro: nf-ACI-norm-ACI finite-fold-deriv*)**+**

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

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

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

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

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

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

**lemma** *satisfies-bounded-NFOR*:  $\text{finite } \Phi \implies \mathfrak{A} \models_b \text{NFOR } \Phi \longleftrightarrow (\exists \varphi \in \Phi. \mathfrak{A} \models_b \varphi)$

**unfolding** *NFOR-def o-apply by (auto simp: satisfies-bounded-nFOR)*

**lemma** *satisfies-bounded-NFAND*:  $\text{finite } \Phi \implies \mathfrak{A} \models_b \text{NFAND } \Phi \longleftrightarrow (\forall \varphi \in \Phi. \mathfrak{A} \models_b \varphi)$

**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*  $\varphi$ ) *auto*

**lemma** *wf-norm-ACI*:  $wf\ idx\ \langle \varphi \rangle = wf\ idx\ \varphi$

**by** (*induct*  $\varphi$  *arbitrary: idx*) (*auto simp: wf-NFOR wf-NFAND ball-Un wf-juncts[symmetric]*)

**lemma** *satisfies-bounded-disjuncts*:

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

**lemma** *satisfies-bounded-conjuncts*:

$\mathfrak{A} \models_b \varphi \longleftrightarrow (\forall \psi \in conjuncts\ \varphi.\ \mathfrak{A} \models_b \psi)$   
**by** (*induct*  $\varphi$  *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*  $\varphi$  *arbitrary:  $\mathfrak{A}$* )  
(*auto simp: satisfies-bounded-NFOR satisfies-bounded-NFAND*)  
*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*:  $\#_V ((SNOC\ x \sim^k) \mathfrak{A}) = \#_V \mathfrak{A}$

**by** (*induct*  $k$ ) (*auto simp: nvars-SNOC*)

**lemma** *wf-fold-rderiv*:  $wf\ idx\ \varphi \implies wf\ idx\ (fold\ rderiv\ (replicate\ k\ x)\ \varphi)$

**by** (*induct*  $k$  *arbitrary:  $\varphi$* ) (*auto simp: wf-rderiv*)

**lemma** *satisfies-bounded-fold-rderiv*:

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

## 2.6 Emptiness Check

**context**

**fixes**  $b :: bool$   
**and**  $idx :: 'n$   
**and**  $\psi :: ('a, 'k) aformula$

**begin**

**abbreviation** *fut-test*  $\equiv \lambda(\varphi, \Phi).\ \varphi \notin set\ \Phi$

**abbreviation** *fut-step*  $\equiv \lambda(\varphi, \Phi).\ (norm\ (rderiv\ (zero\ idx)\ \varphi), \varphi \# \Phi)$

**definition** *fut-derivs*  $k\ \varphi \equiv ((norm\ o\ rderiv\ (zero\ idx)) \sim^k) \varphi$

**lemma** *fut-derivs-Suc[simp]*:  $norm\ (rderiv\ (zero\ idx)\ (fut-derivs\ k\ \varphi)) = fut-derivs\ (Suc\ k)\ \varphi$

**unfolding** *fut-derivs-def* **by** *auto*

**definition** *fut-invariant* =

$(\lambda(\varphi, \Phi). \text{wf idx } \varphi \wedge (\forall \varphi \in \text{set } \Phi. \text{wf idx } \varphi) \wedge$   
 $(\exists k. \varphi = \text{fut-deriv } k \ \psi \wedge \Phi = \text{map } (\lambda i. \text{fut-deriv } i \ \psi) (\text{rev } [0 ..< k])))$

**definition** *fut-spec*  $\varphi\Phi \equiv (\forall \varphi \in \text{set } (\text{snd } \varphi\Phi). \text{wf idx } \varphi) \wedge$

$(\forall \mathfrak{A}. \#_V \mathfrak{A} = \text{idx} \longrightarrow$   
 $(\text{if } b \text{ then } (\exists k. (\text{SNOC } (\text{zero idx}) \rightsquigarrow k) \ \mathfrak{A} \models_b \psi) \longleftrightarrow (\exists \varphi \in \text{set } (\text{snd } \varphi\Phi). \mathfrak{A} \models_b \varphi)$   
 $\text{else } (\forall k. (\text{SNOC } (\text{zero idx}) \rightsquigarrow k) \ \mathfrak{A} \models_b \psi) \longleftrightarrow (\forall \varphi \in \text{set } (\text{snd } \varphi\Phi). \mathfrak{A} \models_b \varphi)))$

**definition** *fut-default* =

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

**lemma** *finite-fold-rderiv-zeros*:  $\text{finite } \{\langle \text{fold rderiv } (\text{replicate } k \ (\text{zero idx})) \ \psi \rangle \mid k. \text{True}\}$

**by**  $(\text{rule finite-subset}[OF - \text{finite-fold-rderiv}[of \ \psi]]) \text{ blast}$

**definition** *fut* ::  $(\text{'a}, \text{'k}) \text{ aformula where}$

$\text{fut} = (\text{if } b \text{ then } n\text{FOR} \text{ else } n\text{FAND}) (\text{snd } (\text{while-default fut-default fut-test fut-step } (\psi, [])))$

**context**

**assumes**  $\text{wf: wf idx } \psi$

**begin**

**lemma** *wf-fut-derivs*:

$\text{wf idx } (\text{fut-deriv } k \ \psi)$

**by**  $(\text{induct } k) (\text{auto simp: wf-norm wf-rderiv wf fut-derivs-def})$

**lemma** *satisfies-bounded-fut-derivs*:

$\#_V \mathfrak{A} = \text{idx} \implies \mathfrak{A} \models_b \text{fut-deriv } k \ \psi \longleftrightarrow (\text{SNOC } (\text{zero idx}) \rightsquigarrow k) \ \mathfrak{A} \models_b \psi$

**by**  $(\text{induct } k \text{ arbitrary: } \mathfrak{A}) (\text{auto simp: fut-derivs-def satisfies-bounded-rderiv satisfies-bounded-norm})$

$\text{wf-norm-rderivs size-zero nvars-SNOC funpow-swap1}[of \ \text{SNOC } x \ \text{for } x] \ \text{wf}$

**lemma** *fut-init*:  $\text{fut-invariant } (\psi, [])$

**unfolding** *fut-invariant-def* **by**  $(\text{auto simp: fut-derivs-def wf})$

**lemma** *fut-spec-default*:  $\text{fut-spec fut-default}$

**using** *satisfies-bounded-fold-rderiv*[OF *iffD2*[OF *wf-norm-ACI wf*] *sym size-zero*]

**unfolding** *fut-spec-def fut-default-def snd-conv*

*set-sorted-list-of-set*[OF *finite-fold-rderiv-zeros*]

**by**  $(\text{auto simp: satisfies-bounded-norm-ACI wf-fold-rderiv wf wf-norm-ACI simp del: fold-replicate})$

**lemma** *fut-invariant*:  $\text{fut-invariant } \varphi\Phi \implies \text{fut-test } \varphi\Phi \implies \text{fut-invariant } (\text{fut-step } \varphi\Phi)$

by (cases  $\varphi\Phi$ ) (auto simp: fut-invariant-def wf-norm wf-rderiv split: if-splits)

**lemma** fut-terminate: fut-invariant  $\varphi\Phi \implies \neg$  fut-test  $\varphi\Phi \implies$  fut-spec  $\varphi\Phi$   
**proof** (induct  $\varphi\Phi$ , unfold prod.case not-not)  
**fix**  $\varphi \Phi$  **assume** fut-invariant ( $\varphi, \Phi$ )  $\varphi \in$  set  $\Phi$   
**then obtain**  $i k$  **where**  $i < k$  **and**  $\varphi$ -def:  $\varphi =$  fut-deriv  $i \psi$   
**and**  $\Phi$ -def:  $\Phi =$  map ( $\lambda i.$  fut-deriv  $i \psi$ ) (rev  $[0..<k]$ )  
**and** \*: fut-deriv  $k \psi =$  fut-deriv  $i \psi$  **unfolding** fut-invariant-def **by** auto  
**have** set  $\Phi =$  {fut-deriv  $k \psi \mid k . True$ }  
**unfolding**  $\Phi$ -def set-map set-rev set-upt **proof** safe  
**fix**  $j$   
**show** fut-deriv  $j \psi \in$  ( $\lambda i.$  fut-deriv  $i \psi$ ) ‘  $\{0..<k\}$   
**proof** (cases  $j < k$ )  
**case** False  
**with** \*  $\langle i < k \rangle$  **have** fut-deriv  $j \psi =$  fut-deriv  $((j - i) \bmod (k - i) + i) \psi$   
**unfolding** fut-deriv-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-deriv **by** (auto simp: wf-fut-deriv)  
**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**  $\#_V \mathfrak{A} =$  idx  
**shows**  $\mathfrak{A} \models_b$  fut  $\longleftrightarrow$   
( $\text{if } b \text{ then } (\exists k. (SNOC (zero\ idx) \overset{\sim}{\sim} k) \mathfrak{A} \models_b \psi) \text{ else } (\forall k. (SNOC (zero\ idx) \overset{\sim}{\sim} k) \mathfrak{A} \models_b \psi)$ )  
**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**  $wf\ final\ ize$ :  $wf\ idx\ \varphi \implies wf\ idx\ (finalize\ idx\ \varphi)$   
**by** ( $induct\ \varphi\ arbitrary$ :  $idx$ ) ( $auto\ simp$ :  $wf\ fut\ wf\ nFEx\ wf\ nFAll$ )

**lemma**  $Length\ SNOCs$ :  $Length\ ((SNOC\ x\ \sim i)\ \mathfrak{A}) = Length\ \mathfrak{A} + i$   
**by** ( $induct\ i\ arbitrary$ :  $\mathfrak{A}$ ) ( $auto\ simp$ :  $Length\ SNOC$ )

**lemma**  $assigns\ SNOCs\ zero$ :  
 $\llbracket LESS\ k\ m\ (\#_V\ \mathfrak{A}); \#_V\ \mathfrak{A} = idx \rrbracket \implies m^{(SNOC\ (zero\ idx)\ \sim i)\ \mathfrak{A}_k} = m^{\mathfrak{A}_k}$   
**by** ( $induct\ i\ arbitrary$ :  $\mathfrak{A}$ ) ( $auto\ simp$ :  $assigns\ SNOC\ zero\ nvars\ SNOC\ fun\ pow\ swap1$ )

**lemma**  $Extend\ SNOCs\ zero\ satisfies$ :  $\llbracket wf\ (SUC\ k\ idx)\ \varphi; \#_V\ \mathfrak{A} = idx; lformula\ \varphi \rrbracket \implies$   
 $Extend\ k\ 0\ ((SNOC\ (zero\ (\#_V\ \mathfrak{A}))\ \sim i)\ \mathfrak{A})\ P \models \varphi \longleftrightarrow Extend\ k\ 0\ \mathfrak{A}\ P \models \varphi$   
**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**  $finalize\ satisfies$ :  $\llbracket wf\ idx\ \varphi; \#_V\ \mathfrak{A} = idx; lformula\ \varphi \rrbracket \implies \mathfrak{A} \models_b\ finalize\ idx\ \varphi \longleftrightarrow \mathfrak{A} \models \varphi$   
**by** ( $induct\ \varphi\ arbitrary$ :  $idx\ \mathfrak{A}$ )  
( $force\ simp\ add$ :  $wf\ nFEx\ wf\ nFAll\ wf\ final\ ize\ Length\ SNOCs\ nvars\ Extend\ nvars\ SNOCs$   
 $satisfies\ bounded\ fut\ satisfies\ bounded\ nFEx\ satisfies\ bounded\ nFAll\ Extend\ SNOCs\ zero\ satisfies$   
 $intro$ :  $le\ add2$ ) $+$

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

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

**lemma**  $nullable\ satisfies\ bounded$ :  $Length\ \mathfrak{A} = 0 \implies nullable\ \varphi \longleftrightarrow \mathfrak{A} \models_b\ \varphi$   
**by** ( $induct\ \varphi$ ) ( $auto\ simp$ :  $nullable0\ satisfies0\ Extend\ empty\ satisfies\ bounded$   
 $len\ cut\ 0$   
 $intro$ :  $exI[of\ -\ cut\ 0\ P\ for\ P]$ )

**lemma** *final-satisfies*:

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

## 2.7 Restrictions

**lemma** *satisfies-gen-restrict-RESTR*:

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

**lemma** *finite-FV: finite*  $(FV\ \varphi\ k)$

**by** (*induct*  $\varphi$ ) (*auto simp: finite-FV0*)

**lemma** *satisfies-gen-restrict*:

*satisfies-gen*  $r\ \mathfrak{A}\ \varphi \wedge (\forall x \in set\ V.\ restrict\ k\ (x^{\mathfrak{A}}k)) \longleftrightarrow$   
*satisfies-gen*  $r\ \mathfrak{A}\ (foldr\ (\lambda x.\ FAnd\ (Restrict\ k\ x))\ V\ \varphi)$   
**by** (*induct*  $V$  *arbitrary: phi*) (*auto simp: restrict-Restrict[symmetric]*)

**lemma** *sat-vars-RESTRICT-VARS*:

**fixes**  $\varphi$   
**defines**  $vs \equiv sorted-list-of-set\ o\ FV\ \varphi$   
**assumes**  $\forall k \in set\ ks.\ finite\ (FV\ \varphi\ k)$   
**shows** *sat-vars-gen*  $r\ (set\ ks)\ \mathfrak{A}\ \varphi \longleftrightarrow satisfies-gen\ r\ \mathfrak{A}\ (RESTRICT-VARS\ ks\ vs\ \varphi)$   
**using** *assms proof* (*induct*  $ks$ )  
**case**  $(Cons\ k\ ks)$   
**with** *satisfies-gen-restrict*[*of*  $r\ \mathfrak{A}\ (RESTRICT-VARS\ ks\ vs\ \varphi)\ vs\ k$ ] **show** *?case*  
**by** *auto*  
**qed** (*simp add: satisfies-gen-restrict-RESTR*)

**lemma** *sat-RESTRICT*: *sat*  $\mathfrak{A}\ \varphi \longleftrightarrow \mathfrak{A} \models RESTRICT\ \varphi$

**unfolding** *sat-def RESTRICT-def* **using** *sat-vars-RESTRICT-VARS*[*of*  $Enum.enum,$  *symmetric*]  
**by** (*auto simp: finite-FV enum-UNIV*)

**lemma** *sat<sub>b</sub>-RESTRICT*: *sat<sub>b</sub>*  $\mathfrak{A}\ \varphi \longleftrightarrow \mathfrak{A} \models_b RESTRICT\ \varphi$

**unfolding** *sat<sub>b</sub>-def RESTRICT-def* **using** *sat-vars-RESTRICT-VARS*[*of*  $Enum.enum,$  *symmetric*]  
**by** (*auto simp: finite-FV enum-UNIV*)

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

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

**lemma** *wf-RESTRICT-VARS*:  $\llbracket wf\ idx\ \varphi; \forall k \in set\ ks.\ \forall v \in set\ (vs\ k).\ LESS\ k\ v\ idx \rrbracket \implies$

*wf idx*  $(RESTRICT-VARS\ ks\ vs\ \varphi)$

**proof** (*induct*  $ks$ )

**case** (*Cons k ks*)  
**moreover**  
 { **fix** *vs*  $\varphi$  **assume**  $\forall v \in \text{set } vs. \text{LESS } k \ v \ \text{idx } wf \ \text{idx } \varphi$   
   **then have** *wf idx* (*foldr* ( $\lambda x. \text{FAnd } (\text{Restrict } k \ x)$ ) *vs*  $\varphi$ )  
   **by** (*induct vs arbitrary:  $\varphi$* ) (*auto simp: wf-Restrict*)  
 }  
**ultimately show** *?case* **by** *auto*  
**qed** (*simp add: wf-RESTR*)

**lemma** *wf-FV-LESS*:  $\llbracket wf \ \text{idx } \varphi; v \in \text{FV } \varphi \ k \rrbracket \implies \text{LESS } k \ v \ \text{idx}$   
**by** (*induct  $\varphi$  arbitrary: idx v*)  
 (*force simp: wf0-FV0-LESS LESS-SUC split: if-splits*)<sup>+</sup>

**lemma** *wf-RESTRIC*T:  $wf \ \text{idx } \varphi \implies wf \ \text{idx } (\text{RESTRIC}T \ \varphi)$   
**unfolding** *RESTRIC*T-def **by** (*rule wf-RESTRIC*-VARS) (*auto simp: list-all-iff wf-FV-LESS finite-FV*)

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

**lemma** *lformula-RESTRIC*-VARS:  $lformula \ \varphi \implies lformula \ (\text{RESTRIC-VARS *ks vs*  $\varphi$ )$

**proof** (*induct ks*)  
**case** (*Cons k ks*)  
**moreover**  
 { **fix** *vs*  $\varphi$  **assume**  $lformula \ \varphi$   
   **then have**  $lformula \ (\text{foldr } (\lambda x. \text{FAnd } (\text{Restrict } k \ x)) \ vs \ \varphi)$   
   **by** (*induct vs arbitrary:  $\varphi$* ) (*auto simp: lformula-Restrict*)  
 }  
**ultimately show** *?case* **by** *auto*  
**qed** (*simp add: lformula-RESTR*)

**lemma** *lformula-RESTRIC*T:  $lformula \ \varphi \implies lformula \ (\text{RESTRIC}T \ \varphi)$   
**unfolding** *RESTRIC*T-def **by** (*rule lformula-RESTRIC*-VARS)

**lemma** *ex-fold-CONS*:  $\exists xs \ \mathfrak{B}. \ \mathfrak{A} = \text{fold } \text{CONS } xs \ \mathfrak{B} \wedge \text{Length } \mathfrak{B} = 0 \wedge \text{Length}$   
 $\mathfrak{A} = \text{length } xs \wedge$

$\#_V \ \mathfrak{B} = \#_V \ \mathfrak{A} \wedge (\forall x \in \text{set } xs. \ \text{size } x = \#_V \ \mathfrak{A})$

**proof** (*induct Length  $\mathfrak{A}$  arbitrary:  $\mathfrak{A}$* )

**case** (*Suc m*)

**from** *Suc(2) CONS-surj* **obtain** *a*  $\mathfrak{B}$  **where**  $\mathfrak{A} = \text{CONS } a \ \mathfrak{B} \ \#_V \ \mathfrak{B} = \#_V \ \mathfrak{A}$   
*size a =  $\#_V \ \mathfrak{A}$*  **by** *force*

**moreover with** *Suc(2)* **have**  $\text{Length } \mathfrak{B} = m$  **by** (*simp add: Length-CONS*)

**with** *Suc(1)[of  $\mathfrak{B}$ ]* **obtain** *xs*  $\mathfrak{C}$  **where**  $\mathfrak{B} = \text{fold } \text{CONS } xs \ \mathfrak{C} \ \text{Length } \mathfrak{C} = 0$   
 $\text{Length } \mathfrak{B} = \text{length } xs$

$\#_V \ \mathfrak{C} = \#_V \ \mathfrak{B} \ \forall x \in \text{set } xs. \ \text{size } x = \#_V \ \mathfrak{B}$  **by** *blast*

**ultimately show** *?case* **by** (*intro exI[of - xs @ [a]] exI[of -  $\mathfrak{C}$ ]*) (*auto simp: Length-CONS*)

**qed** *simp*

**primcorec**  $L$  where

$L \text{ idx } I = \text{Lang } (\exists \mathfrak{A}. \text{Length } \mathfrak{A} = 0 \wedge \#_V \mathfrak{A} = \text{idx} \wedge \mathfrak{A} \in I)$   
*( $\lambda a$ . if size  $a = \text{idx}$  then  $L \text{ idx } \{\mathfrak{B}. \text{CONS } a \ \mathfrak{B} \in I\}$  else Zero)*

**lemma**  $L\text{-empty}$ :  $L \text{ idx } \{\} = \text{Zero}$   
**by** *coinduction auto*

**lemma**  $L\text{-alt}$ :  $L \text{ idx } I =$

*to-language*  $\{xs. \exists \mathfrak{A} \in I. \exists \mathfrak{B}. \mathfrak{A} = \text{fold } \text{CONS } (\text{rev } xs) \ \mathfrak{B} \wedge \text{Length } \mathfrak{B} = 0 \wedge \#_V \ \mathfrak{B} = \text{idx} \wedge (\forall x \in \text{set } xs. \text{size } x = \text{idx})\}$

**by** *(coinduction arbitrary: I)*

*(auto 0 4 simp: L-empty intro: exI[of - {}] arg-cong[of - - to-language])*

**definition**  $\text{lang idx } \varphi = L \text{ idx } \{\mathfrak{A}. \mathfrak{A} \models \varphi \wedge \#_V \mathfrak{A} = \text{idx}\}$

**definition**  $\text{lang}_b \text{ idx } \varphi = L \text{ idx } \{\mathfrak{A}. \mathfrak{A} \models_b \varphi \wedge \#_V \mathfrak{A} = \text{idx}\}$

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

**definition**  $\text{language}_b \text{ idx } \varphi = L \text{ idx } \{\mathfrak{A}. \text{sat}_b \mathfrak{A} \ \varphi \wedge \#_V \mathfrak{A} = \text{idx}\}$

**lemma**  $\text{lformula } \varphi \implies \text{lang } n \ (\text{norm } \varphi) = \text{lang } n \ \varphi$   
**unfolding**  $\text{lang-def}$  **using**  $\text{satisfies-norm}$  **by** *auto*

**lemma**  $\text{in-language-Zero[simp]}$ :  $\neg \text{in-language Zero } w$   
**by** *(induct w) auto*

**lemma**  $\text{in-language-L-size}$ :  $\text{in-language } (L \text{ idx } I) \ w \implies x \in \text{set } w \implies \text{size } x = \text{idx}$

**by** *(induct w arbitrary: x I) (auto split: if-splits)*

**end**

**sublocale**  $\text{Formula } <$

*bounded*:  $\text{DA alphabet idx } \lambda \varphi. \text{norm } (\text{RESTRICT } \varphi) \ \lambda a \ \varphi. \text{norm } (\text{lderiv } a \ \varphi)$   
*nullable*

$\lambda \varphi. \text{wf idx } \varphi \wedge \text{lformula } \varphi \ \text{lang}_b \ \text{idx}$

$\lambda \varphi. \text{wf idx } \varphi \wedge \text{lformula } \varphi \ \text{language}_b \ \text{idx}$  **for**  $\text{idx}$

**using**  $\text{ex-Length-0[of idx]}$

**by** *unfold-locales*

*(auto simp: lformula-norm lformula-lderiv distinct-alphabet alphabet-size wf-norm wf-lderiv*

*lang\_b-def language\_b-def nullable-satisfies-bounded wf-RESTRICT lformula-RESTRICT sat\_b-RESTRICT*

*satisfies-bounded-norm in-language-L-size satisfies-bounded-lderiv nvars-CONS*

*dest: Length-0-inj intro: arg-cong[of - - L (size -)])*

**sublocale**  $\text{Formula } <$

*unbounded?*:  $\text{DA alphabet idx } \lambda \varphi. \text{norm } (\text{RESTRICT } \varphi) \ \lambda a \ \varphi. \text{norm } (\text{lderiv } a \ \varphi)$   
*final idx*

$\lambda \varphi. \text{wf idx } \varphi \wedge \text{lformula } \varphi \ \text{lang } \text{idx}$

$\lambda \varphi. \text{wf } \text{idx } \varphi \wedge \text{lformula } \varphi \text{ language } \text{idx } \mathbf{for } \text{idx}$   
**using** *ex-Length-0*[of *idx*]  
**by** *unfold-locales*  
*(auto simp: lformula-norm lformula-lderiv distinct-alphabet alphabet-size wf-norm wf-lderiv lang-def language-def final-satisfies wf-RESTRICT lformula-RESTRICT sat-RESTRICT satisfies-norm in-language-L-size satisfies-lderiv nvars-CONS dest: Length-0-inj intro: arg-cong[of - - L (size -)])*

**lemma** (in *Formula*) *check-equiv-soundness*:  
 $\llbracket \#_V \mathfrak{A} = \text{idx}; \text{check-equiv } \text{idx } \varphi \psi \rrbracket \implies \text{sat } \mathfrak{A} \varphi \longleftrightarrow \text{sat } \mathfrak{A} \psi$   
**using** *ex-fold-CONS*[of  $\mathfrak{A}$ ]  
**by** (*auto simp: language-def L-alt set-eq-iff dest!: soundness[unfolded rel-language-eq] injD[OF bij-is-inj[OF to-language-bij]]*)  
*(metis Length-0-inj rev-rev-ident set-rev)+*

**lemma** (in *Formula*) *bounded-check-equiv-soundness*:  
 $\llbracket \#_V \mathfrak{A} = \text{idx}; \text{bounded.check-equiv } \text{idx } \varphi \psi \rrbracket \implies \text{sat}_b \mathfrak{A} \varphi \longleftrightarrow \text{sat}_b \mathfrak{A} \psi$   
**using** *ex-fold-CONS*[of  $\mathfrak{A}$ ]  
**by** (*auto simp: language\_b-def L-alt set-eq-iff dest!: bounded.soundness[unfolded rel-language-eq] injD[OF bij-is-inj[OF to-language-bij]]*)  
*(metis Length-0-inj rev-rev-ident set-rev)+*

**end**

### 3 WS1S Interpretations

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

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

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

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

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

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

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

**datatype** *order* = *FO* | *SO*

**derive** *linorder* *order*

**instantiation** *order* :: *enum* **begin**

**definition** *enum-order* = [*FO*, *SO*]

**definition** *enum-all-order*  $P = (P \ \text{FO} \wedge P \ \text{SO})$

**definition** *enum-ex-order*  $P = (P \ \text{FO} \vee P \ \text{SO})$

**lemmas** *enum-defs* = *enum-order-def enum-all-order-def enum-ex-order-def*

**instance** **proof** **qed** (*auto simp: enum-defs, (metis (full-types) order.exhaust)+*)

**end**

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

**setup-lifting** *type-definition-idx*

**lift-definition** *SUC* :: order ⇒ idx ⇒ idx is  
 λord (m, n). case ord of FO ⇒ (Suc m, n) | SO ⇒ (m, Suc n) .  
**lift-definition** *LESS* :: order ⇒ nat ⇒ idx ⇒ bool is  
 λord l (m, n). case ord of FO ⇒ l < m | SO ⇒ l < n .  
**abbreviation** *LEQ* ord l idx ≡ *LESS* ord l (*SUC* ord idx)

**definition** *MSB* *Is* ≡  
 if ∀ P ∈ set *Is*. P = {} then 0 else Suc (Max (⋃ P ∈ set *Is*. fset P))

**lemma** *MSB-Nil*[simp]: *MSB* [] = 0  
**unfolding** *MSB-def* by simp

**lemma** *MSB-Cons*[simp]: *MSB* (I # *Is*) = max (if I = {} then 0 else Suc (fMax I)) (*MSB* *Is*)  
**unfolding** *MSB-def* including fset.lifting  
**by** transfer (auto simp: Max-Un list-all-iff Sup-bot-conv(2)[symmetric] simp del: Sup-bot-conv(2))

**lemma** *MSB-append*[simp]: *MSB* (I1 @ I2) = max (*MSB* I1) (*MSB* I2)  
**by** (induct I1) auto

**lemma** *MSB-insert-nth*[simp]:  
*MSB* (insert-nth n P *Is*) = max (if P = {} then 0 else Suc (fMax P)) (*MSB* *Is*)  
**by** (subst (2) append-take-drop-id[of n *Is*, symmetric])  
 (simp only: insert-nth-take-drop *MSB-append* *MSB-Cons* *MSB-Nil*)

**lemma** *MSB-greater*:  
 [i < length *Is*; p |∈| *Is* ! i] ⇒ p < *MSB* *Is*  
**unfolding** *MSB-def* by (fastforce simp: Bex-def in-set-conv-nth less-Suc-eq-le intro: Max-ge)

**lemma** *MSB-mono*: set I1 ⊆ set I2 ⇒ *MSB* I1 ≤ *MSB* I2  
**unfolding** *MSB-def* including fset.lifting  
**by** transfer (auto simp: list-all-iff intro!: Max-ge)

**lemma** *MSB-map-index'-CONS*[simp]:  
*MSB* (map-index' i (lift bs) *Is*) =  
 (if *MSB* *Is* = 0 ∧ (∀ i ∈ {i ..< i + length *Is*}. ¬ 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* ≤ n ⇒ *MSB* (map-index' i (snoc n bs) *Is*) =  
 (if (∀ i ∈ {i ..< i + length *Is*}. ¬ 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) | 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 ( $\langle$ - $\rightarrow$  [900, 999, 999]*  
*999)*

**is**  $\lambda n$  (*-, I1, I2*) *ord. case ord of FO  $\Rightarrow$  if n < length I1 then I1 ! n else {||}*  
*| SO  $\Rightarrow$  if n < length I2 then I2 ! n else {||} .*

**lift-definition** *nvars :: interp  $\Rightarrow$  idx ( $\langle$ # $\vee$   $\rightarrow$  [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)*  
*| 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*)

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

**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) | SO  $\Rightarrow$  (bs1, b # bs2)*

**lift-definition** *size-atom :: bool list  $\times$  bool list  $\Rightarrow$  idx*

**is**  $\lambda$ (*bs1, bs2*). (*length bs1, length bs2*) .

**lift-definition**  $\sigma :: \text{idx} \Rightarrow \text{atom list}$   
**is**  $(\lambda(n, N). \text{map } (\lambda bs. (\text{take } n \text{ bs}, \text{drop } n \text{ bs})) (\text{List.n-lists } (n + N) [\text{True}, \text{False}])))$   
.

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

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

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

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

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

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

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

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

**lemma**  $assigns\text{-Extend}[simp]$ :  
 $LEQ \ \text{ord} \ i \ (\#_V \ \mathfrak{A}) \implies m^{\text{Extend } \text{ord} \ i \ \mathfrak{A} \ P} \ \text{ord} = (\text{if } m = i \ \text{then } P \ \text{else } (\text{if } m > i \ \text{then } m - \text{Suc } 0 \ \text{else } m)^{\mathfrak{A}} \ \text{ord})$   
 $\text{ord} \neq \text{ord}' \implies m^{\text{Extend } \text{ord} \ i \ \mathfrak{A} \ P} \ \text{ord}' = m^{\mathfrak{A}} \ \text{ord}'$   
**by**  $(\text{transfer, force simp: } \text{min}\text{-def } \text{nth}\text{-append split: } \text{order.splits})+$

**lemma**  $Extend\text{-commute}\text{-safe}[simp]$ :  
 $\llbracket j \leq i; LEQ \ \text{ord} \ i \ (\#_V \ \mathfrak{A}) \rrbracket \implies$   
 $\text{Extend } \text{ord} \ j \ (\text{Extend } \text{ord} \ i \ \mathfrak{A} \ P1) \ P2 = \text{Extend } \text{ord} \ (\text{Suc } i) \ (\text{Extend } \text{ord} \ j \ \mathfrak{A})$



*P2*) *P1*  
**by** (*transfer*,  
*force simp del: insert-nth-take-drop simp: replicate-add[symmetric] split: order.splits*)

**lemma** *Extend-commute-unsafe*:  
 $ord \neq ord' \implies \text{Extend } ord \ j \ (\text{Extend } ord' \ i \ \mathfrak{A} \ P1) \ P2 = \text{Extend } ord' \ i \ (\text{Extend } ord \ j \ \mathfrak{A} \ P2) \ P1$   
**by** (*transfer*, *force simp: replicate-add[symmetric] split: order.splits*)

**lemma** *Length-CONS[simp]*:  
 $\text{Length} \ (\text{CONS } x \ \mathfrak{A}) = \text{Suc} \ (\text{Length} \ \mathfrak{A})$   
**by** (*transfer*, *force split: order.splits*)

**lemma** *Length-SNOC[simp]*:  
 $\text{Length} \ (\text{SNOC } x \ \mathfrak{A}) = \text{Suc} \ (\text{Length} \ \mathfrak{A})$   
**by** (*transfer*, *force simp: Let-def split: order.splits*)

**lemma** *nvars-CONS[simp]*:  
 $\#_V \ (\text{CONS } x \ \mathfrak{A}) = \#_V \ \mathfrak{A}$   
**by** (*transfer*, *force*)

**lemma** *nvars-SNOC[simp]*:  
 $\#_V \ (\text{SNOC } x \ \mathfrak{A}) = \#_V \ \mathfrak{A}$   
**by** (*transfer*, *force simp: Let-def*)

**lemma** *assigns-CONS[simp]*:  
**assumes**  $\#_V \ \mathfrak{A} = \text{size-atom } bs1\text{-}bs2$   
**shows**  $\text{LESS } ord \ x \ (\#_V \ \mathfrak{A}) \implies x^{\text{CONS } bs1\text{-}bs2 \ \mathfrak{A} \ ord} =$   
*(if case-prod case-order bs1-bs2 ord ! x then finsert 0 (upshift (x<sup>ord</sup>)) else upshift (x<sup>ord</sup>))*  
**by** (*insert assms, transfer*) (*auto simp: lift-def split: order.splits*)

**lemma** *assigns-SNOC[simp]*:  
**assumes**  $\#_V \ \mathfrak{A} = \text{size-atom } bs1\text{-}bs2$   
**shows**  $\text{LESS } ord \ x \ (\#_V \ \mathfrak{A}) \implies x^{\text{SNOC } bs1\text{-}bs2 \ \mathfrak{A} \ ord} =$   
*(if case-prod case-order bs1-bs2 ord ! x then finsert (Length  $\mathfrak{A}$ ) (x<sup>ord</sup>) else x<sup>ord</sup>)*  
**by** (*insert assms, transfer*) (*force simp: snoc-def Let-def split: order.splits*)

**lemma** *map-index'-eq-conv[simp]*:  
 $\text{map-index}' \ i \ f \ xs = \text{map-index}' \ j \ g \ xs = (\forall k < \text{length } xs. f \ (i + k) \ (xs \ ! \ k) = g \ (j + k) \ (xs \ ! \ k))$   
**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]*:  $\text{Suc } m \ |\in| \ P \implies \text{fMax} \ (P \ |-| \ \{0\}) = \text{fMax} \ P$

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  $Suc\ (fMax\ ((\lambda x.\ x - Suc\ 0)\ |\cdot|\ P)) = fMax\ P$

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-rotate-simp*:  $rotate\ m\ x\ !\ i = (if\ i < m\ then\ x\ else\ []\ !\ (i - m))$

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 split: if-splits*)

**lemma** *append-rotate-inj*:

assumes  $xs \neq [] \implies last\ xs \neq x$  and  $ys \neq [] \implies last\ ys \neq x$

shows  $xs @ rotate\ m\ x = ys @ rotate\ n\ x \longleftrightarrow (xs = ys \wedge m = n)$

proof *safe*

**from** *assms* **have** *assms'*:  $xs \neq [] \implies \text{rev } xs ! 0 \neq x \text{ } ys \neq [] \implies \text{rev } ys ! 0 \neq x$   
**by** (*auto simp: hd-rev hd-conv-nth[symmetric]*)  
**assume** \*:  $xs @ \text{replicate } m \ x = ys @ \text{replicate } n \ x$   
**then have**  $\text{rev } (xs @ \text{replicate } m \ x) = \text{rev } (ys @ \text{replicate } n \ x)$  ..  
**then have**  $\text{replicate } m \ x @ \text{rev } xs = \text{replicate } n \ x @ \text{rev } ys$  **by** *simp*  
**then have**  $\text{take } (\max \ m \ n) (\text{replicate } m \ x @ \text{rev } xs) = \text{take } (\max \ m \ n) (\text{replicate } n \ x @ \text{rev } ys)$  **by** *simp*  
**then have**  $\text{replicate } m \ x @ \text{take } (\max \ m \ n - m) (\text{rev } xs) = \text{replicate } n \ x @ \text{take } (\max \ m \ n - n) (\text{rev } ys)$  **by** (*auto simp: min-def max-def split: if-splits*)  
**then have**  $(\text{replicate } m \ x @ \text{take } (\max \ m \ n - m) (\text{rev } xs)) ! \min \ m \ n = (\text{replicate } n \ x @ \text{take } (\max \ m \ n - n) (\text{rev } ys)) ! \min \ m \ n$  **by** *simp*  
**with** *arg-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** *fin-lift[simp]*:  $m \in | \text{lift } bs \ i \ (I ! \ i) \longleftrightarrow (\text{case } m \ \text{of } 0 \Rightarrow bs ! \ i \mid \text{Suc } m \Rightarrow m \in | \ I ! \ i)$   
**unfolding** *lift-def upshift-def* **by** (*auto split: nat.splits*)

**lemma** *ex-Length-0[simp]*:  
 $\exists \mathfrak{A}. \text{Length } \mathfrak{A} = 0 \wedge \#_V \ \mathfrak{A} = \text{idx}$   
**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- $\sigma$ -length-atom[simp]*:  $(x \in \text{set } (\sigma \ \text{idx})) \longleftrightarrow \text{idx} = \text{size-atom } x$   
**by** *transfer (auto simp: set-n-lists enum-UNIV image-iff intro!: exI[of - I1 @ I2 for I1 I2])*

**lemma** *distinct- $\sigma$ [simp]*: *distinct*  $(\sigma \ \text{idx})$   
**by** *transfer (auto 0 4 simp: distinct-map distinct-n-lists set-n-lists inj-on-def intro: iffD2[OF append-eq-append-conv] box-equals[OF - append-take-drop-id append-take-drop-id, of n - n for n])*

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

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

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

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

**lemma** *max-Length-fMax[simp]*:  $x \in | m1^{\mathfrak{A}}k \implies \max (\text{Length } \mathfrak{A}) (fMax (m1^{\mathfrak{A}}k)) = \text{Length } \mathfrak{A}$   
**using** *fMax-less-Length[of x m1 \mathfrak{A} k]* **unfolding** *fMax-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]*:  $LESS \text{ ord } m (\#_V \mathfrak{A}) \implies \neg \text{case-prod case-order (zero } (\#_V \mathfrak{A})) \text{ ord } ! m$   
**by** *transfer (auto split: order.splits)*

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

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

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

**lemma** *Suc-in-fimage-Suc*:  $\text{Suc } x \in | \text{Suc } |^{\cdot} X \longleftrightarrow x \in | X$   
**by** *auto*

**lemma** *Suc-in-fimage-Suc-o-Suc[simp]*:  $\text{Suc } x \in | (\text{Suc } \circ \text{Suc}) |^{\cdot} X \longleftrightarrow x \in | \text{Suc } |^{\cdot} X$   
**by** *auto*

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

**lemma** *fimage-Suc-o-Suc-eq-fimage-Suc-iff[simp]*:

$((\text{Suc} \circ \text{Suc}) \mid \uparrow X = \text{Suc} \mid \uparrow Y) \longleftrightarrow (\text{Suc} \mid \uparrow X = Y)$   
**by** (*metis fimage-Suc-inj fset.map-comp*)

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

**lemma** *fimage-Suc-eq-singleton[simp]*:  $(\text{fimage} \text{Suc} Z = \{|y|\}) \longleftrightarrow (\exists x. Z = \{|x|\}) \wedge \text{Suc} x = y$   
**by** (*cases y*) *auto*

**lemma** *len-downshift-helper*:  
 $x \mid \in P \implies \text{Suc} (\text{fMax} ((\lambda x. x - \text{Suc} 0) \mid \uparrow (P \mid - \{|0|\}))) \neq \text{fMax} P \implies xa \mid \in P \implies xa = 0$

**proof** –

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

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

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

**moreover**

{ **assume**  $xa \notin \{|0|\}$

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

**then obtain** *esk1<sub>1</sub>* :: *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**

**lemma** *CONS-inj[simp]*:  $\text{size-atom } x = \#_V \mathfrak{A} \implies \text{size-atom } y = \#_V \mathfrak{A} \implies \#_V \mathfrak{A} = \#_V \mathfrak{B} \implies$

$\text{CONS } x \mathfrak{A} = \text{CONS } y \mathfrak{B} \longleftrightarrow (x = y \wedge \mathfrak{A} = \mathfrak{B})$

**by** *transfer* (*auto simp: list-eq-iff-nth-eq lift-def upshift-def split: if-splits; blast*)

**lemma** *Suc-minus1*:  $\text{Suc} (x - \text{Suc} 0) = (\text{if } x = 0 \text{ then } \text{Suc} 0 \text{ else } x)$

**by** *auto*

**lemma** *fset-eq-empty-iff*:  $(\text{fset } X = \{\}) = (X = \{|\})$

**by** (*metis bot-fset.rep-eq fset-inverse*)

**lemma** *fset-le-singleton-iff*:  $(\text{fset } X \subseteq \{x\}) = (X = \{|\} \vee X = \{|x|\})$

**by** (*metis finsert.rep-eq fset-eq-empty-iff fset-inject order-refl singleton-insert-inj-eq subset-singletonD*)

**lemma** *MSB-decreases*:

$\text{MSB } I \leq \text{Suc } m \implies \text{MSB} (\text{map} (\lambda X. (\lambda I1. I1 - \text{Suc} 0) \mid \uparrow (X \mid - \{|0|\}))) I \leq m$

**unfolding** *MSB-def*

**by** (*auto simp add: not-le less-Suc-eq-le fset-eq-empty-iff fset-le-singleton-iff*

*split: if-splits dest!: iffD1[OF Max-le-iff, rotated -1] iffD1[OF Max-ge-iff, rotated -1]; force*)

**lemma** *CONS-surj*[*dest*]:  
**assumes** *Length*  $\mathfrak{A} > 0$   
**shows**  $\exists x \mathfrak{B}. \mathfrak{A} = \text{CONS } x \mathfrak{B} \wedge \#_V \mathfrak{B} = \#_V \mathfrak{A} \wedge \text{size-atom } x = \#_V \mathfrak{A}$   
**proof** –  
**have**  $\text{MSB } I1 \leq \text{Suc } m \implies \text{MSB } I2 \leq \text{Suc } m \implies \exists a b \text{ ab } ba.$   
 $(\exists I1. \text{length } ab = \text{length } I1 \wedge (\forall i < \text{length } ab. ab ! i = I1 ! i) \wedge$   
 $(\exists I2. \text{length } ba = \text{length } I2 \wedge (\forall i < \text{length } ba. ba ! i = I2 ! i) \wedge$   
 $\text{MSB } I1 \leq m \wedge \text{MSB } I2 \leq m)) \wedge$   
 $I1 = \text{map-index (lift } a) \text{ ab} \wedge$   
 $I2 = \text{map-index (lift } b) \text{ ba} \wedge$   
 $\text{length } ab = \text{length } I1 \wedge$   
 $\text{length } ba = \text{length } I2 \wedge$   
 $\text{length } a = \text{length } I1 \wedge$   
 $\text{length } b = \text{length } I2$   
**for**  $m \ I1 \ I2$   
**by** (*auto simp: list-eq-iff-nth-eq lift-def upshift-def split: if-splits*  
*intro!: exI[of - map ( $\lambda X. 0 \mid \in \mid X$ ) -] exI[of - map ( $\lambda X. (\lambda x. x - \text{Suc } 0) \mid \in \mid X \mid - \mid \{ \mid 0 \} \}$ ) -],*  
*auto simp: MSB-decreases upshift-def Suc-minus1 fimage-iff intro: rev-fBexI*  
*split: if-splits*)  
**with** *assms show ?thesis*  
**by** *transfer (auto simp: gr0-conv-Suc list-eq-iff-nth-eq)*  
**qed**

## 4 Concrete Atomic WS1S Formulas (Minimum Semantics for FO Variables)

**datatype** (*FOV0*: 'fo, *SOV0*: 'so) *atomic* =  
*Fo* 'fo |  
*Eq-Const* nat option 'fo nat |  
*Less* bool option 'fo 'fo |  
*Plus-FO* nat option 'fo 'fo nat |  
*Eq-FO* bool 'fo 'fo |  
*Eq-SO* 'so 'so |  
*Suc-SO* bool bool 'so 'so |  
*Empty* 'so |  
*Singleton* 'so |  
*Subset* 'so 'so |  
*In* bool 'fo 'so |  
*Eq-Max* bool 'fo 'so |  
*Eq-Min* bool 'fo 'so |  
*Eq-Union* 'so 'so 'so |  
*Eq-Inter* 'so 'so 'so |  
*Eq-Diff* 'so 'so 'so |  
*Disjoint* 'so 'so |  
*Eq-Presb* nat option 'so nat

**derive** *linorder option*  
**derive** *linorder atomic* — very slow

**type-synonym** *fo = nat*  
**type-synonym** *so = nat*  
**type-synonym** *ws1s = (fo, so) atomic*  
**type-synonym** *formula = (ws1s, order) aformula*

**primrec** *wf0 where*

*wf0 idx (Fo m) = LESS FO m idx*  
| *wf0 idx (Eq-Const i m n) = (LESS FO m idx  $\wedge$  (case i of Some i  $\Rightarrow$  i  $\leq$  n | -  $\Rightarrow$  True))*  
| *wf0 idx (Less - m1 m2) = (LESS FO m1 idx  $\wedge$  LESS FO m2 idx)*  
| *wf0 idx (Plus-FO i m1 m2 n) = (LESS FO m1 idx  $\wedge$  LESS FO m2 idx  $\wedge$  (case i of Some i  $\Rightarrow$  i  $\leq$  n | -  $\Rightarrow$  True))*  
| *wf0 idx (Eq-FO - m1 m2) = (LESS FO m1 idx  $\wedge$  LESS FO m2 idx)*  
| *wf0 idx (Eq-SO M1 M2) = (LESS SO M1 idx  $\wedge$  LESS SO M2 idx)*  
| *wf0 idx (Suc-SO br bl M1 M2) = (LESS SO M1 idx  $\wedge$  LESS SO M2 idx)*  
| *wf0 idx (Empty M) = LESS SO M idx*  
| *wf0 idx (Singleton M) = LESS SO M idx*  
| *wf0 idx (Subset M1 M2) = (LESS SO M1 idx  $\wedge$  LESS SO M2 idx)*  
| *wf0 idx (In - m M) = (LESS FO m idx  $\wedge$  LESS SO M idx)*  
| *wf0 idx (Eq-Max - m M) = (LESS FO m idx  $\wedge$  LESS SO M idx)*  
| *wf0 idx (Eq-Min - m M) = (LESS FO m idx  $\wedge$  LESS SO M idx)*  
| *wf0 idx (Eq-Union M1 M2 M3) = (LESS SO M1 idx  $\wedge$  LESS SO M2 idx  $\wedge$  LESS SO M3 idx)*  
| *wf0 idx (Eq-Inter M1 M2 M3) = (LESS SO M1 idx  $\wedge$  LESS SO M2 idx  $\wedge$  LESS SO M3 idx)*  
| *wf0 idx (Eq-Diff M1 M2 M3) = (LESS SO M1 idx  $\wedge$  LESS SO M2 idx  $\wedge$  LESS SO M3 idx)*  
| *wf0 idx (Disjoint M1 M2) = (LESS SO M1 idx  $\wedge$  LESS SO M2 idx)*  
| *wf0 idx (Eq-Presb - M n) = LESS SO M idx*

**inductive** *lformula0 where*

*lformula0 (Fo m)*  
| *lformula0 (Eq-Const None m n)*  
| *lformula0 (Less None m1 m2)*  
| *lformula0 (Plus-FO None m1 m2 n)*  
| *lformula0 (Eq-FO False m1 m2)*  
| *lformula0 (Eq-SO M1 M2)*  
| *lformula0 (Suc-SO False bl M1 M2)*  
| *lformula0 (Empty M)*  
| *lformula0 (Singleton M)*  
| *lformula0 (Subset M1 M2)*  
| *lformula0 (In False m M)*  
| *lformula0 (Eq-Max False m M)*  
| *lformula0 (Eq-Min False m M)*  
| *lformula0 (Eq-Union M1 M2 M3)*

```

| lformula0 (Eq-Inter M1 M2 M3)
| lformula0 (Eq-Diff M1 M2 M3)
| lformula0 (Disjoint M1 M2)
| lformula0 (Eq-Presb None M n)

```

**code-pred** *lformula0* .

**declare** *lformula0.intros*[simp]

**inductive-cases** *lformula0E*[elim]: *lformula0 a*

**abbreviation** *FV0*  $\equiv$  *case-order FOV0 SOV0*

**fun** *find0* **where**

```

  find0 FO i (Fo m) = (i = m)
| find0 FO i (Eq-Const - m -) = (i = m)
| find0 FO i (Less - m1 m2) = (i = m1  $\vee$  i = m2)
| find0 FO i (Plus-FO - m1 m2 -) = (i = m1  $\vee$  i = m2)
| find0 FO i (Eq-FO - m1 m2) = (i = m1  $\vee$  i = m2)
| find0 SO i (Eq-SO M1 M2) = (i = M1  $\vee$  i = M2)
| find0 SO i (Suc-SO - - M1 M2) = (i = M1  $\vee$  i = M2)
| find0 SO i (Empty M) = (i = M)
| find0 SO i (Singleton M) = (i = M)
| find0 SO i (Subset M1 M2) = (i = M1  $\vee$  i = M2)
| find0 FO i (In - m -) = (i = m)
| find0 SO i (In - - M) = (i = M)
| find0 FO i (Eq-Max - m -) = (i = m)
| find0 SO i (Eq-Max - - M) = (i = M)
| find0 FO i (Eq-Min - m -) = (i = m)
| find0 SO i (Eq-Min - - M) = (i = M)
| find0 SO i (Eq-Union M1 M2 M3) = (i = M1  $\vee$  i = M2  $\vee$  i = M3)
| find0 SO i (Eq-Inter M1 M2 M3) = (i = M1  $\vee$  i = M2  $\vee$  i = M3)
| find0 SO i (Eq-Diff M1 M2 M3) = (i = M1  $\vee$  i = M2  $\vee$  i = M3)
| find0 SO i (Disjoint M1 M2) = (i = M1  $\vee$  i = M2)
| find0 SO i (Eq-Presb - M -) = (i = M)
| find0 - - - = False

```

**abbreviation** *decr0 ord k*  $\equiv$  *map-atomic (case-order (dec k) id ord) (case-order id (dec k) ord)*

**lemma** *sum-pow2-image-Suc*:

```

finite X  $\implies$  sum (( $\wedge$ ) (2 :: nat)) (Suc ' X) = 2 * sum (( $\wedge$ ) 2) X
by (induct X rule: finite-induct) (auto intro: trans[OF sum.insert])

```

**lemma** *sum-pow2-insert0*:

```

[[finite X; 0  $\notin$  X]]  $\implies$  sum (( $\wedge$ ) (2 :: nat)) (insert 0 X) = Suc (sum (( $\wedge$ ) 2) X)
by (induct X rule: finite-induct) (auto intro: trans[OF sum.insert])

```

**lemma** *sum-pow2-upto*:  $\text{sum } ((\wedge) (2 :: \text{nat})) \{0 ..< x\} = 2^x - 1$



by (induct x) (auto simp: algebra-simps)

**lemma** *sum-pow2-inj*:

$\llbracket \text{finite } X; \text{ finite } Y; (\sum x \in X. 2^{\wedge} x :: \text{nat}) = (\sum x \in Y. 2^{\wedge} x) \rrbracket \implies X = Y$   
(is -  $\implies$  -  $\implies$  ?f X = ?f Y  $\implies$  -)

**proof** (induct X arbitrary: Y rule: finite-linorder-max-induct)

case (insert x X)

from insert(2) have ?f X  $\leq$  ?f {0 ..< x} by (intro sum-mono2) auto

also have ...  $<$   $2^{\wedge} x$  by (induct x) simp-all

finally have ?f X  $<$   $2^{\wedge} x$  .

moreover from insert(1,2) have \*: ?f X +  $2^{\wedge} x =$  ?f Y

using trans[OF sym[OF insert(5)] sum.insert] by auto

ultimately have ?f Y  $<$   $2^{\wedge} \text{Suc } x$  by simp

have  $\forall y \in Y. y \leq x$

**proof** (rule ccontr)

assume  $\neg (\forall y \in Y. y \leq x)$

then obtain y where  $y \in Y$  Suc x  $\leq$  y by auto

from this(2) have  $2^{\wedge} \text{Suc } x \leq (2^{\wedge} y :: \text{nat})$  by (intro power-increasing) auto

also from  $\langle y \in Y \rangle$  insert(4) have ...  $\leq$  ?f Y by (metis order.refl sum.remove trans-le-add1)

finally show False using  $\langle ?f Y < 2^{\wedge} \text{Suc } x \rangle$  by simp

qed

{ assume  $x \notin Y$

with  $\langle \forall y \in Y. y \leq x \rangle$  have ?f Y  $\leq$  ?f {0 ..< x} by (intro sum-mono2) (auto simp: le-less)

also have ...  $<$   $2^{\wedge} x$  by (induct x) simp-all

finally have ?f Y  $<$   $2^{\wedge} x$  .

with \* have False by auto

}

then have  $x \in Y$  by blast

from insert(4) have ?f (Y - {x}) +  $2^{\wedge} x =$  ?f (insert x (Y - {x})) by (subst sum.insert) auto

also have ... = ?f X +  $2^{\wedge} x$  unfolding \* using  $\langle x \in Y \rangle$  by (simp add: insert-absorb)

finally have ?f X = ?f (Y - {x}) by simp

with insert(3,4) have X = Y - {x} by simp

with  $\langle x \in Y \rangle$  show ?case by auto

qed simp

**lemma** *finite-pow2-eq*:

fixes n :: nat

shows finite {i.  $2^{\wedge} i = n$ }

**proof** -

have (( $\cap$ ) 2) ' {i.  $2^{\wedge} i = n$  }  $\subseteq$  {n} by auto

then have finite ((( $\cap$ ) (2 :: nat)) ' {i.  $2^{\wedge} i = n$  }) by (rule finite-subset) blast

then show finite {i.  $2^{\wedge} i = n$  } by (rule finite-imageD) (auto simp: inj-on-def)

qed

**lemma** *finite-pow2-le[simp]*:

**fixes**  $n :: nat$

**shows** *finite*  $\{i. 2 \wedge i \leq n\}$

**by** (*induct n*) (*auto simp: le-Suc-eq finite-pow2-eq*)

**lemma** *le-pow2[simp]*:  $x \leq y \implies x \leq 2 \wedge y$

**by** (*induct x arbitrary: y*) (*force simp add: Suc-le-eq order.strict-iff-order*)+

**lemma** *ld-bounded*:  $Max \{i. 2 \wedge i \leq Suc\ n\} \leq Suc\ n$  (**is**  $?m \leq Suc\ n$ )

**proof** –

**have**  $?m \leq 2 \wedge ?m$  **by** (*rule le-pow2*) *simp*

**moreover**

**have**  $?m \in \{i. 2 \wedge i \leq Suc\ n\}$  **by** (*rule Max-in*) (*auto intro: exI[of - 0]*)

**then have**  $2 \wedge ?m \leq Suc\ n$  **by** *simp*

**ultimately show** *?thesis* **by** *linarith*

qed

**primrec** *satisfies0* **where**

*satisfies0*  $\mathfrak{A} (Fo\ m) = (m \mathfrak{A} FO \neq \{\})$

| *satisfies0*  $\mathfrak{A} (Eq\ Const\ i\ m\ n) =$

(*let*  $P = m \mathfrak{A} FO$  *in if*  $P = \{\}$

*then* (*case*  $i$  *of*  $Some\ i \Rightarrow Length\ \mathfrak{A} = i \mid - \Rightarrow False$ )

*else*  $fMin\ P = n$ )

| *satisfies0*  $\mathfrak{A} (Less\ b\ m1\ m2) =$

(*let*  $P1 = m1 \mathfrak{A} FO; P2 = m2 \mathfrak{A} FO$  *in if*  $P1 = \{\} \vee P2 = \{\}$

*then* (*case*  $b$  *of*  $None \Rightarrow False \mid Some\ True \Rightarrow P2 = \{\} \mid Some\ False \Rightarrow P1 \neq$

$\{\}$ )

*else*  $fMin\ P1 < fMin\ P2$ )

| *satisfies0*  $\mathfrak{A} (Plus\ FO\ i\ m1\ m2\ n) =$

(*let*  $P1 = m1 \mathfrak{A} FO; P2 = m2 \mathfrak{A} FO$  *in if*  $P1 = \{\} \vee P2 = \{\}$

*then* (*case*  $i$  *of*  $Some\ 0 \Rightarrow P1 = P2 \mid Some\ i \Rightarrow P2 \neq \{\} \wedge fMin\ P2 + i =$

$Length\ \mathfrak{A} \mid - \Rightarrow False$ )

*else*  $fMin\ P1 = fMin\ P2 + n$ )

| *satisfies0*  $\mathfrak{A} (Eq\ FO\ b\ m1\ m2) =$

(*let*  $P1 = m1 \mathfrak{A} FO; P2 = m2 \mathfrak{A} FO$  *in if*  $P1 = \{\} \vee P2 = \{\}$

*then*  $b \wedge P1 = P2$

*else*  $fMin\ P1 = fMin\ P2$ )

| *satisfies0*  $\mathfrak{A} (Eq\ SO\ M1\ M2) = (M1 \mathfrak{A} SO = M2 \mathfrak{A} SO)$

| *satisfies0*  $\mathfrak{A} (Suc\ SO\ br\ bl\ M1\ M2) =$

((*if*  $br$  *then* *finsert* ( $Length\ \mathfrak{A}$ ) *else* *id*) ( $M1 \mathfrak{A} SO$ ) =

(*if*  $bl$  *then* *finsert*  $0$  *else* *id*) ( $Suc \mid \uparrow M2 \mathfrak{A} SO$ ))

| *satisfies0*  $\mathfrak{A} (Empty\ M) = (M \mathfrak{A} SO = \{\})$

| *satisfies0*  $\mathfrak{A} (Singleton\ M) = (\exists x. M \mathfrak{A} SO = \{x\})$

| *satisfies0*  $\mathfrak{A} (Subset\ M1\ M2) = (M1 \mathfrak{A} SO \subseteq M2 \mathfrak{A} SO)$

| *satisfies0*  $\mathfrak{A} (In\ b\ m\ M) =$

(*let*  $P = m \mathfrak{A} FO$  *in if*  $P = \{\}$  *then*  $b$  *else*  $fMin\ P \in M \mathfrak{A} SO$ )

$| \text{satisfies0 } \mathfrak{A} (\text{Eq-Max } b \ m \ M) =$   
 $\quad (\text{let } P1 = m^{\mathfrak{A}}\text{FO}; P2 = M^{\mathfrak{A}}\text{SO in if } b \text{ then } P1 = \{\|\})$   
 $\quad \text{else if } P1 = \{\|\} \vee P2 = \{\|\} \text{ then False else } f\text{Min } P1 = f\text{Max } P2)$   
 $| \text{satisfies0 } \mathfrak{A} (\text{Eq-Min } b \ m \ M) =$   
 $\quad (\text{let } P1 = m^{\mathfrak{A}}\text{FO}; P2 = M^{\mathfrak{A}}\text{SO in if } P1 = \{\|\} \vee P2 = \{\|\} \text{ then } b \wedge P1 = P2$   
 $\text{else } f\text{Min } P1 = f\text{Min } P2)$   
 $| \text{satisfies0 } \mathfrak{A} (\text{Eq-Union } M1 \ M2 \ M3) = (M1^{\mathfrak{A}}\text{SO} = M2^{\mathfrak{A}}\text{SO} \mid \cup \mid M3^{\mathfrak{A}}\text{SO})$   
 $| \text{satisfies0 } \mathfrak{A} (\text{Eq-Inter } M1 \ M2 \ M3) = (M1^{\mathfrak{A}}\text{SO} = M2^{\mathfrak{A}}\text{SO} \mid \cap \mid M3^{\mathfrak{A}}\text{SO})$   
 $| \text{satisfies0 } \mathfrak{A} (\text{Eq-Diff } M1 \ M2 \ M3) = (M1^{\mathfrak{A}}\text{SO} = M2^{\mathfrak{A}}\text{SO} \mid - \mid M3^{\mathfrak{A}}\text{SO})$   
 $| \text{satisfies0 } \mathfrak{A} (\text{Disjoint } M1 \ M2) = (M1^{\mathfrak{A}}\text{SO} \mid \cap \mid M2^{\mathfrak{A}}\text{SO} = \{\|\})$   
 $| \text{satisfies0 } \mathfrak{A} (\text{Eq-Presb } i \ M \ n) = (((\sum x \in \text{fset } (M^{\mathfrak{A}}\text{SO}). 2 \wedge x) = n) \wedge$   
 $\quad (\text{case } i \text{ of None } \Rightarrow \text{True} \mid \text{Some } l \Rightarrow \text{Length } \mathfrak{A} = l))$

**fun lderiv0 where**

$\text{lderv0 } (bs1, bs2) (\text{Fo } m) = (\text{if } bs1 \ ! \ m \text{ then } \text{FBool True} \text{ else } \text{FBase } (\text{Fo } m))$   
 $| \text{lderv0 } (bs1, bs2) (\text{Eq-Const None } m \ n) = (\text{if } n = 0 \wedge bs1 \ ! \ m \text{ then } \text{FBool True}$   
 $\quad \text{else if } n = 0 \vee bs1 \ ! \ m \text{ then } \text{FBool False} \text{ else } \text{FBase } (\text{Eq-Const None } m \ (n -$   
 $\quad 1)))$   
 $| \text{lderv0 } (bs1, bs2) (\text{Less None } m1 \ m2) = (\text{case } (bs1 \ ! \ m1, bs1 \ ! \ m2) \text{ of}$   
 $\quad (\text{False, False}) \Rightarrow \text{FBase } (\text{Less None } m1 \ m2)$   
 $\quad | (\text{True, False}) \Rightarrow \text{FBase } (\text{Fo } m2)$   
 $\quad | - \Rightarrow \text{FBool False})$   
 $\text{lderv0 } (bs1, bs2) (\text{Eq-FO False } m1 \ m2) = (\text{case } (bs1 \ ! \ m1, bs1 \ ! \ m2) \text{ of}$   
 $\quad (\text{False, False}) \Rightarrow \text{FBase } (\text{Eq-FO False } m1 \ m2)$   
 $\quad | (\text{True, True}) \Rightarrow \text{FBool True}$   
 $\quad | - \Rightarrow \text{FBool False})$   
 $| \text{lderv0 } (bs1, bs2) (\text{Plus-FO None } m1 \ m2 \ n) = (\text{if } n = 0$   
 $\quad \text{then}$   
 $\quad \quad (\text{case } (bs1 \ ! \ m1, bs1 \ ! \ m2) \text{ of}$   
 $\quad \quad \quad (\text{False, False}) \Rightarrow \text{FBase } (\text{Plus-FO None } m1 \ m2 \ n)$   
 $\quad \quad \quad | (\text{True, True}) \Rightarrow \text{FBool True}$   
 $\quad \quad \quad | - \Rightarrow \text{FBool False})$   
 $\quad \text{else}$   
 $\quad \quad (\text{case } (bs1 \ ! \ m1, bs1 \ ! \ m2) \text{ of}$   
 $\quad \quad \quad (\text{False, False}) \Rightarrow \text{FBase } (\text{Plus-FO None } m1 \ m2 \ n)$   
 $\quad \quad \quad | (\text{False, True}) \Rightarrow \text{FBase } (\text{Eq-Const None } m1 \ (n - 1))$   
 $\quad \quad \quad | - \Rightarrow \text{FBool False})$   
 $| \text{lderv0 } (bs1, bs2) (\text{Eq-SO } M1 \ M2) =$   
 $\quad (\text{if } bs2 \ ! \ M1 = bs2 \ ! \ M2 \text{ then } \text{FBase } (\text{Eq-SO } M1 \ M2) \text{ else } \text{FBool False})$   
 $| \text{lderv0 } (bs1, bs2) (\text{Suc-SO False } bl \ M1 \ M2) = (\text{if } bl = bs2 \ ! \ M1$   
 $\quad \text{then } \text{FBase } (\text{Suc-SO False } (bs2 \ ! \ M2) \ M1 \ M2) \text{ else } \text{FBool False})$   
 $| \text{lderv0 } (bs1, bs2) (\text{Empty } M) = (\text{case } bs2 \ ! \ M \text{ of}$   
 $\quad \text{True} \Rightarrow \text{FBool False}$   
 $\quad | \text{False} \Rightarrow \text{FBase } (\text{Empty } M))$   
 $| \text{lderv0 } (bs1, bs2) (\text{Singleton } M) = (\text{case } bs2 \ ! \ M \text{ of}$   
 $\quad \text{True} \Rightarrow \text{FBase } (\text{Empty } M)$   
 $\quad | \text{False} \Rightarrow \text{FBase } (\text{Singleton } M))$   
 $| \text{lderv0 } (bs1, bs2) (\text{Subset } M1 \ M2) = (\text{case } (bs2 \ ! \ M1, bs2 \ ! \ M2) \text{ of}$

```

    (True, False) ⇒ FBool False
  | - ⇒ FBase (Subset M1 M2))
| lderiv0 (bs1, bs2) (In False m M) = (case (bs1 ! m, bs2 ! M) of
  (False, -) ⇒ FBase (In False m M)
| (True, True) ⇒ FBool True
| - ⇒ FBool False)
| lderiv0 (bs1, bs2) (Eq-Max False m M) = (case (bs1 ! m, bs2 ! M) of
  (False, -) ⇒ FBase (Eq-Max False m M)
| (True, True) ⇒ FBase (Empty M)
| - ⇒ FBool False)
| lderiv0 (bs1, bs2) (Eq-Min False m M) = (case (bs1 ! m, bs2 ! M) of
  (False, False) ⇒ FBase (Eq-Min False m M)
| (True, True) ⇒ FBool True
| - ⇒ FBool False)
| lderiv0 (bs1, bs2) (Eq-Union M1 M2 M3) = (if bs2 ! M1 = (bs2 ! M2 ∨ bs2 !
M3)
  then FBase (Eq-Union M1 M2 M3) else FBool False)
| lderiv0 (bs1, bs2) (Eq-Inter M1 M2 M3) = (if bs2 ! M1 = (bs2 ! M2 ∧ bs2 !
M3)
  then FBase (Eq-Inter M1 M2 M3) else FBool False)
| lderiv0 (bs1, bs2) (Eq-Diff M1 M2 M3) = (if bs2 ! M1 = (bs2 ! M2 ∧ ¬ bs2 !
M3)
  then FBase (Eq-Diff M1 M2 M3) else FBool False)
| lderiv0 (bs1, bs2) (Disjoint M1 M2) =
  (if bs2 ! M1 ∧ bs2 ! M2 then FBool False else FBase (Disjoint M1 M2))
| lderiv0 (bs1, bs2) (Eq-Presb None M n) = (if bs2 ! M = (n mod 2 = 0)
  then FBool False else FBase (Eq-Presb None M (n div 2)))
| lderiv0 - - = undefined

```

**fun** *ld* **where**

```

  ld 0 = 0
| ld (Suc 0) = 0
| ld n = Suc (ld (n div 2))

```

**lemma** *ld-alt[simp]*:  $n > 0 \implies ld\ n = \text{Max}\ \{i.\ 2^i \leq n\}$

**proof** (*safe intro!*: *Max-eqI[symmetric]*)

**assume**  $n > 0$  **then show**  $2^{\text{ld}\ n} \leq n$  **by** (*induct n rule: ld.induct*) *auto*

**next**

**fix** *y*

**assume**  $2^y \leq n$

**then show**  $y \leq \text{ld}\ n$

**proof** (*induct n arbitrary: y rule: ld.induct*)

**case** ( $\exists z$ )

**then have**  $y - 1 \leq \text{ld}\ (\text{Suc}\ (\text{Suc}\ z)\ \text{div}\ 2)$

**by** (*cases y*) *simp-all*

**then show** *?case* **by** *simp*

**qed** (*auto simp: le-eq-less-or-eq*)

**qed** *simp*

```

fun rderiv0 where
  rderiv0 (bs1, bs2) (Fo m) = (if bs1 ! m then FBool True else FBase (Fo m))
| rderiv0 (bs1, bs2) (Eq-Const i m n) = (case bs1 ! m of
  False ⇒ FBase (Eq-Const (case i of Some (Suc i) ⇒ Some i | - ⇒ None) m n)
  | True ⇒ FBase (Eq-Const (Some n) m n))
| rderiv0 (bs1, bs2) (Less b m1 m2) = (case bs1 ! m2 of
  False ⇒ (case b of
    Some False ⇒ (case bs1 ! m1 of
      True ⇒ FBase (Less (Some True) m1 m2)
      | False ⇒ FBase (Less (Some False) m1 m2))
    | - ⇒ FBase (Less b m1 m2))
  | True ⇒ FBase (Less (Some False) m1 m2))
| rderiv0 (bs1, bs2) (Plus-FO i m1 m2 n) = (if n = 0
then
  (case (bs1 ! m1, bs1 ! m2) of
    (False, False) ⇒ FBase (Plus-FO i m1 m2 n)
    | (True, True) ⇒ FBase (Plus-FO (Some 0) m1 m2 n)
    | - ⇒ FBase (Plus-FO None m1 m2 n))
else
  (case bs1 ! m1 of
    True ⇒ FBase (Plus-FO (Some n) m1 m2 n)
    | False ⇒ (case bs1 ! m2 of
      False ⇒ (case i of
        Some (Suc (Suc i)) ⇒ FBase (Plus-FO (Some (Suc i)) m1 m2 n)
        | Some (Suc 0) ⇒ FBase (Plus-FO None m1 m2 n)
        | - ⇒ FBase (Plus-FO i m1 m2 n))
      | True ⇒ (case i of
        Some (Suc i) ⇒ FBase (Plus-FO (Some i) m1 m2 n)
        | - ⇒ FBase (Plus-FO None m1 m2 n))))))
| rderiv0 (bs1, bs2) (Eq-FO b m1 m2) = (case (bs1 ! m1, bs1 ! m2) of
  (False, False) ⇒ FBase (Eq-FO b m1 m2)
  | (True, True) ⇒ FBase (Eq-FO True m1 m2)
  | - ⇒ FBase (Eq-FO False m1 m2))
| rderiv0 (bs1, bs2) (Eq-SO M1 M2) =
  (if bs2 ! M1 = bs2 ! M2 then FBase (Eq-SO M1 M2) else FBool False)
| rderiv0 (bs1, bs2) (Suc-SO br bl M1 M2) = (if br = bs2 ! M2
  then FBase (Suc-SO (bs2 ! M1) bl M1 M2) else FBool False)
| rderiv0 (bs1, bs2) (Empty M) = (case bs2 ! M of
  True ⇒ FBool False
  | False ⇒ FBase (Empty M))
| rderiv0 (bs1, bs2) (Singleton M) = (case bs2 ! M of
  True ⇒ FBase (Empty M)
  | False ⇒ FBase (Singleton M))
| rderiv0 (bs1, bs2) (Subset M1 M2) = (case (bs2 ! M1, bs2 ! M2) of
  (True, False) ⇒ FBool False
  | - ⇒ FBase (Subset M1 M2))
| rderiv0 (bs1, bs2) (In b m M) = (case (bs1 ! m, bs2 ! M) of
  (True, True) ⇒ FBase (In True m M)
  | (True, False) ⇒ FBase (In False m M)

```

| -  $\Rightarrow$  *FBase* (*In* *b m M*)  
 | *rderiv0* (*bs1*, *bs2*) (*Eq-Max* *b m M*) = (*case* (*bs1 ! m*, *bs2 ! M*) of  
   (*True*, *True*)  $\Rightarrow$  if *b* then *FBool* *False* else *FBase* (*Eq-Max* *True m M*)  
   (*True*, *False*)  $\Rightarrow$  if *b* then *FBool* *False* else *FBase* (*Eq-Max* *False m M*)  
   (*False*, *True*)  $\Rightarrow$  if *b* then *FBase* (*Eq-Max* *True m M*) else *FBool* *False*  
   (*False*, *False*)  $\Rightarrow$  *FBase* (*Eq-Max* *b m M*)  
 | *rderiv0* (*bs1*, *bs2*) (*Eq-Min* *b m M*) = (*case* (*bs1 ! m*, *bs2 ! M*) of  
   (*True*, *True*)  $\Rightarrow$  *FBase* (*Eq-Min* *True m M*)  
   (*False*, *False*)  $\Rightarrow$  *FBase* (*Eq-Min* *b m M*)  
   -  $\Rightarrow$  *FBase* (*Eq-Min* *False m M*)  
 | *rderiv0* (*bs1*, *bs2*) (*Eq-Union* *M1 M2 M3*) = (*if* *bs2 ! M1* = (*bs2 ! M2*  $\vee$  *bs2 ! M3*)  
   then *FBase* (*Eq-Union* *M1 M2 M3*) else *FBool* *False*)  
 | *rderiv0* (*bs1*, *bs2*) (*Eq-Inter* *M1 M2 M3*) = (*if* *bs2 ! M1* = (*bs2 ! M2*  $\wedge$  *bs2 ! M3*)  
   then *FBase* (*Eq-Inter* *M1 M2 M3*) else *FBool* *False*)  
 | *rderiv0* (*bs1*, *bs2*) (*Eq-Diff* *M1 M2 M3*) = (*if* *bs2 ! M1* = (*bs2 ! M2*  $\wedge$   $\neg$  *bs2 ! M3*)  
   then *FBase* (*Eq-Diff* *M1 M2 M3*) else *FBool* *False*)  
 | *rderiv0* (*bs1*, *bs2*) (*Disjoint* *M1 M2*) =  
   (*if* *bs2 ! M1*  $\wedge$  *bs2 ! M2* then *FBool* *False* else *FBase* (*Disjoint* *M1 M2*))  
 | *rderiv0* (*bs1*, *bs2*) (*Eq-Presb* *l M n*) = (*case* *l* of  
   None  $\Rightarrow$  if *bs2 ! M* then  
     if *n* = 0 then *FBool* *False*  
     else let *l* = *ld n* in *FBase* (*Eq-Presb* (*Some l*) *M* (*n* - 2  $\wedge$  *l*))  
     else *FBase* (*Eq-Presb* *l M n*)  
   Some 0  $\Rightarrow$  *FBool* *False*  
   Some (*Suc l*)  $\Rightarrow$  if *bs2 ! M*  $\wedge$  *n*  $\geq$  2  $\wedge$  *l* then *FBase* (*Eq-Presb* (*Some l*) *M* (*n* - 2  $\wedge$  *l*))  
   else if  $\neg$  *bs2 ! M*  $\wedge$  *n* < 2  $\wedge$  *l* then *FBase* (*Eq-Presb* (*Some l*) *M n*)  
   else *FBool* *False*)

**primrec** *nullable0* where

*nullable0* (*Fo m*) = *False*  
 | *nullable0* (*Eq-Const* *i m n*) = (*i* = *Some 0*)  
 | *nullable0* (*Less* *b m1 m2*) = (*case* *b* of None  $\Rightarrow$  *False* | *Some b*  $\Rightarrow$  *b*)  
 | *nullable0* (*Plus-FO* *i m1 m2 n*) = (*i* = *Some 0*)  
 | *nullable0* (*Eq-FO* *b m1 m2*) = *b*  
 | *nullable0* (*Eq-SO* *M1 M2*) = *True*  
 | *nullable0* (*Suc-SO* *br bl M1 M2*) = (*bl* = *br*)  
 | *nullable0* (*Empty* *M*) = *True*  
 | *nullable0* (*Singleton* *M*) = *False*  
 | *nullable0* (*Subset* *M1 M2*) = *True*  
 | *nullable0* (*In* *b m M*) = *b*  
 | *nullable0* (*Eq-Max* *b m M*) = *b*  
 | *nullable0* (*Eq-Min* *b m M*) = *b*  
 | *nullable0* (*Eq-Union* *M1 M2 M3*) = *True*  
 | *nullable0* (*Eq-Inter* *M1 M2 M3*) = *True*  
 | *nullable0* (*Eq-Diff* *M1 M2 M3*) = *True*

| *nullable0* (*Disjoint M1 M2*) = *True*  
| *nullable0* (*Eq-Presb l M n*) = (*n = 0*  $\wedge$  (*l = Some 0*  $\vee$  *l = None*))

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

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

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

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

*extend size-atom zero  $\sigma$  eval downshift upshift finsert cut len restrict Restrict lformula0 FV0 find0 wf0 decr0 satisfies0 nullable0 lderiv0 rderiv0 undefined*  
**defines** *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 find0 decr0* :: *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* :: -  $\Rightarrow$  *formula*  
**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-lformula* = *Formula-Operations.lformula lformula0* :: *formula*  $\Rightarrow$  -  
**and** *check-eqv* =  $\lambda idx$ . *DAs.check-eqv*  
( $\sigma$  *idx*) ( $\lambda \varphi$ . *norm (RESTRICT*  $\varphi$ ) :: (*ws1s*, *order*) *aformula*)  
( $\lambda a$   $\varphi$ . *norm (deriv (lderv0* :: -  $\Rightarrow$  -  $\Rightarrow$  *formula*) (*a* :: *atom*)  $\varphi$ ))  
(*final idx*) ( $\lambda \varphi$  :: *formula*. *ws1s-wf idx*  $\varphi$   $\wedge$  *ws1s-lformula*  $\varphi$ )  
( $\sigma$  *idx*) ( $\lambda \varphi$ . *norm (RESTRICT*  $\varphi$ ) :: (*ws1s*, *order*) *aformula*)  
( $\lambda a$   $\varphi$ . *norm (deriv (lderv0* :: -  $\Rightarrow$  -  $\Rightarrow$  *formula*) (*a* :: *atom*)  $\varphi$ ))  
(*final idx*) ( $\lambda \varphi$  :: *formula*. *ws1s-wf idx*  $\varphi$   $\wedge$  *ws1s-lformula*  $\varphi$ ) (=)  
**and** *bounded-check-eqv* =  $\lambda idx$ . *DAs.check-eqv*  
( $\sigma$  *idx*) ( $\lambda \varphi$ . *norm (RESTRICT*  $\varphi$ ) :: (*ws1s*, *order*) *aformula*)  
( $\lambda a$   $\varphi$ . *norm (deriv (lderv0* :: -  $\Rightarrow$  -  $\Rightarrow$  *formula*) (*a* :: *atom*)  $\varphi$ ))  
*nullable* ( $\lambda \varphi$  :: *formula*. *ws1s-wf idx*  $\varphi$   $\wedge$  *ws1s-lformula*  $\varphi$ )  
( $\sigma$  *idx*) ( $\lambda \varphi$ . *norm (RESTRICT*  $\varphi$ ) :: (*ws1s*, *order*) *aformula*)  
( $\lambda a$   $\varphi$ . *norm (deriv (lderv0* :: -  $\Rightarrow$  -  $\Rightarrow$  *formula*) (*a* :: *atom*)  $\varphi$ ))  
*nullable* ( $\lambda \varphi$  :: *formula*. *ws1s-wf idx*  $\varphi$   $\wedge$  *ws1s-lformula*  $\varphi$ ) (=)  
**and** *automaton* = *DA.automaton*

```

    ( $\lambda a \varphi. \text{norm} (\text{deriv lderiv0} (a :: \text{atom}) \varphi :: \text{formula})$ )
proof
  fix  $k \text{ idx}$  and  $a :: \text{ws1s}$  and  $l$  assume  $\text{wf0} (SUC\ k\ \text{idx})\ a\ \text{LESS}\ k\ l\ (SUC\ k\ \text{idx})$ 
 $\neg \text{find0}\ k\ l\ a$ 
  then show  $\text{wf0}\ \text{idx}\ (\text{decr0}\ k\ l\ a)$ 
    by ( $\text{induct}\ a$ ) ( $\text{unfold}\ \text{wf0.simps}\ \text{atomic.map}\ \text{find0.simps},$ 
      ( $\text{transfer}, \text{force}\ \text{simp:}\ \text{dec-def}\ \text{split!}: \text{if-splits}\ \text{order.splits}$ ) $+$ )
next
  fix  $k$  and  $a :: \text{ws1s}$  and  $l$  assume  $l\text{formula0}\ a$ 
  then show  $l\text{formula0}\ (\text{decr0}\ k\ l\ a)$  by ( $\text{induct}\ a$ )  $\text{auto}$ 
next
  fix  $i\ k$  and  $a :: \text{ws1s}$  and  $\mathfrak{A} :: \text{interp}$  and  $P$  assume  $*$ :  $\neg \text{find0}\ k\ i\ a\ \text{LESS}\ k\ i$ 
 $(SUC\ k\ (\#\vee\ \mathfrak{A}))$ 
    and  $\text{disj}: l\text{formula0}\ a \vee \text{len}\ P \leq \text{Length}\ \mathfrak{A}$ 
  from  $\text{disj}$  show  $\text{satisfies0}\ (\text{Extend}\ k\ i\ \mathfrak{A}\ P)\ a = \text{satisfies0}\ \mathfrak{A}\ (\text{decr0}\ k\ i\ a)$ 
proof
  assume  $l\text{formula0}\ a$ 
  then show  $?thesis$  using  $*$ 
    by ( $\text{induct}\ a\ \text{rule}: l\text{formula0.induct}$ )
      ( $\text{auto}\ \text{simp:}\ \text{dec-def}\ \text{split}: \text{if-splits}\ \text{order.split}\ \text{option.splits}\ \text{bool.splits}$ ) —  $\text{slow}$ 
next
  note  $\text{dec-def}[\text{simp}]$ 
  assume  $\text{len}\ P \leq \text{Length}\ \mathfrak{A}$ 
  with  $*$  show  $?thesis$ 
proof ( $\text{induct}\ a$ )
  case  $F_0$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 
next
  case  $\text{Eq-Const}$  then show  $?case$ 
    by ( $\text{cases}\ k$ ) ( $\text{auto}\ \text{simp:}\ \text{Let-def}\ \text{len-def}\ \text{split}: \text{if-splits}\ \text{option.splits}$ )
next
  case  $\text{Less}$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 
next
  case  $\text{Plus-FO}$  then show  $?case$ 
    by ( $\text{cases}\ k$ ) ( $\text{auto}\ \text{simp:}\ \text{max-def}\ \text{len-def}\ \text{Let-def}\ \text{split}: \text{option.splits}\ \text{nat.splits}$ )
next
  case  $\text{Eq-FO}$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 
next
  case  $\text{Eq-SO}$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 
next
  case ( $\text{Suc-SO}\ \text{br}\ \text{bl}\ M1\ M2$ ) then show  $?case$ 
    by ( $\text{cases}\ k$ ) ( $\text{auto}\ \text{simp:}\ \text{max-def}\ \text{len-def}$ )
next
  case  $\text{Empty}$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 
next
  case  $\text{Singleton}$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 
next
  case  $\text{Subset}$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 
next
  case  $\text{In}$  then show  $?case$  by ( $\text{cases}\ k$ )  $\text{auto}$ 

```



```

    qed (auto simp: len-def max-def split!: option.splits order.splits)
  qed
next
  fix idx and a :: ws1s and x assume lformula0 a wf0 idx a
  then show Formula-Operations.wf SUC wf0 idx (lderiv0 x a)
  by (induct a rule: lderiv0.induct)
  (auto simp: Formula-Operations.wf.simps Let-def split: bool.splits order.splits)
next
  fix a :: ws1s and x assume lformula0 a
  then show Formula-Operations.lformula lformula0 (lderiv0 x a)
  by (induct a rule: lderiv0.induct)
  (auto simp: Formula-Operations.lformula.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 a rule: lderiv0.induct)
  (auto simp: Formula-Operations.wf.simps Let-def sorted-append
    split: bool.splits order.splits nat.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 3 dest: MSB-greater split: prod.splits if-splits option.splits
    bool.splits nat.splits) [])+ — slow
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 lformula0 a 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)
  case 18
  then show ?case
    apply (auto simp: sum-pow2-image-Suc sum-pow2-insert0 image-iff split:
prod.splits)
    apply presburger+
    done
  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 lformula0 a 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)
  case 18

```

```

then show ?case
  apply (auto simp: sum-pow2-image-Suc sum-pow2-insert0 image-iff split:
prod.splits)
  apply presburger+
  done
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  $\#_V \mathfrak{A} = \text{size-atom } x$ 
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 Eq-Const then show ?case by (auto split: prod.splits option.splits
nat.splits)
next
case Less then show ?case
  by (auto split: prod.splits option.splits bool.splits) (metis fMin-less-Length
less-not-sym)+
next
case (Plus-FO i m1 m2 n) then show ?case
  by (auto simp: min commute dest: fMin-less-Length
split: prod.splits option.splits nat.splits bool.splits)
next
case Eq-FO then show ?case
  by (auto split: prod.splits option.splits bool.splits) (metis fMin-less-Length
less-not-sym)+
next
case Eq-SO then show ?case
  by (auto split: prod.splits option.splits bool.splits)
(metis assigns-less-Length finsertI1 less-not-refl)+
next
case Suc-SO then show ?case
  apply (auto 2 1 dest: assigns-less-Length split: prod.splits)
  apply (metis fimage.rep-eq finsert-iff less-not-refl)
  apply (metis fimage.rep-eq finsert-iff less-not-refl)
  apply (metis fimage.rep-eq finsert-iff n-not-Suc-n)
  apply (metis Suc-lessD assigns-less-Length fimage.rep-eq finsert-iff
not-less-eq)
  apply (metis Suc-less-eq assigns-less-Length fimageI finsert-iff
less-not-refl)
  apply (metis fimage.rep-eq finsert-fminus-single finsert-iff n-not-Suc-n)
  apply (metis assigns-less-Length dual-order.strict-trans fimage.rep-eq
finsert-fminus-single finsert-iff lessI less-not-refl)
  apply (metis Suc-n-not-le-n assigns-less-Length finsert-iff less-or-eq-imp-le)
  apply (metis Suc-n-not-le-n assigns-less-Length finsertE finsertI1
less-or-eq-imp-le)
  apply (metis assigns-less-Length fimage.rep-eq finsert-iff lessI or-

```

```

der-less-imp-not-less)
  apply (metis Length-notin-assigns finsert-fimage finsert-iff nat.inject)
  apply (metis assigns-less-Length fimage.rep-eq finsert-fminus-single finsert-iff not-add-less2 plus-1-eq-Suc)
  apply (metis assigns-less-Length finsertI1 finsert-commute not-add-less2 plus-1-eq-Suc)
  apply (metis assigns-less-Length finsertI1 not-add-less2 plus-1-eq-Suc)
  apply (metis Length-notin-assigns Suc-in-fimage-Suc finsert-iff)
  by (metis Length-notin-assigns Suc-in-fimage-Suc finsertI1)
next
case In then show ?case by (auto split: prod.splits) (metis fMin-less-Length less-not-sym)+
next
case (Eq-Max b m M) then show ?case
  by (auto split: prod.splits bool.splits)
  (metis fMax-less-Length less-not-sym, (metis fMin-less-Length less-not-sym)+)
next
case Eq-Min then show ?case
  by (auto split: prod.splits bool.splits) (metis fMin-less-Length less-not-sym)+
next
case Eq-Union then show ?case
  by (auto 0 0 simp add: fset-eq-iff split: prod.splits) (metis assigns-less-Length less-not-refl)+
next
case Eq-Inter then show ?case
  by (auto 0 0 simp add: fset-eq-iff split: prod.splits) (metis assigns-less-Length less-not-refl)+
next
case Eq-Diff then show ?case
  by (auto 0 1 simp add: fset-eq-iff split: prod.splits) (metis assigns-less-Length less-not-refl)+
next
let ?f = sum (( $\wedge$ ) (2 :: nat))
case (Eq-Presb l M n)
moreover
let ?M = fset (MℳSO) and ?L = Length ℳ
have ?f (insert ?L ?M) = 2 ^ ?L + ?f ?M
  by (subst sum.insert) auto
moreover have n > 0  $\implies$  2 ^ Max {i. 2 ^ i ≤ n} ≤ n
  using Max-in[of {i. 2 ^ i ≤ n}, simplified, OF exI[of - 0]] by auto
moreover
{ have ?f ?M ≤ ?f {0 ..< ?L} by (rule sum-mono2) auto
  also have ... = 2 ^ ?L - 1 by (rule sum-pow2-upto)
  also have ... < 2 ^ ?L by simp
  finally have ?f ?M < 2 ^ ?L .
}
moreover have Max {i. 2 ^ i ≤ 2 ^ ?L + ?f ?M} = ?L
proof (intro Max-eqI, safe)
  fix y assume 2 ^ y ≤ 2 ^ ?L + ?f ?M

```

```

    { assume ?L < y
      then have (2 :: nat) ^ ?L + 2 ^ ?L ≤ 2 ^ y
        by (cases y) (auto simp: less-Suc-eq-le numeral-eq-Suc add-le-mono)
      also note ⟨2 ^ y ≤ 2 ^ ?L + ?f ?M⟩
      finally have 2 ^ ?L ≤ ?f ?M by simp
      with ⟨?f ?M < 2 ^ ?L⟩ have False by auto
    } then show y ≤ ?L by (intro leI) blast
  qed auto
  ultimately show ?case by (auto split: prod.splits option.splits nat.splits)
  qed (auto split: prod.splits)
next
  fix a :: ws1s and  $\mathfrak{A} \mathfrak{B} :: \text{interp}$ 
  assume wf0 (#V  $\mathfrak{B}$ ) a #V  $\mathfrak{A} = \#_V \mathfrak{B} (\bigwedge m k. \text{LESS } k m (\#_V \mathfrak{B}) \implies m^{\mathfrak{A}} k = m^{\mathfrak{B}} k)$  lformula0 a
  then show satisfies0  $\mathfrak{A} a \longleftrightarrow \text{satisfies0 } \mathfrak{B} a$  by (induct a) auto
next
  fix a :: ws1s
  assume lformula0 a
  moreover
  define d where d = Formula-Operations.deriv extend lderiv0
  define  $\Phi :: - \Rightarrow (\text{ws1s}, \text{order}) \text{ aformula set}$ 
  where  $\Phi a =$ 
    (case a of
      | Fo m  $\Rightarrow \{ \text{FBase } (\text{Fo } m), \text{FBool True} \}$ 
      | Eq-Const None m n  $\Rightarrow \{ \text{FBase } (\text{Eq-Const None } m i) \mid i . i \leq n \} \cup \{ \text{FBool True, FBool False} \}$ 
      | Less None m1 m2  $\Rightarrow \{ \text{FBase } (\text{Less None } m1 m2), \text{FBase } (\text{Fo } m2), \text{FBool True, FBool False} \}$ 
      | Plus-FO None m1 m2 n  $\Rightarrow \{ \text{FBase } (\text{Eq-Const None } m1 i) \mid i . i \leq n \} \cup \{ \text{FBase } (\text{Plus-FO None } m1 m2 n), \text{FBool True, FBool False} \}$ 
      | Eq-FO False m1 m2  $\Rightarrow \{ \text{FBase } (\text{Eq-FO False } m1 m2), \text{FBool True, FBool False} \}$ 
      | Eq-SO M1 M2  $\Rightarrow \{ \text{FBase } (\text{Eq-SO } M1 M2), \text{FBool False} \}$ 
      | Suc-SO False bl M1 M2  $\Rightarrow \{ \text{FBase } (\text{Suc-SO False True } M1 M2), \text{FBase } (\text{Suc-SO False False } M1 M2), \text{FBool False} \}$ 
      | Empty M  $\Rightarrow \{ \text{FBase } (\text{Empty } M), \text{FBool False} \}$ 
      | Singleton M  $\Rightarrow \{ \text{FBase } (\text{Singleton } M), \text{FBase } (\text{Empty } M), \text{FBool False} \}$ 
      | Subset M1 M2  $\Rightarrow \{ \text{FBase } (\text{Subset } M1 M2), \text{FBool False} \}$ 
      | In False i I  $\Rightarrow \{ \text{FBase } (\text{In False } i I), \text{FBool True, FBool False} \}$ 
      | Eq-Max False m M  $\Rightarrow \{ \text{FBase } (\text{Eq-Max False } m M), \text{FBase } (\text{Empty } M), \text{FBool False} \}$ 
      | Eq-Min False m M  $\Rightarrow \{ \text{FBase } (\text{Eq-Min False } m M), \text{FBool True, FBool False} \}$ 
      | Eq-Union M1 M2 M3  $\Rightarrow \{ \text{FBase } (\text{Eq-Union } M1 M2 M3), \text{FBool False} \}$ 
      | Eq-Inter M1 M2 M3  $\Rightarrow \{ \text{FBase } (\text{Eq-Inter } M1 M2 M3), \text{FBool False} \}$ 
      | Eq-Diff M1 M2 M3  $\Rightarrow \{ \text{FBase } (\text{Eq-Diff } M1 M2 M3), \text{FBool False} \}$ 
      | Disjoint M1 M2  $\Rightarrow \{ \text{FBase } (\text{Disjoint } M1 M2), \text{FBool False} \}$ 
      | Eq-Presb None M n  $\Rightarrow \{ \text{FBase } (\text{Eq-Presb None } M i) \mid i . i \leq n \} \cup \{ \text{FBool False} \}$ 
    )

```

```

False}
  | - ⇒ {}) for a
  { fix xs
    note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
  Φ-def[simp]
    from ⟨lformula0 a⟩ have FBase a ∈ Φ a by auto
    moreover have ∧x φ. φ ∈ Φ a ⇒ d x φ ∈ Φ a
      by (auto simp: d-def split: atomic.splits list.splits bool.splits if-splits op-
tion.splits)
    then have ∧φ. φ ∈ Φ a ⇒ fold d xs φ ∈ Φ a by (induct xs) auto
    ultimately have fold d xs (FBase a) ∈ Φ a by blast
  }
  moreover have finite (Φ a) using ⟨lformula0 a⟩ unfolding Φ-def by (auto
split: atomic.splits)
  ultimately show finite {fold d xs (FBase a) | xs. True} by (blast intro: fi-
nite-subset)
next
fix a :: ws1s
define d where d = Formula-Operations.deriv extend rderiv0
define Φ :: - ⇒ (ws1s, order) aformula set
  where Φ a =
    (case a of
      Fo m ⇒ {FBase (Fo m), FBool True}
    | Eq-Const i m n ⇒
      {FBase (Eq-Const (Some j) m n) | j . j ≤ (case i of Some i ⇒ max i n |
- ⇒ n)} ∪
      {FBase (Eq-Const None m n)}
    | Less b m1 m2 ⇒ {FBase (Less None m1 m2), FBase (Less (Some True)
m1 m2),
      FBase (Less (Some False) m1 m2)}
    | Plus-FO i m1 m2 n ⇒
      {FBase (Plus-FO (Some j) m1 m2 n) | j . j ≤ (case i of Some i ⇒ max i
n | - ⇒ n)} ∪
      {FBase (Plus-FO None m1 m2 n)}
    | Eq-FO b m1 m2 ⇒ {FBase (Eq-FO True m1 m2), FBase (Eq-FO False m1
m2)}
    | Eq-SO M1 M2 ⇒ {FBase (Eq-SO M1 M2), FBool False}
    | Suc-SO br bl M1 M2 ⇒ {FBase (Suc-SO False True M1 M2), FBase
(Suc-SO False False M1 M2),
      FBase (Suc-SO True True M1 M2), FBase (Suc-SO True False M1 M2),
FBool False}
    | Empty M ⇒ {FBase (Empty M), FBool False}
    | Singleton M ⇒ {FBase (Singleton M), FBase (Empty M), FBool False}
    | Subset M1 M2 ⇒ {FBase (Subset M1 M2), FBool False}
    | In b i I ⇒ {FBase (In True i I), FBase (In False i I)}
    | Eq-Max b m M ⇒ {FBase (Eq-Max False m M), FBase (Eq-Max True m
M), FBool False}
    | Eq-Min b m M ⇒ {FBase (Eq-Min False m M), FBase (Eq-Min True m
M)}
  )

```

```

| Eq-Union M1 M2 M3 ⇒ {FBase (Eq-Union M1 M2 M3), FBool False}
| Eq-Inter M1 M2 M3 ⇒ {FBase (Eq-Inter M1 M2 M3), FBool False}
| Eq-Diff M1 M2 M3 ⇒ {FBase (Eq-Diff M1 M2 M3), FBool False}
| Disjoint M1 M2 ⇒ {FBase (Disjoint M1 M2), FBool False}
| Eq-Presb i M n ⇒ {FBase (Eq-Presb (Some l) M j) | j l .
  j ≤ (case i of Some i ⇒ max i n | - ⇒ n) ∧ l ≤ (case i of Some i ⇒ max
i n | - ⇒ n)} ∪
  {FBase (Eq-Presb None M n), FBool False} } for a
{ fix xs
note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
Φ-def[simp]
then have FBase a ∈ Φ a by (auto split: atomic.splits option.splits)
moreover have ∧x φ. φ ∈ Φ a ⇒ d x φ ∈ Φ a
by (auto simp add: d-def Let-def not-le gr0-conv-Suc leD[OF ld-bounded]
split: atomic.splits list.splits bool.splits if-splits option.splits nat.splits)
then have ∧φ. φ ∈ Φ a ⇒ fold d xs φ ∈ Φ a
by (induct xs) auto
ultimately have fold d xs (FBase a) ∈ Φ a by blast
}
moreover have finite (Φ a) unfolding Φ-def using [[simproc add: finite-Collect]]
by (auto split: atomic.splits)
ultimately show finite {fold d xs (FBase a) | xs. True} by (blast intro: fi-
nite-subset)
next
fix k l and a :: ws1s
show find0 k l a ↔ l ∈ FV0 k a by (induct a rule: find0.induct) auto
next
fix a :: ws1s and k :: order
show finite (FV0 k a) by (cases k) (induct a, auto)+
next
fix idx a k v
assume wf0 idx a v ∈ FV0 k a
then show LESS k v idx by (cases k) (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 and i :: nat
show Formula-Operations.lformula lformula0 (Restrict k i)
unfolding Restrict-def by (cases k) (auto simp: Formula-Operations.lformula.simps)
next
fix i ℳ k P r
assume iℳk = P
then show restrict k P ↔
  Formula-Operations.satisfies-gen Extend Length satisfies0 r ℳ (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*  
*dec-def eval-def cut-def len-downshift-helper dest!: CONS-surj*  
*dest: fMax-ge fMax-ffilter-less fMax-boundedD fsubset-fsingletonD*  
*split!: order.splits if-splits*)

**lemma** *check-equiv-code*[code]: *check-equiv idx r s =*  
*((ws1s-wf idx r ∧ ws1s-lformula r) ∧ (ws1s-wf idx s ∧ ws1s-lformula s) ∧*  
*(case rtrancl-while (λ(p, q). final idx p = final idx q)*  
*(λ(p, q). map (λa. (norm (deriv lderiv0 a p), norm (deriv lderiv0 a q))) (σ idx))*  
*(norm (RESTRICT r), norm (RESTRICT s)) of*  
*None ⇒ False*  
*| Some ([], x) ⇒ True*  
*| Some (a # list, x) ⇒ False))*  
**unfolding** *check-equiv-def WS1S.check-equiv-def WS1S.step-alt ..*

**definition** *while where* [code del, code-abbrev]: *while idx φ = while-default (fut-default idx φ)*

**declare** *while-default-code*[of fut-default idx φ for idx φ, folded while-def, code]

**lemma** *check-equiv-sound*:

$\llbracket \#_V \mathfrak{A} = \text{idx}; \text{check-equiv idx } \varphi \ \psi \rrbracket \implies (WS1S.\text{sat } \mathfrak{A} \ \varphi \longleftrightarrow WS1S.\text{sat } \mathfrak{A} \ \psi)$

**unfolding** *check-equiv-def by (rule WS1S.check-equiv-soundness)*

**lemma** *bounded-check-equiv-sound*:

$\llbracket \#_V \mathfrak{A} = \text{idx}; \text{bounded-check-equiv idx } \varphi \ \psi \rrbracket \implies (WS1S.\text{sat}_b \ \mathfrak{A} \ \varphi \longleftrightarrow WS1S.\text{sat}_b \ \mathfrak{A} \ \psi)$

**unfolding** *bounded-check-equiv-def by (rule WS1S.bounded-check-equiv-soundness)*

**method-setup** *check-equiv =* ‹

*let*

*fun tac ctxt =*

*let*

*val conv = @{computation-check terms: Trueprop*

*0 :: nat 1 :: nat 2 :: nat 3 :: nat Suc*

*plus :: nat ⇒ - minus :: nat ⇒ -*

*times :: nat ⇒ - divide :: nat ⇒ - modulo :: nat ⇒ -*

*0 :: int 1 :: int 2 :: int 3 :: int -1 :: int*

*check-equiv datatypes: formula int list integer idx*

*nat × nat nat option bool option} ctxt*

*in*

*CONVERSION (Conv.params-conv ~ 1 (K (Conv.concl-conv ~ 1 conv)) ctxt)*

*THEN'*

*resolve-tac ctxt [TrueI]*

*end*

*in*

*Scan.succeed (SIMPLE-METHOD' o tac)*

*end*

>

**end**

## 5 Concrete Atomic WS1S Formulas (Singleton Semantics for FO Variables)

```
datatype (FOV0: 'fo, SOV0: 'so) atomic =  
  Fo 'fo |  
  Z 'fo |  
  Less 'fo 'fo |  
  In 'fo 'so
```

```
derive linorder atomic
```

```
type-synonym fo = nat  
type-synonym so = nat  
type-synonym ws1s = (fo, so) atomic  
type-synonym formula = (ws1s, order) aformula
```

```
primrec wf0 where  
  wf0 idx (Fo m) = LESS FO m idx  
| wf0 idx (Z m) = LESS FO m idx  
| wf0 idx (Less m1 m2) = (LESS FO m1 idx  $\wedge$  LESS FO m2 idx)  
| wf0 idx (In m M) = (LESS FO m idx  $\wedge$  LESS SO M idx)
```

```
inductive lformula0 where  
  lformula0 (Fo m)  
| lformula0 (Z m)  
| lformula0 (Less m1 m2)  
| lformula0 (In m M)
```

```
code-pred lformula0 .
```

```
declare lformula0.intros[simp]
```

```
inductive-cases lformula0E[elim]: lformula0 a
```

```
abbreviation FV0  $\equiv$  case-order FOV0 SOV0
```

```
fun find0 where  
  find0 FO i (Fo m) = (i = m)  
| find0 FO i (Z m) = (i = m)  
| find0 FO i (Less m1 m2) = (i = m1  $\vee$  i = m2)  
| find0 FO i (In m -) = (i = m)  
| find0 SO i (In - M) = (i = M)  
| find0 - - - = False
```



**abbreviation**  $\text{decr0 ord } k \equiv \text{map-atomic (case-order (dec } k) \text{ id ord) (case-order id (dec } k) \text{ ord)}$

**primrec**  $\text{satisfies0}$  **where**

$\text{satisfies0 } \mathfrak{A} (Fo\ m) = (\exists x. m^{\mathfrak{A}}FO = \{|x|\})$   
 $| \text{satisfies0 } \mathfrak{A} (Z\ m) = (m^{\mathfrak{A}}FO = \{|\})$   
 $| \text{satisfies0 } \mathfrak{A} (\text{Less } m1\ m2) =$   
 $(\text{let } P1 = m1^{\mathfrak{A}}FO; P2 = m2^{\mathfrak{A}}FO \text{ in if } \neg(\exists x. P1 = \{|x|\}) \vee \neg(\exists x. P2 = \{|x|\})$   
 $\text{ then False}$   
 $\text{ else fthe-elem } P1 < \text{fthe-elem } P2)$   
 $| \text{satisfies0 } \mathfrak{A} (\text{In } m\ M) =$   
 $(\text{let } P = m^{\mathfrak{A}}FO \text{ in if } \neg(\exists x. P = \{|x|\}) \text{ then False else fMin } P \text{ |} \in \text{ } M^{\mathfrak{A}}SO)$

**fun**  $\text{lderiv0}$  **where**

$\text{lderiv0} (bs1, bs2) (Fo\ m) = (\text{if } bs1 ! m \text{ then } FBase (Z\ m) \text{ else } FBase (Fo\ m))$   
 $| \text{lderiv0} (bs1, bs2) (Z\ m) = (\text{if } bs1 ! m \text{ then } FBool\ False \text{ else } FBase (Z\ m))$   
 $| \text{lderiv0} (bs1, bs2) (\text{Less } m1\ m2) = (\text{case } (bs1 ! m1, bs1 ! m2) \text{ of}$   
 $(False, False) \Rightarrow FBase (\text{Less } m1\ m2)$   
 $| (True, False) \Rightarrow FAnd (FBase (Z\ m1)) (FBase (Fo\ m2))$   
 $| - \Rightarrow FBool\ False)$   
 $| \text{lderiv0} (bs1, bs2) (\text{In } m\ M) = (\text{case } (bs1 ! m, bs2 ! M) \text{ of}$   
 $(False, -) \Rightarrow FBase (\text{In } m\ M)$   
 $| (True, True) \Rightarrow FBase (Z\ m)$   
 $| - \Rightarrow FBool\ False)$

**primrec**  $\text{rev}$  **where**

$\text{rev} (Fo\ m) = Fo\ m$   
 $| \text{rev} (Z\ m) = Z\ m$   
 $| \text{rev} (\text{Less } m1\ m2) = \text{Less } m2\ m1$   
 $| \text{rev} (\text{In } m\ M) = \text{In } m\ M$

**abbreviation**  $\text{rderiv0 } v \equiv \text{map-aformula rev id o lderiv0 } v \text{ o rev}$

**primrec**  $\text{nullable0}$  **where**

$\text{nullable0} (Fo\ m) = False$   
 $| \text{nullable0} (Z\ m) = True$   
 $| \text{nullable0} (\text{Less } m1\ m2) = False$   
 $| \text{nullable0} (\text{In } m\ M) = False$

**lemma**  $\text{fimage-Suc-fsubset0[simp]}$ :  $\text{Suc } |^{\mathfrak{A}} A \text{ |} \subseteq \text{ } \{|0|\} \longleftrightarrow A = \{|\}$   
**by**  $\text{blast}$

**lemma**  $\text{fsubset-singleton-iff}$ :  $A \text{ |} \subseteq \text{ } \{|x|\} \longleftrightarrow A = \{|\} \vee A = \{|x|\}$   
**by**  $\text{blast}$

**definition**  $\text{restrict ord } P = (\text{case ord of } FO \Rightarrow \exists x. P = \{|x|\} \text{ | } SO \Rightarrow True)$

**definition**  $\text{Restrict ord } i = (\text{case ord of } FO \Rightarrow FBase (Fo\ i) \text{ | } SO \Rightarrow FBool\ True)$

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

**global-interpretation** *WS1S-Alt: Formula SUC LESS assigns nvars Extend CONS SNOC Length*

*extend size-atom zero  $\sigma$  eval downshift upshift finsert cut len restrict Restrict lformula0 FV0 find0 wf0 decr0 satisfies0 nullable0 lderiv0 rderiv0 undefined*

**defines** *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 find0 decr0 :: 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 :: -  $\Rightarrow$  formula*

**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-lformula = Formula-Operations.lformula lformula0 :: formula  $\Rightarrow$  -*

**and** *check-eqv =  $\lambda idx$ . DAs.check-eqv*

*( $\sigma$  idx) ( $\lambda \varphi$ . norm (RESTRICT  $\varphi$ ) :: (ws1s, order) aformula)*

*( $\lambda a$   $\varphi$ . norm (deriv (ldderiv0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula) (a :: atom)  $\varphi$ ))*

*(final idx) ( $\lambda \varphi$  :: formula. ws1s-wf idx  $\varphi$   $\wedge$  ws1s-lformula  $\varphi$ )*

*( $\sigma$  idx) ( $\lambda \varphi$ . norm (RESTRICT  $\varphi$ ) :: (ws1s, order) aformula)*

*( $\lambda a$   $\varphi$ . norm (deriv (ldderiv0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula) (a :: atom)  $\varphi$ ))*

*(final idx) ( $\lambda \varphi$  :: formula. ws1s-wf idx  $\varphi$   $\wedge$  ws1s-lformula  $\varphi$ ) (=)*

**and** *bounded-check-eqv =  $\lambda idx$ . DAs.check-eqv*

*( $\sigma$  idx) ( $\lambda \varphi$ . norm (RESTRICT  $\varphi$ ) :: (ws1s, order) aformula)*

*( $\lambda a$   $\varphi$ . norm (deriv (ldderiv0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula) (a :: atom)  $\varphi$ ))*

*nullable ( $\lambda \varphi$  :: formula. ws1s-wf idx  $\varphi$   $\wedge$  ws1s-lformula  $\varphi$ )*

*( $\sigma$  idx) ( $\lambda \varphi$ . norm (RESTRICT  $\varphi$ ) :: (ws1s, order) aformula)*

*( $\lambda a$   $\varphi$ . norm (deriv (ldderiv0 :: -  $\Rightarrow$  -  $\Rightarrow$  formula) (a :: atom)  $\varphi$ ))*

*nullable ( $\lambda \varphi$  :: formula. ws1s-wf idx  $\varphi$   $\wedge$  ws1s-lformula  $\varphi$ ) (=)*

**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 atomic.map find0.simps,*

```

      (transfer, force simp: dec-def split: if-splits order.splits)+) — slow
next
  fix k and a :: ws1s and l assume lformula0 a
  then show lformula0 (decr0 k l a) by (induct a) auto
next
  fix i k and a :: ws1s and  $\mathfrak{A}$  :: interp and P assume *:  $\neg$  find0 k i a LESS k i
  (SUC k ( $\#_V \mathfrak{A}$ ))
  and disj: lformula0 a  $\vee$  len P  $\leq$  Length  $\mathfrak{A}$ 
  from disj show satisfies0 (Extend k i  $\mathfrak{A}$  P) a = satisfies0  $\mathfrak{A}$  (decr0 k i a)
  proof
    assume lformula0 a
    then show ?thesis using *
      by (induct a)
      (auto simp: dec-def split: if-splits order.split option.splits bool.splits) — slow
  next
    assume len P  $\leq$  Length  $\mathfrak{A}$ 
    with * show ?thesis
    proof (induct a)
      case Fo then show ?case by (cases k) (auto simp: dec-def)
    next
      case Z then show ?case by (cases k) (auto simp: dec-def)
    next
      case Less then show ?case by (cases k) (auto simp: dec-def)
    next
      case In then show ?case by (cases k) (auto simp: dec-def)
    qed
  qed
next
  fix idx and a :: ws1s and x assume lformula0 a wf0 idx a
  then show Formula-Operations.wf SUC wf0 idx (lderiv0 x a)
  by (induct a rule: lderiv0.induct)
  (auto simp: Formula-Operations.wf.simps Let-def split: bool.splits order.splits)
next
  fix a :: ws1s and x assume lformula0 a
  then show Formula-Operations.lformula lformula0 (lderiv0 x a)
  by (induct a rule: lderiv0.induct)
  (auto simp: Formula-Operations.lformula.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 a rule: lderiv0.induct)
  (auto simp: Formula-Operations.wf.simps Let-def sorted-append
  split: bool.splits order.splits nat.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 3 dest: MSB-greater split: prod.splits if-splits option.splits)

```

```

bool.splits nat.splits) [])+ — slow
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 lformula0 a wf0 ( $\#_V \mathfrak{A}$ ) a  $\#_V \mathfrak{A} = \text{size-atom } x$ 
  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 lformula0 a wf0 ( $\#_V \mathfrak{A}$ ) a  $\#_V \mathfrak{A} = \text{size-atom } x$ 
  then show Formula-Operations.satisfies-bounded Extend Length len satisfies0  $\mathfrak{A}$ 
(lderiv0 x a) =
  satisfies0 (CONS x  $\mathfrak{A}$ ) a
  by (induct a) (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  $\#_V \mathfrak{A} = \text{size-atom } x$ 
  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
    apply (auto 2 0 split: prod.splits option.splits bool.splits)
    apply (auto simp add: fsubset-singleton-iff)
    apply (metis assigns-less-Length fininsertCI less-not-sym)
    apply force
    apply (metis assigns-less-Length fininsertCI less-not-sym)
    apply force
  done
  next
  case In then show ?case by (force split: prod.splits)
  qed (auto split: prod.splits)
next
  fix a :: ws1s and  $\mathfrak{A} \mathfrak{B}$  :: interp
  assume wf0 ( $\#_V \mathfrak{B}$ ) a  $\#_V \mathfrak{A} = \#_V \mathfrak{B} (\bigwedge m k. \text{LESS } k m (\#_V \mathfrak{B}) \implies m^{\mathfrak{A}} k = m^{\mathfrak{B}} k)$  lformula0 a
  then show satisfies0  $\mathfrak{A}$  a  $\longleftrightarrow$  satisfies0  $\mathfrak{B}$  a by (induct a) auto
next
  fix a :: ws1s
  assume lformula0 a
  moreover
  define d where d = Formula-Operations.deriv extend lderiv0
  define  $\Phi$  :: -  $\Rightarrow$  (ws1s, order) aformula set
  where  $\Phi a =$ 

```

```

(case a of
  Fo m ⇒ {FBase (Fo m), FBase (Z m), FBool False}
| Z m ⇒ {FBase (Z m), FBool False}
| Less m1 m2 ⇒ {FBase (Less m1 m2),
  FAnd (FBase (Z m1)) (FBase (Fo m2)),
  FAnd (FBase (Z m1)) (FBase (Z m2)),
  FAnd (FBase (Z m1)) (FBool False),
  FAnd (FBool False) (FBase (Fo m2)),
  FAnd (FBool False) (FBase (Z m2)),
  FAnd (FBool False) (FBool False),
  FBool False}
| In i I ⇒ {FBase (In i I), FBase (Z i), FBool False}) for a
{ fix xs
  note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
Φ-def[simp]
  from ⟨lformula0 a⟩ have FBase a ∈ Φ a by (cases a) auto
  moreover have  $\bigwedge x \varphi. \varphi \in \Phi a \implies d x \varphi \in \Phi a$ 
    by (auto simp: d-def split: atomic.splits list.splits bool.splits if-splits op-
tion.splits)
  then have  $\bigwedge \varphi. \varphi \in \Phi a \implies \text{fold } d \text{ xs } \varphi \in \Phi a$  by (induct xs) auto
  ultimately have fold d xs (FBase a) ∈ Φ a by blast
}
  moreover have finite (Φ a) using ⟨lformula0 a⟩ unfolding Φ-def by (auto
split: atomic.splits)
  ultimately show finite {fold d xs (FBase a) | xs. True} by (blast intro: fi-
nite-subset)
next
fix a :: ws1s
define d where d = Formula-Operations.deriv extend rderiv0
define Φ :: - ⇒ (ws1s, order) aformula set
  where Φ a =
    (case a of
      Fo m ⇒ {FBase (Fo m), FBase (Z m), FBool False}
| Z m ⇒ {FBase (Z m), FBool False}
| Less m1 m2 ⇒ {FBase (Less m1 m2),
  FAnd (FBase (Z m2)) (FBase (Fo m1)) ,
  FAnd (FBase (Z m2)) (FBase (Z m1)),
  FAnd (FBase (Z m2)) (FBool False),
  FAnd (FBool False) (FBase (Fo m1)),
  FAnd (FBool False) (FBase (Z m1)),
  FAnd (FBool False) (FBool False),
  FBool False}
| In i I ⇒ {FBase (In i I), FBase (Z i), FBool False}) for a
{ fix xs
  note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
Φ-def[simp]
  then have FBase a ∈ Φ a by (auto split: atomic.splits option.splits)
  moreover have  $\bigwedge x \varphi. \varphi \in \Phi a \implies d x \varphi \in \Phi a$ 
    by (auto simp add: d-def Let-def not-le gr0-conv-Suc

```

```

      split: atomic.splits list.splits bool.splits if-splits option.splits nat.splits)
    then have  $\bigwedge \varphi. \varphi \in \Phi a \implies \text{fold } d \text{ } xs \varphi \in \Phi a$ 
      by (induct xs) auto
    ultimately have  $\text{fold } d \text{ } xs (FBase a) \in \Phi a$  by blast
  }
  moreover have finite  $(\Phi a)$  unfolding  $\Phi$ -def using [[simproc add: finite-Collect]]
    by (auto split: atomic.splits)
  ultimately show finite  $\{\text{fold } d \text{ } xs (FBase a) \mid xs. \text{True}\}$  by (blast intro: fi-
nite-subset)
next
  fix k l and a :: ws1s
  show  $\text{find0 } k \text{ } l \text{ } a \longleftrightarrow l \in FV0 \text{ } k \text{ } a$  by (induct a rule: find0.induct) auto
next
  fix a :: ws1s and k :: order
  show finite  $(FV0 \text{ } k \text{ } a)$  by (cases k) (induct a, auto)+
next
  fix idx a k v
  assume wf0 idx a v  $\in FV0 \text{ } k \text{ } a$ 
  then show LESS k v idx by (cases k) (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 and i :: nat
  show Formula-Operations.lformula lformula0 (Restrict k i)
    unfolding Restrict-def by (cases k) (auto simp: Formula-Operations.lformula.simps)
next
  fix i  $\mathfrak{A} \text{ } k \text{ } P \text{ } r$ 
  assume  $i^{\mathfrak{A}} k = P$ 
  then show  $\text{restrict } k \text{ } P \longleftrightarrow$ 
    Formula-Operations.satisfies-gen Extend Length satisfies0 r  $\mathfrak{A}$  (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
dec-def eval-def cut-def len-downshift-helper CONS-inj dest!: CONS-surj
dest: fMax-ge fMax-filter-less fMax-boundedD fsubset-fsingletonD
split: order.splits if-splits)

```

**lemma** *check-equiv-code*[code]: *check-equiv idx r s =*  
 $((ws1s\text{-wf } idx \text{ } r \wedge ws1s\text{-lformula } r) \wedge (ws1s\text{-wf } idx \text{ } s \wedge ws1s\text{-lformula } s) \wedge$   
 $(\text{case } r\text{tranc1}\text{-while } (\lambda(p, q). \text{final } idx \text{ } p = \text{final } idx \text{ } q)$   
 $(\lambda(p, q). \text{map } (\lambda a. (\text{norm } (\text{deriv } l\text{deriv0 } a \text{ } p), \text{norm } (\text{deriv } l\text{deriv0 } a \text{ } q)))) (\sigma \text{ } idx))$   
 $(\text{norm } (RESTRICT \text{ } r), \text{norm } (RESTRICT \text{ } s)) \text{ of}$   
 $\text{None} \Rightarrow \text{False}$   
 $\mid \text{Some } ([], x) \Rightarrow \text{True}$

```

| Some (a # list, x) => False))
unfolding check-eqv-def WS1S-Alt.check-eqv-def WS1S-Alt.step-alt ..

definition while where [code del, code-abbrev]: while idx  $\varphi$  = while-default (fut-default
idx  $\varphi$ )
declare while-default-code[of fut-default idx  $\varphi$  for idx  $\varphi$ , folded while-def, code]

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

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

method-setup check-equiv = <
  let
    fun tac ctxt =
      let
        val conv = @{computation-check terms: Trueprop
          0 :: nat 1 :: nat 2 :: nat 3 :: nat Suc
          plus :: nat => - minus :: nat => -
          times :: nat => - divide :: nat => - modulo :: nat => -
          0 :: int 1 :: int 2 :: int 3 :: int -1 :: int
          check-eqv datatypes: formula int list integer idx
          nat  $\times$  nat nat option bool option} ctxt
        in
          CONVERSION (Conv.params-conv ~ 1 (K (Conv.concl-conv ~ 1 conv)) ctxt)
        THEN'
          resolve-tac ctxt [TrueI]
        end
      in
        Scan.succeed (SIMPLE-METHOD' o tac)
      end
  >
end

```

## 6 Concrete Atomic Presburger Formulas

```

declare [[coercion of-bool :: bool => nat]]
declare [[coercion int]]
declare [[coercion-map map]]
declare [[coercion-enabled]]

fun len :: nat => nat where — FIXME yet another logarithm

```

```

  len 0 = 0
| len (Suc 0) = 1
| len n = Suc (len (n div 2))

```

**lemma** *len-eq0-iff*:  $len\ n = 0 \longleftrightarrow n = 0$   
**by** (*induct n rule: len.induct*) *auto*

**lemma** *len-mult2[simp]*:  $len\ (2 * x) = (if\ x = 0\ then\ 0\ else\ Suc\ (len\ x))$   
**proof** (*induct x rule: len.induct*)  
**show**  $len\ (2 * Suc\ 0) = (if\ Suc\ 0 = 0\ then\ 0\ else\ Suc\ (len\ (Suc\ 0)))$  **by** (*simp*  
*add: numeral-eq-Suc*)  
**qed** *auto*

**lemma** *len-mult2'[simp]*:  $len\ (x * 2) = (if\ x = 0\ then\ 0\ else\ Suc\ (len\ x))$   
**using** *len-mult2 [of x]* **by** (*simp add: ac-simps*)

**lemma** *len-Suc-mult2[simp]*:  $len\ (Suc\ (2 * x)) = Suc\ (len\ x)$   
**proof** (*induct x rule: len.induct*)  
**show**  $len\ (Suc\ (2 * Suc\ 0)) = Suc\ (len\ (Suc\ 0))$   
**by** (*metis div-less One-nat-def div2-Suc-Suc len.simps(3) lessI mult.right-neutral*  
*numeral-2-eq-2*)  
**qed** *auto*

**lemma** *len-le-iff*:  $len\ x \leq l \longleftrightarrow x < 2 \wedge l$   
**proof** (*induct x arbitrary: l rule: len.induct*)  
**fix** *l* **show**  $(len\ (Suc\ 0) \leq l) = (Suc\ 0 < 2 \wedge l)$   
**proof** (*cases l*)  
**case** *Suc* **then show** *?thesis using le-less by fastforce*  
**qed** *simp*  
**next**  
**fix** *v l* **assume**  $\bigwedge l. (len\ (Suc\ (Suc\ v)\ div\ 2) \leq l) = (Suc\ (Suc\ v)\ div\ 2 < 2 \wedge l)$   
**then show**  $(len\ (Suc\ (Suc\ v)) \leq l) = (Suc\ (Suc\ v) < 2 \wedge l)$   
**by** (*cases l*) (*simp-all, linarith*)  
**qed** *simp*

**lemma** *len-pow2[simp]*:  $len\ (2 \wedge x) = Suc\ x$   
**by** (*induct x*) *auto*

**lemma** *len-div2[simp]*:  $len\ (x\ div\ 2) = len\ x - 1$   
**by** (*induct x rule: len.induct*) *auto*

**lemma** *less-pow2-len[simp]*:  $x < 2 \wedge len\ x$   
**by** (*induct x rule: len.induct*) *auto*

**lemma** *len-alt*:  $len\ x = (LEAST\ i. x < 2 \wedge i)$   
**proof** (*rule antisym*)  
**show**  $len\ x \leq (LEAST\ i. x < 2 \wedge i)$   
**unfolding** *len-le-iff* **by** (*rule LeastI*) (*rule less-pow2-len*)  
**qed** (*auto intro: Least-le*)



**lemma** *len-mono*[simp]:  $x \leq y \implies \text{len } x \leq \text{len } y$   
**unfolding** *len-le-iff* **using** *less-pow2-len*[of *y*] **by** *linarith*

**lemma** *len-div-pow2*[simp]:  $\text{len } (x \text{ div } 2 \wedge m) = \text{len } x - m$   
**by** (*induct m arbitrary: x*) (*auto simp: div-mult2-eq*)

**lemma** *len-mult-pow2*[simp]:  $\text{len } (x * 2 \wedge m) = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{len } x + m)$   
**by** (*induct m arbitrary: x*) (*auto simp: div-mult2-eq mult.assoc[symmetric] mult.commute[of - 2]*)

**lemma** *map-index'-Suc*[simp]:  $\text{map-index}' (\text{Suc } i) f \text{ xs} = \text{map-index}' i (\lambda i. f (\text{Suc } i)) \text{ xs}$   
**by** (*induct xs arbitrary: i*) *auto*

**abbreviation** (*input*) *zero*  $n \equiv \text{replicate } n \text{ False}$   
**abbreviation** (*input*) *SUC*  $\equiv \lambda :: \text{unit}. \text{Suc}$   
**definition** *test-bit*  $m \ n \equiv (m :: \text{nat}) \text{ div } 2 \wedge n \text{ mod } 2 = 1$   
**lemma** *test-bit-eq-bit*:  $\langle \text{test-bit} = \text{bit} \rangle$   
**by** (*simp add: fun-eq-iff test-bit-def bit-iff-odd-drop-bit mod-2-eq-odd flip: drop-bit-eq-div*)  
**definition** *downshift*  $m \equiv (m :: \text{nat}) \text{ div } 2$   
**definition** *upshift*  $m \equiv (m :: \text{nat}) * 2$   
**lemma** *set-bit-def*:  $\text{set-bit } n \ m \equiv m + (\text{if } \neg \text{test-bit } m \ n \text{ then } 2 \wedge n \text{ else } (0 :: \text{nat}))$   
**apply** (*rule eq-reflection*)  
**apply** (*rule bit-eqI*)  
**apply** (*subst disjunctive-add*)  
**apply** (*auto simp add: bit-simps test-bit-eq-bit*)  
**done**  
**definition** *cut-bits*  $n \ m \equiv (m :: \text{nat}) \text{ mod } 2 \wedge n$   
**lemma** *cut-bits-eq-take-bit*:  $\langle \text{cut-bits} = \text{take-bit} \rangle$   
**by** (*simp add: fun-eq-iff cut-bits-def take-bit-eq-mod*)

**typedef** *interp* =  $\{(n :: \text{nat}, \text{xs} :: \text{nat list}). \forall x \in \text{set } \text{xs}. \text{len } x \leq n\}$   
**by** (*force intro: exI[of - []]*)

**setup-lifting** *type-definition-interp*  
**type-synonym** *atom* = *bool list*  
**type-synonym** *value* = *nat*  
**datatype** *presb* = *Eq (tm: int list) (const: int) (offset: int)*  
**derive** *linorder list*  
**derive** *linorder presb*  
**type-synonym** *formula* = (*presb, unit*) *aformula*

**lift-definition** *assigns* ::  $\text{nat} \Rightarrow \text{interp} \Rightarrow \text{unit} \Rightarrow \text{value} \ (\langle \text{-} \rangle [900, 999, 999] 999)$   
**is**  
 $\lambda n \ (-, I) \ -. \ \text{if } n < \text{length } I \text{ then } I ! n \text{ else } 0 \ .$

**lift-definition** *nvars* ::  $\text{interp} \Rightarrow \text{nat} \ (\langle \#_V \rightarrow [1000] 900 \rangle) \text{ is}$

$\lambda(-, I). \text{length } I$  .

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

**lift-definition**  $\text{Extend} :: \text{unit} \Rightarrow \text{nat} \Rightarrow \text{interp} \Rightarrow \text{value} \Rightarrow \text{interp} \text{ is}$   
 $\lambda- i (n, I) m. (\text{max } n (\text{len } m), \text{insert-nth } i m I)$   
**by** (*force simp: max-def dest: in-set-takeD in-set-dropD*)

**lift-definition**  $\text{CONS} :: \text{atom} \Rightarrow \text{interp} \Rightarrow \text{interp} \text{ is}$   
 $\lambda bs (n, I). (\text{Suc } n, \text{map-index } (\lambda i n. 2 * n + (\text{if } bs ! i \text{ then } 1 \text{ else } 0))) I)$   
**by** (*auto simp: set-zip*)

**lift-definition**  $\text{SNOC} :: \text{atom} \Rightarrow \text{interp} \Rightarrow \text{interp} \text{ is}$   
 $\lambda bs (n, I). (\text{Suc } n, \text{map-index } (\lambda i m. m + (\text{if } bs ! i \text{ then } 2 ^ n \text{ else } 0))) I)$   
**by** (*auto simp: all-set-conv-all-nth len-le-iff*)

**definition**  $\text{extend} :: \text{unit} \Rightarrow \text{bool} \Rightarrow \text{atom} \Rightarrow \text{atom} \text{ where}$   
 $\text{extend } b bs \equiv b \# bs$

**abbreviation** (*input*)  $\text{size-atom} :: \text{atom} \Rightarrow \text{nat} \text{ where}$   
 $\text{size-atom} \equiv \text{length}$

**definition**  $\text{FV0} :: \text{unit} \Rightarrow \text{presb} \Rightarrow \text{nat} \text{ set} \text{ where}$   
 $\text{FV0 } fm = (\text{case } fm \text{ of } Eq \text{ is } - \Rightarrow \{n. n < \text{length } is \wedge is ! n \neq 0\})$

**lemma**  $\text{FV0-code}[code]:$

$\text{FV0 } x (Eq \text{ is } i \text{ off}) = \text{Option.these } (\text{set } (\text{map-index } (\lambda i x. \text{if } x = 0 \text{ then } \text{None} \text{ else } \text{Some } i) \text{ is}))$

**unfolding**  $\text{FV0-def}$  **by** (*force simp: Option.these-def image-iff*)

**primrec**  $\text{wf0} :: \text{nat} \Rightarrow \text{presb} \Rightarrow \text{bool} \text{ where}$   
 $\text{wf0 } idx (Eq \text{ is } -) = (\text{length } is = idx)$

**fun**  $\text{find0} \text{ where}$   
 $\text{find0 } (-::\text{unit}) n (Eq \text{ is } -) = (is ! n \neq 0)$

**primrec**  $\text{decr0} \text{ where}$   
 $\text{decr0 } (-::\text{unit}) k (Eq \text{ is } i d) = Eq (\text{take } k \text{ is } @ \text{drop } (\text{Suc } k) \text{ is}) i d$

**definition**  $\text{scalar-product} :: \text{nat} \text{ list} \Rightarrow \text{int} \text{ list} \Rightarrow \text{int} \text{ where}$   
 $\text{scalar-product } ns \text{ is} =$   
 $\text{sum-list } (\text{map-index } (\lambda i b. (\text{if } i < \text{length } ns \text{ then } ns ! i \text{ else } 0) * b) \text{ is})$

**lift-definition**  $\text{eval-tm} :: \text{interp} \Rightarrow \text{int} \text{ list} \Rightarrow \text{int} \text{ is}$   
 $\lambda(-, I). \text{scalar-product } I$  .

**primrec**  $\text{satisfies0} \text{ where}$   
 $\text{satisfies0 } I (Eq \text{ is } i d) = (\text{eval-tm } I \text{ is} = i - (2 ^ \text{Length } I) * d)$

**inductive** *lformula0* **where**  
*lformula0* (*Eq is i 0*)

**code-pred** *lformula0* .

**fun** *lderiv0* :: *bool list*  $\Rightarrow$  *presb*  $\Rightarrow$  *formula* **where**  
*lderiv0* *bs* (*Eq is i d*) = (*if d  $\neq$  0 then undefined else*  
*(let v = i - scalar-product bs is*  
*in if v mod 2 = 0 then FBase (Eq is (v div 2) 0) else FBool False))*

**fun** *rderiv0* :: *bool list*  $\Rightarrow$  *presb*  $\Rightarrow$  *formula* **where**  
*rderiv0* *bs* (*Eq is i d*) =  
*(let*  
*l = - sum-list [i. i  $\leftarrow$  is, i < 0];*  
*h = - sum-list [i. i  $\leftarrow$  is, i > 0];*  
*d' = scalar-product bs is + 2 \* d*  
*in if d'  $\in$  {min h i .. max l i} then FBase (Eq is i d') else FBool False)*

**primrec** *nullable0* **where**  
*nullable0* (*Eq is i off*) = (*i = off*)

**definition**  $\sigma$  :: *nat*  $\Rightarrow$  *atom list* **where**  
 $\sigma$  *n* = *List.n-lists n [True, False]*

**named-theorems** *Presb-simps*

**lemma** *nvars-Extend[Presb-simps]*:  $\#_V$  (*Extend* () *i*  $\mathfrak{A}$  *P*) = *Suc* ( $\#_V$   $\mathfrak{A}$ )  
**by** (*transfer, auto*)

**lemma** *Length-Extend[Presb-simps]*: *Length* (*Extend* () *i*  $\mathfrak{A}$  *P*) = *max* (*Length*  $\mathfrak{A}$ )  
(*len P*)  
**by** (*transfer, auto*)

**lemma** *Length0-inj[Presb-simps]*: *Length*  $\mathfrak{A}$  = 0  $\implies$  *Length*  $\mathfrak{B}$  = 0  $\implies$   $\#_V$   $\mathfrak{A}$  =  
 $\#_V$   $\mathfrak{B}$   $\implies$   $\mathfrak{A}$  =  $\mathfrak{B}$   
**by** *transfer (auto intro: nth-equalityI simp: all-set-conv-all-nth len-eq0-iff)*

**lemma** *ex-Length0[Presb-simps]*:  $\exists \mathfrak{A}. \text{Length } \mathfrak{A} = 0 \wedge \#_V \mathfrak{A} = \text{idx}$   
**by** (*transfer fixing: idx*) (*auto intro: exI[of - replicate idx 0]*)

**lemma** *Extend-commute-safe[Presb-simps]*:  $\llbracket j \leq i; i < \text{Suc } (\#_V \mathfrak{A}) \rrbracket \implies$   
*Extend k j* (*Extend k i*  $\mathfrak{A}$  *P*) *Q* = *Extend k* (*Suc i*) (*Extend k j*  $\mathfrak{A}$  *Q*) *P*  
**by** *transfer (auto simp add: min-def take-Cons take-drop le-imp-diff-is-add split: nat.splits)*

**lemma** *Extend-commute-unsafe[Presb-simps]*:  
*k  $\neq$  k'  $\implies$  Extend k j (Extend k' i  $\mathfrak{A}$  P) Q = Extend k' i (Extend k j  $\mathfrak{A}$  Q) P*  
**by** *transfer auto*

**lemma** *assigns-Extend*[Presb-simps]:  $i < \text{Suc } (\#_V \mathfrak{A}) \implies$   
 $m^{\text{Extend } k \ i \ \mathfrak{A}} P_{k'} = (\text{if } k = k' \text{ then if } m = i \text{ then } P \text{ else dec } i \ m^{\mathfrak{A}}_k \text{ else } m^{\mathfrak{A}}_{k'})$   
**by** *transfer* (*auto simp: nth-append dec-def min-def*)

**lemma** *assigns-SNOC-zero*[Presb-simps]:  $m < \#_V \mathfrak{A} \implies m^{\text{SNOC } (\text{zero } (\#_V \mathfrak{A}))} \mathfrak{A}_k$   
 $= m^{\mathfrak{A}}_k$   
**by** *transfer auto*

**lemma** *Length-CONS*[Presb-simps]:  $\text{Length } (\text{CONS } x \ \mathfrak{A}) = \text{Suc } (\text{Length } \mathfrak{A})$   
**by** *transfer auto*

**lemma** *Length-SNOC*[Presb-simps]:  $\text{Length } (\text{SNOC } x \ \mathfrak{A}) = \text{Suc } (\text{Length } \mathfrak{A})$   
**by** *transfer auto*

**lemma** *nvars-CONS*[Presb-simps]:  $\#_V (\text{CONS } x \ \mathfrak{A}) = \#_V \mathfrak{A}$   
**by** *transfer auto*

**lemma** *nvars-SNOC*[Presb-simps]:  $\#_V (\text{SNOC } x \ \mathfrak{A}) = \#_V \mathfrak{A}$   
**by** *transfer auto*

**lemma** *Extend-CONS*[Presb-simps]:  $\#_V \mathfrak{A} = \text{length } x \implies$   
 $\text{Extend } k \ 0 \ (\text{CONS } x \ \mathfrak{A}) \ P = \text{CONS } (\text{extend } k \ (\text{test-bit } P \ 0) \ x) \ (\text{Extend } k \ 0 \ \mathfrak{A}$   
 $(\text{downshift } P))$   
**by** *transfer* (*auto simp: extend-def downshift-def test-bit-def, presburger+*)

**lemma** *Extend-SNOC*[Presb-simps]:  $\llbracket \#_V \mathfrak{A} = \text{length } x; \text{len } P \leq \text{Length } (\text{SNOC } x$   
 $\mathfrak{A}) \rrbracket \implies$   
 $\text{Extend } k \ 0 \ (\text{SNOC } x \ \mathfrak{A}) \ P =$   
 $\text{SNOC } (\text{extend } k \ (\text{test-bit } P \ (\text{Length } \mathfrak{A})) \ x) \ (\text{Extend } k \ 0 \ \mathfrak{A} \ (\text{cut-bits } (\text{Length } \mathfrak{A})$   
 $P))$   
**apply** *transfer*  
**apply** (*auto simp: cut-bits-def extend-def test-bit-def nth-Cons' max-absorb1 len-le-iff*  
*split: if-splits cong del: if-weak-cong*)  
**apply** (*metis add.commute div-mult-mod-eq less-mult-imp-div-less mod-less mult-numeral-1*  
*numeral-1-eq-Suc-0*)  
**apply** (*metis div-eq-0-iff less-mult-imp-div-less mod2-eq-if mod-less power-not-zero*  
*zero-neq-numeral*)  
**done**

**lemma** *odd-neq-even*:  
 $\text{Suc } (2 * x) = 2 * y \longleftrightarrow \text{False}$   
 $2 * y = \text{Suc } (2 * x) \longleftrightarrow \text{False}$   
**by** *presburger+*

**lemma** *CONS-inj*[Presb-simps]:  $\text{size } x = \#_V \mathfrak{A} \implies \text{size } y = \#_V \mathfrak{A} \implies \#_V \mathfrak{A} =$   
 $\#_V \mathfrak{B} \implies$   
 $\text{CONS } x \ \mathfrak{A} = \text{CONS } y \ \mathfrak{B} \longleftrightarrow (x = y \wedge \mathfrak{A} = \mathfrak{B})$   
**by** *transfer* (*auto simp: list-eq-iff-nth-eq odd-neq-even split: if-splits*)

**lemma** *mod-2-Suc-iff*:

$x \bmod 2 = \text{Suc } 0 \iff x = \text{Suc } (2 * (x \text{ div } 2))$

**by** *presburger+*

**lemma** *CONS-surj*[*Presb-simps*]:  $\text{Length } \mathfrak{A} \neq 0 \implies$

$\exists x \mathfrak{B}. \mathfrak{A} = \text{CONS } x \mathfrak{B} \wedge \#_V \mathfrak{B} = \#_V \mathfrak{A} \wedge \text{size } x = \#_V \mathfrak{A}$

**by** *transfer*

(*auto simp: gr0-conv-Suc list-eq-iff-nth-eq len-le-iff split: if-splits*

*intro!*: *exI*[*of - map* ( $\lambda n. n \bmod 2 \neq 0$ ) -] *exI*[*of - map* ( $\lambda n. n \text{ div } 2$ ) -];

*auto simp: mod-2-Suc-iff*)

**lemma** [*Presb-simps*]:

$\text{length } (\text{extend } k \ b \ x) = \text{Suc } (\text{length } x)$

$\text{downshift } (\text{upshift } P) = P$

$\text{downshift } (\text{set-bit } 0 \ P) = \text{downshift } P$

$\text{test-bit } (\text{set-bit } n \ P) \ n$

$\neg \text{test-bit } (\text{upshift } P) \ 0$

$\text{len } P \leq p \implies \neg \text{test-bit } P \ p$

$\text{len } (\text{cut-bits } n \ P) \leq n$

$\text{len } P \leq n \implies \text{cut-bits } n \ P = P$

$\text{len } (\text{upshift } P) = (\text{case } \text{len } P \ \text{of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow \text{Suc } (\text{Suc } n))$

$\text{len } (\text{downshift } P) = (\text{case } \text{len } P \ \text{of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow n)$

**by** (*simp-all add: downshift-def upshift-def test-bit-eq-bit extend-def cut-bits-def*  
*bit-simps mult.commute* [*of - 2*] *len-le-iff len-eq0-iff set-bit-0 split: nat.split*)

(*simp add: bit-iff-odd*)

**lemma** *Suc0-div-pow2-eq*:  $\text{Suc } 0 \ \text{div } 2^i = (\text{if } i = 0 \ \text{then } 1 \ \text{else } 0)$

**by** (*induct i*) (*auto simp: div-mult2-eq*)

**lemma** *set-unset-bit-preserves-len*:

**assumes**  $x \ \text{div } 2^m = 2 * q \ m < \text{len } x$

**shows**  $x + 2^m < 2^{\text{len } x}$

**using** *assms proof* (*induct m arbitrary: x*)

**case** 0 **then show** ?*case*

**by** (*auto simp: div-mult2-eq len-Suc-mult2*[*symmetric*])

*simp del: len-Suc-mult2 power-Suc split: if-splits*)

**next**

**case** (*Suc m*)

**with** *Suc(1)*[*of x div 2*] **show** ?*case* **by** (*cases len x*) (*auto simp: div-mult2-eq*)

**qed**

**lemma** *len-set-bit*[*Presb-simps*]:  $\text{len } (\text{set-bit } m \ P) = \max (\text{Suc } m) (\text{len } P)$

**proof** (*rule antisym*)

**show**  $\text{len } (\text{set-bit } m \ P) \leq \max (\text{Suc } m) (\text{len } P)$

**by** (*auto simp: set-bit-def test-bit-def max-def Suc-le-eq not-less len-le-iff*

*set-unset-bit-preserves-len simp del: One-nat-def*)

**next**

**have**  $P < 2^{\text{len } (P + 2^m)}$  **by** (*rule order.strict-trans2*[*OF less-pow2-len*])

*auto*

**moreover have**  $m < \text{len } (P + 2^m)$  **by**  $(\text{rule order.strict-trans2}[\text{OF - len-mono}[\text{of } 2^m]]) \text{ auto}$   
**ultimately show**  $\max (\text{Suc } m) (\text{len } P) \leq \text{len } (\text{set-bit } m P)$   
**by**  $(\text{auto simp: set-bit-def test-bit-def max-def Suc-le-eq not-less len-le-iff})$   
**qed**

**lemma mod-pow2-div-pow2:**  
**fixes**  $p m n :: \text{nat}$   
**shows**  $m < n \implies p \bmod 2^n \text{ div } 2^m = p \text{ div } 2^m \bmod 2^{(n-m)}$   
**by**  $(\text{induct } m \text{ arbitrary: } p n) (\text{auto simp: div-mult2-eq mod-mult2-eq Suc-less-eq2})$

**lemma irrelevant-set-bit[simp]:**  
**fixes**  $p m n :: \text{nat}$   
**assumes**  $n \leq m$   
**shows**  $(p + 2^m) \bmod 2^n = p \bmod 2^n$   
**proof** –  
**from** *assms* **obtain**  $q :: \text{nat}$  **where**  $2^m = q * 2^n$   
**by**  $(\text{metis le-add-diff-inverse mult.commute power-add})$   
**then show** *?thesis* **by** *simp*  
**qed**

**lemma mod-lemma:**  $\llbracket (0 :: \text{nat}) < c; r < b \rrbracket \implies b * (q \bmod c) + r < b * c$   
**by**  $(\text{metis add-gr-0 div-le-mono div-mult-self1-is-m less-imp-add-positive mod-less-divisor not-less split-div})$

**lemma relevant-set-bit[simp]:**  
**fixes**  $p m n :: \text{nat}$   
**assumes**  $m < n$   $p \text{ div } 2^m = 2 * q$   
**shows**  $(p + 2^m) \bmod 2^n = p \bmod 2^n + 2^m$   
**proof** –  
**have**  $p \bmod 2^n + 2^m < 2^n$   
**using** *assms* **proof**  $(\text{induct } m \text{ arbitrary: } p n)$   
**case 0** **then show** *?case*  
**by**  $(\text{auto simp: gr0-conv-Suc})$   
 $(\text{metis One-nat-def Suc-eq-plus1 lessI mod-lemma numeral-2-eq-2 zero-less-numeral zero-less-power})$   
**next**  
**case**  $(\text{Suc } m)$   
**from**  $\text{Suc}(1)[\text{of } n - 1 \text{ } p \text{ div } 2] \text{ Suc}(2,3)$  **show** *?case*  
**by**  $(\text{auto simp: div-mult2-eq mod-mult2-eq Suc-less-eq2})$   
**qed**  
**with**  $\langle m < n \rangle$  **show** *?thesis* **by**  $(\text{subst mod-add-eq [symmetric]}) \text{ auto}$   
**qed**

**lemma cut-bits-set-bit[Presb-simps]:**  $\text{cut-bits } n (\text{set-bit } m p) =$   
 $(\text{if } n \leq m \text{ then } \text{cut-bits } n p \text{ else } \text{set-bit } m (\text{cut-bits } n p))$   
**unfolding** *cut-bits-def set-bit-def test-bit-def*  
**by**  $(\text{auto simp: not-le mod-pow2-div-pow2 mod-mod-cancel simp del: One-nat-def})$

**lemma** *wf0-decr0*[Presb-simps]:

*wf0 (Suc idx) a  $\implies$  l < Suc idx  $\implies$   $\neg$  find0 k l a  $\implies$  wf0 idx (decr0 k l a)*

**by** (*induct a*) *auto*

**lemma** *lformula0-decr0*[Presb-simps]: *lformula0 a  $\implies$  lformula0 (decr0 k l a)*

**by** (*induct a*) (*auto elim: lformula0.cases intro: lformula0.intros*)

**abbreviation** *sat0-syn* (**infix**  $\langle \models_0 \rangle$  65) **where**

*sat0-syn  $\equiv$  satisfies0*

**abbreviation** *sat-syn* (**infix**  $\langle \models \rangle$  65) **where**

*sat-syn  $\equiv$  Formula-Operations.satisfies Extend Length satisfies0*

**abbreviation** *sat-bounded-syn* (**infix**  $\langle \models_b \rangle$  65) **where**

*sat-bounded-syn  $\equiv$  Formula-Operations.satisfies-bounded Extend Length len satisfies0*

**lemma** *scalar-product-Nil*[simp]: *scalar-product [] xs = 0*

**by** (*induct xs*) (*auto simp: scalar-product-def*)

**lemma** *scalar-product-Nil2*[simp]: *scalar-product xs [] = 0*

**by** (*induct xs*) (*auto simp: scalar-product-def*)

**lemma** *scalar-product-Cons*[simp]:

*scalar-product xs (y # ys) = (case xs of x # xs  $\Rightarrow$  x \* y + scalar-product xs ys | []  $\Rightarrow$  0)*

**by** (*cases xs*) (*simp, auto simp: scalar-product-def cong del: if-weak-cong*)

**lemma** *scalar-product-append*[simp]: *scalar-product ns (xs @ ys) =*

*scalar-product (take (length xs) ns) xs + scalar-product (drop (length xs) ns) ys*

**by** (*induct xs arbitrary: ns*) (*auto split: list.splits*)

**lemma** *scalar-product-trim*: *scalar-product ns xs = scalar-product (take (length xs) ns) xs*

**by** (*induct xs arbitrary: ns*) (*auto split: list.splits*)

**lemma** *Extend-satisfies0-decr0*[Presb-simps]:

**assumes**  $\neg$  *find0 k i a i < Suc (# $\vee$   $\mathfrak{A}$ ) lformula0 a  $\vee$  len P  $\leq$  Length  $\mathfrak{A}$*

**shows** *Extend k i  $\mathfrak{A}$  P  $\models_0$  a =  $\mathfrak{A}$   $\models_0$  decr0 k i a*

**proof** –

{ **fix** *is :: int list*

**assume** *is ! i = 0*

**with** *assms(1,2)* **have** *eval-tm (Extend k i  $\mathfrak{A}$  P) is = eval-tm  $\mathfrak{A}$  (take i is @ drop (Suc i) is)*

**by** (*cases a, transfer*)

(*force intro: trans[OF scalar-product-trim] simp: min-def*

*arg-cong2[OF refl id-take-nth-drop, of i - scalar-product take i xs @ - for i x xs]*)

} **note** *\* = this*

**from** *assms* **show** *?thesis*

**by** (*cases a*) (*auto dest!: \* simp: Length-Extend max-def elim: lformula0.cases*)

**qed**

**lemma** *scalar-product-eq0*:  $\forall c \in \text{set } ns. c = 0 \implies \text{scalar-product } ns \text{ is } = 0$

**proof** (*induct is arbitrary: ns*)

**case** *Cons*

**then show** ?*case by* (*cases ns*) (*auto simp: scalar-product-def cong del: if-weak-cong*)

**qed** (*simp add: scalar-product-def*)

**lemma** *nullable0-satisfies0*[*Presb-simps*]:  $\text{Length } \mathfrak{A} = 0 \implies \text{nullable0 } a = \mathfrak{A} \models 0$   
*a*

**proof** (*induct a*)

**case** *Eq* **then show** ?*case unfolding* *nullable0.simps satisfies0.simps*

**by** *transfer* (*auto simp: len-eq0-iff scalar-product-eq0*)

**qed**

**lemma** *satisfies0-cong*:  $\text{wf0 } (\#_V \mathfrak{B}) a \implies \#_V \mathfrak{A} = \#_V \mathfrak{B} \implies \text{lformula0 } a \implies$

$(\bigwedge m k. m < \#_V \mathfrak{B} \implies m^{\mathfrak{A}k} = m^{\mathfrak{B}k}) \implies \mathfrak{A} \models 0 a = \mathfrak{B} \models 0 a$

**proof** (*induct a*)

**case** *Eq* **then show** ?*case unfolding* *satisfies0.simps*

**by** *transfer* (*auto simp: scalar-product-def*)

*intro!*: *arg-cong*[*of - - sum-list*] *map-index-cong elim!*: *lformula0.cases*)

**qed**

**lemma** *wf-ldderiv0*[*Presb-simps*]:

$\text{wf0 } idx \ a \implies \text{lformula0 } a \implies \text{Formula-Operations.wf } (\lambda-. \text{Suc}) \ \text{wf0 } idx \ (\text{ldderiv0 } x \ a)$

**by** (*induct a*) (*auto elim: lformula0.cases simp: Formula-Operations.wf.simps Let-def*)

**lemma** *lformula-ldderiv0*[*Presb-simps*]:

$\text{lformula0 } a \implies \text{Formula-Operations.lformula } \text{lformula0 } (\text{ldderiv0 } x \ a)$

**by** (*induct a*)

(*auto elim: lformula0.cases intro: lformula0.intros simp: Let-def Formula-Operations.lformula.simps*)

**lemma** *wf-rderiv0*[*Presb-simps*]:

$\text{wf0 } idx \ a \implies \text{Formula-Operations.wf } (\lambda-. \text{Suc}) \ \text{wf0 } idx \ (\text{rderiv0 } x \ a)$

**by** (*induct a*) (*auto elim: lformula0.cases simp: Formula-Operations.wf.simps Let-def*)

**lemma** *find0-FV0*[*Presb-simps*]:  $\llbracket \text{wf0 } idx \ a; l < idx \rrbracket \implies \text{find0 } k \ l \ a = (l \in \text{FV0 } k \ a)$

**by** (*induct a*) (*auto simp: FV0-def*)

**lemma** *FV0-less*[*Presb-simps*]:  $\text{wf0 } idx \ a \implies v \in \text{FV0 } k \ a \implies v < idx$

**by** (*induct a*) (*auto simp: FV0-def*)

**lemma** *finite-FV0*[*Presb-simps*]:  $\text{finite } (\text{FV0 } k \ a)$

**by** (*induct a*) (*auto simp: FV0-def*)



```

lemma finite-lderiv0[Presb-simps]:
  assumes lformula0 a
  shows finite { $\varphi$ .  $\exists xs$ .  $\varphi = \text{fold} (\text{Formula-Operations.deriv extend lderiv0}) xs$ 
    (FBase a)}
proof -
  define d where d = Formula-Operations.deriv extend lderiv0
  define l where l is = sum-list [i. i  $\leftarrow$  is, i < 0] for is :: int list
  define h where h is = sum-list [i. i  $\leftarrow$  is, i > 0] for is :: int list
  define  $\Phi$  where  $\Phi$  a = (case a of
    Eq is n z  $\Rightarrow$  {FBase (Eq is i 0) | i . i  $\in$  {min (- h is) n .. max (- l is) n}}  $\cup$ 
    {FBool False :: formula}) for a
  { fix xs
    note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
     $\Phi$ -def[simp]
    from <lformula0 a> have FBase a  $\in$   $\Phi$  a by (auto simp: elim!: lformula0.cases)
    moreover have  $\bigwedge x \varphi$ .  $\varphi \in \Phi$  a  $\Longrightarrow$  d x  $\varphi \in \Phi$  a
    proof (induct a, unfold  $\Phi$ -def presb.case, elim UnE CollectE insertE emptyE
    exE conjE)
      fix is :: int list and bs :: bool list and i n :: int and  $\varphi$  :: formula
      assume i  $\in$  {min (- h is) n..max (- l is) n}  $\varphi = \text{FBase} (\text{presb.Eq is i 0})$ 
      moreover have scalar-product bs is  $\leq$  h is
      proof (induct is arbitrary: bs)
        case (Cons x xs)
          from Cons[of tl bs] show ?case by (cases bs) (auto simp: h-def)
        qed (auto simp: h-def scalar-product-def)
      moreover have l is  $\leq$  scalar-product bs is
      proof (induct is arbitrary: bs)
        case (Cons x xs)
          from Cons[of tl bs] show ?case by (cases bs) (auto simp: l-def)
        qed (auto simp: l-def scalar-product-def)
      ultimately show d bs  $\varphi \in$ 
        {FBase (presb.Eq is i 0) | i. i  $\in$  {min (- h is) n..max (- l is) n}}  $\cup$  {FBool
        False}
      by (auto simp: d-def Let-def)
    qed (auto simp: d-def)
    then have  $\bigwedge \varphi$ .  $\varphi \in \Phi$  a  $\Longrightarrow$  fold d xs  $\varphi \in \Phi$  a by (induct xs) auto
    ultimately have fold d xs (FBase a)  $\in$   $\Phi$  a by blast
  }
  moreover have finite ( $\Phi$  a) unfolding  $\Phi$ -def by (auto split: presb.splits)
  ultimately show ?thesis unfolding d-def by (blast intro: finite-subset)
qed

```

```

lemma finite-rderiv0[Presb-simps]:
  finite { $\varphi$ .  $\exists xs$ .  $\varphi = \text{fold} (\text{Formula-Operations.deriv extend rderiv0}) xs$  (FBase a)}
proof -
  define d where d = Formula-Operations.deriv extend rderiv0
  define l where l is = sum-list [i. i  $\leftarrow$  is, i < 0] for is :: int list
  define h where h is = sum-list [i. i  $\leftarrow$  is, i > 0] for is :: int list
  define  $\Phi$  where  $\Phi$  a = (case a of

```

```

    Eq is n z ⇒ {FBase (Eq is n i) | i . i ∈ {min (- h is) (min n z) .. max (- l
is) (max n z)}} ∪
    {FBool False :: formula} for a
  { fix xs
    note Formula-Operations.fold-deriv-FBool[simp] Formula-Operations.deriv.simps[simp]
Φ-def[simp]
    have FBase a ∈ Φ a by (auto split: presb.splits)
    moreover have ∧x φ. φ ∈ Φ a ⇒ d x φ ∈ Φ a
    proof (induct a, unfold Φ-def presb.case, elim UnE CollectE insertE emptyE
exE conjE)
      fix is :: int list and bs :: bool list and i n m :: int and φ :: formula
      assume i ∈ {min (- h is) (min n m)..max (- l is) (max n m)} φ = FBase
(presb.Eq is n i)
      moreover have scalar-product bs is ≤ h is
      proof (induct is arbitrary: bs)
        case (Cons x xs)
          from Cons[of tl bs] show ?case by (cases bs) (auto simp: h-def)
        qed (auto simp: h-def scalar-product-def)
      moreover have l is ≤ scalar-product bs is
      proof (induct is arbitrary: bs)
        case (Cons x xs)
          from Cons[of tl bs] show ?case by (cases bs) (auto simp: l-def)
        qed (auto simp: l-def scalar-product-def)
      ultimately show d bs φ ∈
        {FBase (presb.Eq is n i) | i. i ∈ {min (- h is) (min n m)..max (- l is) (max
n m)}} ∪ {FBool False}
        by (auto 0 1 simp: d-def Let-def h-def l-def)
      qed (auto simp: d-def)
    then have ∧φ. φ ∈ Φ a ⇒ fold d xs φ ∈ Φ a by (induct xs) auto
    ultimately have fold d xs (FBase a) ∈ Φ a by blast
  }
  moreover have finite (Φ a) unfolding Φ-def by (auto split: presb.splits)
  ultimately show ?thesis unfolding d-def by (blast intro: finite-subset)
qed

```

**lemma** *scalar-product-CONS*:  $\text{length } xs = \text{length } (bs :: \text{bool list}) \implies$   
 $\text{scalar-product } (\text{map-index } (\lambda i n. 2 * n + bs ! i) xs) is =$   
 $\text{scalar-product } bs is + 2 * \text{scalar-product } xs is$   
**by** (induct is arbitrary: bs xs) (auto split: list.splits simp: algebra-simps)

**lemma** *eval-tm-CONS*[simp]:  
 $\llbracket \text{length } is \leq \#_V \mathfrak{A}; \#_V \mathfrak{A} = \text{length } x \rrbracket \implies$   
 $\text{eval-tm } (CONS x \mathfrak{A}) is = \text{scalar-product } x is + 2 * \text{eval-tm } \mathfrak{A} is$   
**by** transfer (auto simp: scalar-product-CONS[symmetric]  
intro!: arg-cong2[of - - - scalar-product] nth-equalityI)

**lemma** *satisfies-lderiv0*[Presb-simps]:  
 $\llbracket wf0 (\#_V \mathfrak{A}) a; \#_V \mathfrak{A} = \text{length } x; lformula0 a \rrbracket \implies \mathfrak{A} \models \text{lderiv0 } x a \longleftrightarrow CONS$   
 $x \mathfrak{A} \models 0 a$

**by** (*auto simp: Let-def Formula-Operations.satisfies-gen.simps*  
*split: if-splits elim!: lformula0.cases*)

**lemma** *satisfies-bounded-lderiv0*[Presb-simps]:

$\llbracket wf0 (\#_V \mathfrak{A}) a; \#_V \mathfrak{A} = \text{length } x; lformula0 a \rrbracket \implies \mathfrak{A} \models_b \text{lderiv0 } x a \longleftrightarrow CONS$   
 $x \mathfrak{A} \models_0 a$

**by** (*auto simp: Let-def Formula-Operations.satisfies-gen.simps*  
*split: if-splits elim!: lformula0.cases*)

**lemma** *scalar-product-SNOC*:  $\text{length } xs = \text{length } (bs :: \text{bool list}) \implies$

$\text{scalar-product } (\text{map-index } (\lambda i m. m + 2^a * bs ! i) xs) is =$   
 $\text{scalar-product } xs is + 2^a * \text{scalar-product } bs is$

**by** (*induct is arbitrary: bs xs*) (*auto split: list.splits simp: algebra-simps*)

**lemma** *eval-tm-SNOC*[simp]:

$\llbracket \text{length } is \leq \#_V \mathfrak{A}; \#_V \mathfrak{A} = \text{length } x \rrbracket \implies$

$\text{eval-tm } (SNOC x \mathfrak{A}) is = \text{eval-tm } \mathfrak{A} is + 2^{\text{Length } \mathfrak{A}} * \text{scalar-product } x is$

**by** *transfer* (*auto simp: scalar-product-SNOC[symmetric]*  
*intro!: arg-cong2[of - - - scalar-product] nth-equalityI*)

**lemma** *Length-eq0-eval-tm-eq0*[simp]:  $\text{Length } \mathfrak{A} = 0 \implies \text{eval-tm } \mathfrak{A} is = 0$

**by** *transfer* (*auto simp: len-eq0-iff scalar-product-eq0*)

**lemma** *less-pow2*:  $x < 2^a \implies \text{int } x < 2^a$

**by** (*metis of-nat-less-iff of-nat-numeral of-nat-power [symmetric]*)

**lemma** *scalar-product-upper-bound*:  $\forall x \in \text{set } b. \text{len } x \leq a \implies$

$\text{scalar-product } b is \leq (2^a - 1) * \text{sum-list } [i. i \leftarrow is, i > 0]$

**proof** (*induct is arbitrary: b*)

**case** (*Cons i is*)

**then have**  $\text{scalar-product } (tl b) is \leq (2^a - 1) * \text{sum-list } [i. i \leftarrow is, i > 0]$

**by** (*auto simp: in-set-tlD*)

**with** *Cons(2)* **show** *?case*

**by** (*auto 0 3 split: list.splits simp: len-le-iff mult-le-0-iff*

*distrib-left add commute[of - (2^a - 1) \* i] less-pow2*

*intro: add-mono elim: order-trans[OF add-mono[OF order-refl]]*)

**qed** *simp*

**lemma** *scalar-product-lower-bound*:  $\forall x \in \text{set } b. \text{len } x \leq a \implies$

$\text{scalar-product } b is \geq (2^a - 1) * \text{sum-list } [i. i \leftarrow is, i < 0]$

**proof** (*induct is arbitrary: b*)

**case** (*Cons i is*)

**then have**  $\text{scalar-product } (tl b) is \geq (2^a - 1) * \text{sum-list } [i. i \leftarrow is, i < 0]$

**by** (*auto simp: in-set-tlD*)

**with** *Cons(2)* **show** *?case*

**by** (*auto 0 3 split: list.splits simp: len-le-iff mult-le-0-iff*

*distrib-left add commute[of - (2^a - 1) \* i] less-pow2*

*intro: add-mono elim: order-trans[OF add-mono[OF order-refl]] order-trans*)

**qed** *simp*

**lemma** *eval-tm-upper-bound*:  $eval\text{-}tm\ \mathfrak{A}\ is \leq (2 \wedge Length\ \mathfrak{A} - 1) * sum\text{-}list\ [i.\ i \leftarrow is, i > 0]$

**by** *transfer* (*auto simp: scalar-product-upper-bound*)

**lemma** *eval-tm-lower-bound*:  $eval\text{-}tm\ \mathfrak{A}\ is \geq (2 \wedge Length\ \mathfrak{A} - 1) * sum\text{-}list\ [i.\ i \leftarrow is, i < 0]$

**by** *transfer* (*auto simp: scalar-product-lower-bound*)

**lemma** *satisfies-bounded-rderiv0*[*Presb-simps*]:

$\llbracket wf0\ (\#_V\ \mathfrak{A})\ a;\ \#_V\ \mathfrak{A} = length\ x \rrbracket \implies \mathfrak{A} \models_b\ rderiv0\ x\ a \longleftrightarrow SNOC\ x\ \mathfrak{A} \models_0\ a$

**proof** (*induct a*)

**case** (*Eq is n d*)

**let**  $?l = Length\ \mathfrak{A}$

**define**  $d'$  **where**  $d' = scalar\text{-}product\ x\ is + 2 * d$

**define**  $l$  **where**  $l = sum\text{-}list\ [i.\ i \leftarrow is, i < 0]$

**define**  $h$  **where**  $h = sum\text{-}list\ [i.\ i \leftarrow is, i > 0]$

**from** *Eq* **show**  $?case$

**unfolding** *wf0.simps satisfies0.simps rderiv0.simps Let-def*

**proof** (*split if-splits, simp only: Formula-Operations.satisfies-gen.simps satisfies0.simps*)

*Length-SNOC eval-tm-SNOC simp-thms(13) de-Morgan-conj not-le*

*min-less-iff-conj max-less-iff-conj d'-def[symmetric] h-def[symmetric] l-def[symmetric],*

*safe*)

**assume**  $eval\text{-}tm\ \mathfrak{A}\ is + 2 \wedge ?l * scalar\text{-}product\ x\ is = n - 2 \wedge Suc\ ?l * d$

**with** *eval-tm-upper-bound*[*of \mathfrak{A} is*] *eval-tm-lower-bound*[*of \mathfrak{A} is*] **have**

$*$ :  $n + h \leq 2 \wedge ?l * (h + d')\ 2 \wedge ?l * (l + d') \leq n + l$

**by** (*auto simp: algebra-simps h-def l-def d'-def*)

**assume**  $**$ :  $d' \notin \{min\ (-h)\ n..max\ (-l)\ n\}$

**{ assume**  $0 \leq n + h$

**from** *order-trans*[*OF this \*(1)*] **have**  $0 \leq h + d'$

**unfolding** *zero-le-mult-iff* **using** *zero-less-power*[*of 2*] **by** *presburger*

**}**

**moreover**

**{ assume**  $n + h < 0$

**with**  $*(1)$  **have**  $n \leq d'$  **by** (*auto dest: order-trans*[*OF - mult-right-mono-neg*][*of 1*]])

**}**

**moreover**

**{ assume**  $n + l < 0$

**from** *le-less-trans*[*OF \*(2) this*] **have**  $l + d' < 0$  **unfolding** *mult-less-0-iff*

**by** *auto*

**}**

**moreover**

**{ assume**  $0 \leq n + l$

**then** **have**  $0 \leq l + d'$  **using**  $**$  *calculation(1-2)* **by** *force*

**with**  $*(2)$  **have**  $d' \leq n$  **by** (*force dest: order-trans*[*OF mult-right-mono*][*of 1*],

*rotated*)

**}**

```

ultimately have min (- h) n ≤ d' d' ≤ max (- l) n by (auto simp: min-def
max-def)
with ** show False by auto
qed (auto simp: algebra-simps d'-def)
qed

```

```

declare [[goals-limit = 100]]

```

```

global-interpretation Presb: Formula

```

```

where SUC = SUC and LESS = λ-. (<) and Length = Length
and assigns = assigns and nvars = nvars and Extend = Extend and CONS =
CONS and SNOC = SNOC
and extend = extend and size = size-atom and zero = zero and alphabet = σ
and eval = test-bit
and downshift = downshift and upshift = upshift and add = set-bit and cut =
cut-bits and len = len
and restrict = λ- -. True and Restrict = λ- -. FBool True and lformula0 =
lformula0
and FV0 = FV0 and find0 = find0 and wf0 = wf0 and decr0 = decr0 and
satisfies0 = satisfies0
and nullable0 = nullable0 and lderiv0 = lderiv0 and rderiv0 = rderiv0
and TYPEVARS = undefined
defines norm = Formula-Operations.norm find0 decr0
and nFOr = Formula-Operations.nFOr :: formula ⇒ -
and nFAnd = Formula-Operations.nFAnd :: formula ⇒ -
and nFNot = Formula-Operations.nFNot find0 decr0 :: formula ⇒ -
and nFEx = Formula-Operations.nFEx find0 decr0
and nFAll = Formula-Operations.nFAll find0 decr0
and decr = Formula-Operations.decr decr0 :: - ⇒ - ⇒ formula ⇒ -
and find = Formula-Operations.find find0 :: - ⇒ - ⇒ formula ⇒ -
and FV = Formula-Operations.FV FV0
and RESTR = Formula-Operations.RESTR (λ- -. FBool True) :: - ⇒ formula
and RESTRICT = Formula-Operations.RESTR (λ- -. FBool True) FV0
and deriv = λd0 (a :: atom) (φ :: formula). Formula-Operations.deriv extend d0
a φ
and nullable = λφ :: formula. Formula-Operations.nullable nullable0 φ
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 :: nat ⇒ formula ⇒ -
and presb-wf = Formula-Operations.wf SUC (wf0 :: nat ⇒ presb ⇒ -)
and presb-lformula = Formula-Operations.lformula (lformula0 :: presb ⇒ -) ::
formula ⇒ -
and check-eqv = λidx. DAs.check-eqv
(σ idx) (λφ. norm (RESTRICT φ) :: formula)
(λa φ. norm (deriv (lderiv0 :: - ⇒ - ⇒ formula) (a :: atom) φ))
(final idx) (λφ :: formula. presb-wf idx φ ∧ presb-lformula φ)
(σ idx) (λφ. norm (RESTRICT φ) :: formula)

```

```

    (λa φ. norm (deriv (lderiv0 :: - ⇒ - ⇒ formula) (a :: atom) φ))
    (final idx) (λφ :: formula. presb-wf idx φ ∧ presb-lformula φ)
    (=)
and bounded-check-equiv = λidx. DAs.check-equiv
    (σ idx) (λφ. norm (RESTRICT φ) :: formula)
    (λa φ. norm (deriv (lderiv0 :: - ⇒ - ⇒ formula) (a :: atom) φ))
    nullable (λφ :: formula. presb-wf idx φ ∧ presb-lformula φ)
    (σ idx) (λφ. norm (RESTRICT φ) :: formula)
    (λa φ. norm (deriv (lderiv0 :: - ⇒ - ⇒ formula) (a :: atom) φ))
    nullable (λφ :: formula. presb-wf idx φ ∧ presb-lformula φ)
    (=)
and automaton = DA.automaton
    (λa φ. norm (deriv lderiv0 (a :: atom) φ) :: formula))
apply standard apply (auto simp: Presb-simps σ-def set-n-lists distinct-n-lists
    Formula-Operations.lformula.simps Formula-Operations.satisfies-gen.simps For-
    mula-Operations.wf.simps
    dest: satisfies0-cong split: presb.splits if-splits)
done

```

```

lemma check-equiv-code[code]: check-equiv idx r s =
    ((presb-wf idx r ∧ presb-lformula r) ∧ (presb-wf idx s ∧ presb-lformula s) ∧
    (case rtrancl-while (λ(p, q). final idx p = final idx q)
    (λ(p, q). map (λa. (norm (deriv lderiv0 a p), norm (deriv lderiv0 a q))) (σ idx))
    (norm (RESTRICT r), norm (RESTRICT s)) of
    None ⇒ False
    | Some ([], x) ⇒ True
    | Some (a # list, x) ⇒ False))
unfolding check-equiv-def Presb.check-equiv-def Presb.step-alt ..

```

**definition** while where [code del, code-abbrev]: while idx φ = while-default (fut-default idx φ)

**declare** while-default-code[of fut-default idx φ for idx φ, folded while-def, code]

**lemma** check-equiv-sound:

$\llbracket \#_V \mathfrak{A} = \text{idx}; \text{check-equiv idx } \varphi \ \psi \rrbracket \implies (\text{Presb.sat } \mathfrak{A} \ \varphi \longleftrightarrow \text{Presb.sat } \mathfrak{A} \ \psi)$

**unfolding** check-equiv-def **by** (rule Presb.check-equiv-soundness)

**lemma** bounded-check-equiv-sound:

$\llbracket \#_V \mathfrak{A} = \text{idx}; \text{bounded-check-equiv idx } \varphi \ \psi \rrbracket \implies (\text{Presb.sat}_b \ \mathfrak{A} \ \varphi \longleftrightarrow \text{Presb.sat}_b \ \mathfrak{A} \ \psi)$

**unfolding** bounded-check-equiv-def **by** (rule Presb.bounded-check-equiv-soundness)

**method-setup** check-equiv = <

let

fun tac ctxt =

let

val conv = @{computation-check terms: Trueprop

0 :: nat 1 :: nat 2 :: nat 3 :: nat Suc

```

    plus :: nat ⇒ - minus :: nat ⇒ -
    times :: nat ⇒ - divide :: nat ⇒ - modulo :: nat ⇒ -
    0 :: int 1 :: int 2 :: int 3 :: int -1 :: int
    check-eqv datatypes: formula int list integer bool} ctxt
  in
    CONVERSION (Conv.params-conv ~ 1 (K (Conv.concl-conv ~ 1 conv)) ctxt)
THEN'
  resolve-tac ctxt [TrueI]
end
in
  Scan.succeed (SIMPLE-METHOD' o tac)
end
>
end

```

## 7 Comparing WS1S Formulas with Presburger Formulas

**lift-definition** *letter-*eq** :: *idx* ⇒ *nat* ⇒ *bool list* × *bool list* ⇒ *bool list* ⇒ *bool is*  
 $\lambda(m1, m2) n (bs1, bs2) bs. m1 = 0 \wedge m2 = n \wedge bs1 = [] \wedge bs2 = bs .$

**lemma** *letter-*eq**[*dest*]:

*letter-*eq** *idx* *n* *a* *b* ⇒ ( *a* ∈ *set* ( *WS1S-Prelim*.σ *idx* ) ) = ( *b* ∈ *set* ( *Presburger-Formula*.σ *n* ) )

by *transfer* (force *simp*: *Presburger-Formula*.σ-def *set-n-lists* *image-iff*)

**global-interpretation** *WS1S-Presb*: *DAs*

```

  WS1S-Prelim.σ idx
  (λφ. norm (RESTRICT φ) :: (ws1s, order) aformula)
  (λa φ. norm (deriv lderiv0 (a :: atom) φ))
  (WS1S.final idx)
  (λφ :: formula. ws1s-wf idx φ ∧ ws1s-lformula φ)
  λφ. Formula.lang WS1S-Prelim.nvars
    WS1S-Prelim.Extend WS1S-Prelim.CONS WS1S-Prelim.Length WS1S-Prelim.size-atom
    WS1S-Formula.satisfies0 idx φ
  (λφ :: formula. ws1s-wf idx φ ∧ ws1s-lformula φ)
  λφ. Formula.language WS1S-Prelim.assigns
    WS1S-Prelim.nvars WS1S-Prelim.Extend WS1S-Prelim.CONS
    WS1S-Prelim.Length WS1S-Prelim.size-atom restrict WS1S-Formula.FV0
    WS1S-Formula.satisfies0 idx φ
  (Presburger-Formula.σ n)
  (λφ. Presburger-Formula.norm (Presburger-Formula.RESTRICT φ))
  (λa φ. Presburger-Formula.norm (Presburger-Formula.deriv Presburger-Formula.ldderiv0
  a φ))
  (Presburger-Formula.final n)
  (λφ. presb-wf n φ ∧ presb-lformula φ)

```

```

( $\lambda\varphi$ . Formula.lang Presburger-Formula.nvars
  Presburger-Formula.Extend Presburger-Formula.CONS Presburger-Formula.Length
  Presburger-Formula.size-atom ( $\models 0$ ) n  $\varphi$ )
( $\lambda\varphi$ . presb-wf n  $\varphi$   $\wedge$  presb-lformula  $\varphi$ )
```

*( $\lambda\varphi$ . *Formula.language Presburger-Formula.assigns**

```

  Presburger-Formula.nvars Presburger-Formula.Extend Presburger-Formula.CONS
  Presburger-Formula.Length Presburger-Formula.size-atom ( $\lambda$ - . True)
  Presburger-Formula.FV0 ( $\models 0$ )
  n  $\varphi$ )
letter-eq idx n
defines check-equiv =  $\lambda$ idx n. DAs.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-lformula  $\varphi$ )
  (Presburger-Formula. $\sigma$  n) ( $\lambda\varphi$ . Presburger-Formula.norm (Presburger-Formula.RESTRICT
 $\varphi$ ))
  ( $\lambda a \varphi$ . Presburger-Formula.norm (Presburger-Formula.deriv Presburger-Formula.lderiv0
a  $\varphi$ ))
  (Presburger-Formula.final n) ( $\lambda\varphi$ . presb-wf n  $\varphi$   $\wedge$  presb-lformula  $\varphi$ )) (letter-eq
idx n)
by unfold-locales auto
```

**lemma** *check-equiv-code[code]: check-equiv idx n r s  $\longleftrightarrow$*

```

  ((ws1s-wf idx r  $\wedge$  ws1s-lformula r)  $\wedge$  (presb-wf n s  $\wedge$  presb-lformula s)  $\wedge$ 
  (case rtrancl-while ( $\lambda(p, q)$ . final idx p = Presburger-Formula.final n q)
  ( $\lambda(p, q)$ .
    map ( $\lambda(a, b)$ . (norm (deriv lderiv0 a p),
      Presburger-Formula.norm (Presburger-Formula.deriv Presburger-Formula.lderiv0
b q)))
    [(x, y) $\leftarrow$ List.product ( $\sigma$  idx) (Presburger-Formula. $\sigma$  n). letter-eq idx n x y])
    (norm (RESTRICT r), Presburger-Formula.norm (Presburger-Formula.RESTRICT
s)) of
    None  $\Rightarrow$  False
    | Some ([], x)  $\Rightarrow$  True
    | Some (a # list, x)  $\Rightarrow$  False))
unfolding check-equiv-def WS1S-Presb.check-equiv-def ..
```

**method-setup** *check-equiv =  $\langle$*

```

  let
    fun tac ctxt =
      let
        val conv = @{computation-check terms: Trueprop
          0 :: nat 1 :: nat 2 :: nat 3 :: nat Suc
          plus :: nat  $\Rightarrow$  - minus :: nat  $\Rightarrow$  -
          times :: nat  $\Rightarrow$  - divide :: nat  $\Rightarrow$  - modulo :: nat  $\Rightarrow$  -
          0 :: int 1 :: int 2 :: int 3 :: int -1 :: int
          check-equiv datatypes: idx (presb, unit) aformula ((nat, nat) atomic,
WS1S-Prelim.order) aformula
```



```

      nat × nat nat option bool option int list integer} ctxt
    in
      CONVERSION (Conv.params-conv ~ 1 (K (Conv.concl-conv ~ 1 conv)) ctxt)
  THEN'
    resolve-tac ctxt [TrueI]
  end
in
  Scan.succeed (SIMPLE-METHOD' o tac)
end
>

```

## 8 Nameful WS1S Formulas

**declare** [[*coercion of-char* :: *char* ⇒ *nat*, *coercion-enabled*]]

**definition** *is-upper* :: *char* ⇒ *bool* **where** [*simp*]: *is-upper* *c* = (*c* ∈ {65..90 :: *nat*})

**definition** *is-lower* :: *char* ⇒ *bool* **where** [*simp*]: *is-lower* *c* = (*c* ∈ {97..122 :: *nat*})

**typedef** *fo* = {*s*. *s* ≠ [] ∧ *is-lower* (hd *s*)} **by** (*auto intro!*: *exI*[*of* - "'x'"])

**typedef** *so* = {*s*. *s* ≠ [] ∧ *is-upper* (hd *s*)} **by** (*auto intro!*: *exI*[*of* - "'X'"])

**datatype** *ws1s* =

```

  T | F | Or ws1s ws1s | And ws1s ws1s | Not ws1s
| Ex1 fo ws1s | Ex2 so ws1s | All1 fo ws1s | All2 so ws1s
| Lt fo fo
| In fo so
| Eq-Const fo nat
| Eq-Presb so nat
| Eq-FO fo fo
| Eq-FO-Offset fo fo nat
| Eq-SO so so
| Eq-SO-Inc so so
| Eq-Max fo so
| Eq-Min fo so
| Empty so
| Singleton so
| Subset so so
| Disjoint so so
| Eq-Union so so so
| Eq-Inter so so so
| Eq-Diff so so so

```

**primrec** *satisfies* :: (*fo*  $\Rightarrow$  *nat*)  $\Rightarrow$  (*so*  $\Rightarrow$  *nat fset*)  $\Rightarrow$  *ws1s*  $\Rightarrow$  *bool* **where**

*satisfies* *I1* *I2* *T* = *True*  
| *satisfies* *I1* *I2* *F* = *False*  
| *satisfies* *I1* *I2* (*Or*  $\varphi$   $\psi$ ) = (*satisfies* *I1* *I2*  $\varphi$   $\vee$  *satisfies* *I1* *I2*  $\psi$ )  
| *satisfies* *I1* *I2* (*And*  $\varphi$   $\psi$ ) = (*satisfies* *I1* *I2*  $\varphi$   $\wedge$  *satisfies* *I1* *I2*  $\psi$ )  
| *satisfies* *I1* *I2* (*Not*  $\varphi$ ) = ( $\neg$  *satisfies* *I1* *I2*  $\varphi$ )  
| *satisfies* *I1* *I2* (*Ex1*  $x$   $\varphi$ ) = ( $\exists n.$  *satisfies* (*I1* ( $x := n$ )) *I2*  $\varphi$ )  
| *satisfies* *I1* *I2* (*Ex2* *X*  $\varphi$ ) = ( $\exists N.$  *satisfies* *I1* (*I2* ( $X := N$ ))  $\varphi$ )  
| *satisfies* *I1* *I2* (*All1*  $x$   $\varphi$ ) = ( $\forall n.$  *satisfies* (*I1* ( $x := n$ )) *I2*  $\varphi$ )  
| *satisfies* *I1* *I2* (*All2* *X*  $\varphi$ ) = ( $\forall N.$  *satisfies* *I1* (*I2* ( $X := N$ ))  $\varphi$ )

| *satisfies* *I1* *I2* (*Lt*  $x$   $y$ ) = (*I1*  $x < I1$   $y$ )  
| *satisfies* *I1* *I2* (*In*  $x$  *X*) = (*I1*  $x \in I2$  *X*)  
| *satisfies* *I1* *I2* (*Eq-Const*  $x$   $n$ ) = (*I1*  $x = n$ )  
| *satisfies* *I1* *I2* (*Eq-Presb* *X*  $n$ ) = ( $(\sum x \in \text{fset } (I2 \text{ } X). 2 \wedge x) = n$ )  
| *satisfies* *I1* *I2* (*Eq-FO*  $x$   $y$ ) = (*I1*  $x = I1$   $y$ )  
| *satisfies* *I1* *I2* (*Eq-FO-Offset*  $x$   $y$   $n$ ) = (*I1*  $x = I1$   $y + n$ )  
| *satisfies* *I1* *I2* (*Eq-SO* *X* *Y*) = (*I2* *X* = *I2* *Y*)  
| *satisfies* *I1* *I2* (*Eq-SO-Inc* *X* *Y*) = (*I2* *X* = *Suc*  $| \cdot |$  *I2* *Y*)  
| *satisfies* *I1* *I2* (*Eq-Max*  $x$  *X*) = (*let* *Z* = *I2* *X* *in*  $Z \neq \{\}\} \wedge I1$   $x = \text{fMax } Z$ )  
| *satisfies* *I1* *I2* (*Eq-Min*  $x$  *X*) = (*let* *Z* = *I2* *X* *in*  $Z \neq \{\}\} \wedge I1$   $x = \text{fMin } Z$ )  
| *satisfies* *I1* *I2* (*Empty* *X*) = (*I2* *X* =  $\{\}$ )  
| *satisfies* *I1* *I2* (*Singleton* *X*) = ( $\exists x.$  *I2* *X* =  $\{x\}$ )  
| *satisfies* *I1* *I2* (*Subset* *X* *Y*) = (*I2* *X*  $\subseteq$  *I2* *Y*)  
| *satisfies* *I1* *I2* (*Disjoint* *X* *Y*) = (*I2* *X*  $\cap$  *I2* *Y* =  $\{\}$ )  
| *satisfies* *I1* *I2* (*Eq-Union* *X* *Y* *Z*) = (*I2* *X* = *I2* *Y*  $\cup$  *I2* *Z*)  
| *satisfies* *I1* *I2* (*Eq-Inter* *X* *Y* *Z*) = (*I2* *X* = *I2* *Y*  $\cap$  *I2* *Z*)  
| *satisfies* *I1* *I2* (*Eq-Diff* *X* *Y* *Z*) = (*I2* *X* = *I2* *Y*  $\setminus$  *I2* *Z*)

**primrec** *fos* **where**

*fos* *T* =  $\square$   
| *fos* *F* =  $\square$   
| *fos* (*Or*  $\varphi$   $\psi$ ) = *List.union* (*fos*  $\varphi$ ) (*fos*  $\psi$ )  
| *fos* (*And*  $\varphi$   $\psi$ ) = *List.union* (*fos*  $\varphi$ ) (*fos*  $\psi$ )  
| *fos* (*Not*  $\varphi$ ) = *fos*  $\varphi$   
| *fos* (*Ex1*  $x$   $\varphi$ ) = *List.remove1*  $x$  (*fos*  $\varphi$ )  
| *fos* (*Ex2* *X*  $\varphi$ ) = *fos*  $\varphi$   
| *fos* (*All1*  $x$   $\varphi$ ) = *List.remove1*  $x$  (*fos*  $\varphi$ )  
| *fos* (*All2* *X*  $\varphi$ ) = *fos*  $\varphi$   
| *fos* (*Lt*  $x$   $y$ ) = *remdups* [ $x$ ,  $y$ ]  
| *fos* (*In*  $x$  *X*) = [ $x$ ]  
| *fos* (*Eq-Const*  $x$   $n$ ) = [ $x$ ]  
| *fos* (*Eq-Presb* *X*  $n$ ) =  $\square$   
| *fos* (*Eq-FO*  $x$   $y$ ) = *remdups* [ $x$ ,  $y$ ]  
| *fos* (*Eq-FO-Offset*  $x$   $y$   $n$ ) = *remdups* [ $x$ ,  $y$ ]  
| *fos* (*Eq-SO* *X* *Y*) =  $\square$   
| *fos* (*Eq-SO-Inc* *X* *Y*) =  $\square$   
| *fos* (*Eq-Max*  $x$  *X*) = [ $x$ ]  
| *fos* (*Eq-Min*  $x$  *X*) = [ $x$ ]

$| \text{fos } (\text{Empty } X) = []$   
 $| \text{fos } (\text{Singleton } X) = []$   
 $| \text{fos } (\text{Subset } X Y) = []$   
 $| \text{fos } (\text{Disjoint } X Y) = []$   
 $| \text{fos } (\text{Eq-Union } X Y Z) = []$   
 $| \text{fos } (\text{Eq-Inter } X Y Z) = []$   
 $| \text{fos } (\text{Eq-Diff } X Y Z) = []$

**primrec** *sos* where

$| \text{sos } T = []$   
 $| \text{sos } F = []$   
 $| \text{sos } (\text{Or } \varphi \psi) = \text{List.union } (\text{sos } \varphi) (\text{sos } \psi)$   
 $| \text{sos } (\text{And } \varphi \psi) = \text{List.union } (\text{sos } \varphi) (\text{sos } \psi)$   
 $| \text{sos } (\text{Not } \varphi) = \text{sos } \varphi$   
 $| \text{sos } (\text{Ex1 } x \varphi) = \text{sos } \varphi$   
 $| \text{sos } (\text{Ex2 } X \varphi) = \text{List.remove1 } X (\text{sos } \varphi)$   
 $| \text{sos } (\text{All1 } x \varphi) = \text{sos } \varphi$   
 $| \text{sos } (\text{All2 } X \varphi) = \text{List.remove1 } X (\text{sos } \varphi)$   
 $| \text{sos } (\text{Lt } x y) = []$   
 $| \text{sos } (\text{In } x X) = [X]$   
 $| \text{sos } (\text{Eq-Const } x n) = []$   
 $| \text{sos } (\text{Eq-Presb } X n) = [X]$   
 $| \text{sos } (\text{Eq-FO } x y) = []$   
 $| \text{sos } (\text{Eq-FO-Offset } x y n) = []$   
 $| \text{sos } (\text{Eq-SO } X Y) = \text{remdups } [X, Y]$   
 $| \text{sos } (\text{Eq-SO-Inc } X Y) = \text{remdups } [X, Y]$   
 $| \text{sos } (\text{Eq-Max } x X) = [X]$   
 $| \text{sos } (\text{Eq-Min } x X) = [X]$   
 $| \text{sos } (\text{Empty } X) = [X]$   
 $| \text{sos } (\text{Singleton } X) = [X]$   
 $| \text{sos } (\text{Subset } X Y) = \text{remdups } [X, Y]$   
 $| \text{sos } (\text{Disjoint } X Y) = \text{remdups } [X, Y]$   
 $| \text{sos } (\text{Eq-Union } X Y Z) = \text{remdups } [X, Y, Z]$   
 $| \text{sos } (\text{Eq-Inter } X Y Z) = \text{remdups } [X, Y, Z]$   
 $| \text{sos } (\text{Eq-Diff } X Y Z) = \text{remdups } [X, Y, Z]$

**lemma** *distinct-fos[simp]*: *distinct* (*fos*  $\varphi$ ) **by** (*induct*  $\varphi$ ) *auto*

**lemma** *distinct-sos[simp]*: *distinct* (*sos*  $\varphi$ ) **by** (*induct*  $\varphi$ ) *auto*

**primrec**  $\varepsilon$  where

$| \varepsilon \text{ bs1 bs2 } T = \text{FBool True}$   
 $| \varepsilon \text{ bs1 bs2 } F = \text{FBool False}$   
 $| \varepsilon \text{ bs1 bs2 } (\text{Or } \varphi \psi) = \text{FOr } (\varepsilon \text{ bs1 bs2 } \varphi) (\varepsilon \text{ bs1 bs2 } \psi)$   
 $| \varepsilon \text{ bs1 bs2 } (\text{And } \varphi \psi) = \text{FAnd } (\varepsilon \text{ bs1 bs2 } \varphi) (\varepsilon \text{ bs1 bs2 } \psi)$   
 $| \varepsilon \text{ bs1 bs2 } (\text{Not } \varphi) = \text{FNot } (\varepsilon \text{ bs1 bs2 } \varphi)$   
 $| \varepsilon \text{ bs1 bs2 } (\text{Ex1 } x \varphi) = \text{FEx FO } (\varepsilon (x \# \text{bs1}) \text{bs2 } \varphi)$   
 $| \varepsilon \text{ bs1 bs2 } (\text{Ex2 } X \varphi) = \text{FEx SO } (\varepsilon \text{ bs1 } (X \# \text{bs2}) \varphi)$   
 $| \varepsilon \text{ bs1 bs2 } (\text{All1 } x \varphi) = \text{FAll FO } (\varepsilon (x \# \text{bs1}) \text{bs2 } \varphi)$   
 $| \varepsilon \text{ bs1 bs2 } (\text{All2 } X \varphi) = \text{FAll SO } (\varepsilon \text{ bs1 } (X \# \text{bs2}) \varphi)$

$| \varepsilon \text{ bs1 bs2 } (Lt \ x \ y) = FBase \ (Less \ None \ (index \ bs1 \ x) \ (index \ bs1 \ y))$   
 $| \varepsilon \text{ bs1 bs2 } (In \ x \ X) = FBase \ (WS1S-Formula.In \ False \ (index \ bs1 \ x) \ (index \ bs2 \ X))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-Const \ x \ n) = FBase \ (WS1S-Formula.Eq-Const \ None \ (index \ bs1 \ x) \ n)$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-Presb \ X \ n) = FBase \ (WS1S-Formula.Eq-Presb \ None \ (index \ bs2 \ X) \ n)$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-FO \ x \ y) = FBase \ (WS1S-Formula.Eq-FO \ False \ (index \ bs1 \ x) \ (index \ bs1 \ y))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-FO-Offset \ x \ y \ n) = FBase \ (WS1S-Formula.Plus-FO \ None \ (index \ bs1 \ x) \ (index \ bs1 \ y) \ n)$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-SO \ X \ Y) = FBase \ (WS1S-Formula.Eq-SO \ (index \ bs2 \ X) \ (index \ bs2 \ Y))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-SO-Inc \ X \ Y) = FBase \ (WS1S-Formula.Suc-SO \ False \ False \ (index \ bs2 \ X) \ (index \ bs2 \ Y))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-Max \ x \ X) = FBase \ (WS1S-Formula.Eq-Max \ False \ (index \ bs1 \ x) \ (index \ bs2 \ X))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-Min \ x \ X) = FBase \ (WS1S-Formula.Eq-Min \ False \ (index \ bs1 \ x) \ (index \ bs2 \ X))$   
 $| \varepsilon \text{ bs1 bs2 } (Empty \ X) = FBase \ (WS1S-Formula.Empty \ (index \ bs2 \ X))$   
 $| \varepsilon \text{ bs1 bs2 } (Singleton \ X) = FBase \ (WS1S-Formula.Singleton \ (index \ bs2 \ X))$   
 $| \varepsilon \text{ bs1 bs2 } (Subset \ X \ Y) = FBase \ (WS1S-Formula.Subset \ (index \ bs2 \ X) \ (index \ bs2 \ Y))$   
 $| \varepsilon \text{ bs1 bs2 } (Disjoint \ X \ Y) = FBase \ (WS1S-Formula.Disjoint \ (index \ bs2 \ X) \ (index \ bs2 \ Y))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-Union \ X \ Y \ Z) = FBase \ (WS1S-Formula.Eq-Union \ (index \ bs2 \ X) \ (index \ bs2 \ Y) \ (index \ bs2 \ Z))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-Inter \ X \ Y \ Z) = FBase \ (WS1S-Formula.Eq-Inter \ (index \ bs2 \ X) \ (index \ bs2 \ Y) \ (index \ bs2 \ Z))$   
 $| \varepsilon \text{ bs1 bs2 } (Eq-Diff \ X \ Y \ Z) = FBase \ (WS1S-Formula.Eq-Diff \ (index \ bs2 \ X) \ (index \ bs2 \ Y) \ (index \ bs2 \ Z))$

**lift-definition** *mk-I* ::

$(fo \Rightarrow nat) \Rightarrow (so \Rightarrow nat \ fset) \Rightarrow fo \ list \Rightarrow so \ list \Rightarrow interp \ \mathbf{is}$   
 $\lambda I1 \ I2 \ fs \ ss. \ let \ I1s = map \ (\lambda x. \ \{\{I1 \ x\}\}) \ fs; \ I2s = map \ I2 \ ss \ in \ (MSB \ (I1s \ @ \ I2s), \ I1s, \ I2s)$   
**by** (*auto simp: Let-def*)

**definition** *dec-I1* ::  $interp \Rightarrow fo \ list \Rightarrow fo \Rightarrow nat$  **where** *dec-I1*  $\mathfrak{A} \ fs \ x = fMin \ (index \ fs \ x \ \mathfrak{A} \ FO)$

**definition** *dec-I2* ::  $interp \Rightarrow so \ list \Rightarrow so \Rightarrow nat \ fset$  **where** *dec-I2*  $\mathfrak{A} \ ss \ X = index \ ss \ X \ \mathfrak{A} \ SO$

**lemma** *nvars-mk-I[simp]*:  $\#_V \ (mk-I \ I1 \ I2 \ fs \ ss) = Abs-idx \ (length \ fs, \ length \ ss)$   
**by** *transfer (auto simp: Let-def)*

**lemma** *assigns-mk-I-FO[simp]*:

$m^{mk-I \ I1 \ I2 \ bs1 \ bs2 \ FO} = (if \ m < length \ bs1 \ then \ \{\{I1 \ (bs1 \ ! \ m)\}\} \ else \ \{\{\}\})$   
**by** *transfer (auto simp: Let-def)*

**lemma** *assigns-mk-I-SO[simp]*:

$m^{mk-I} I1 I2 bs1 bs2 SO = (if\ m < length\ bs2\ then\ I2\ (bs2\ !\ m)\ else\ \{\})$   
**by** *transfer (auto simp: Let-def)*

**lemma** *satisfies-cong*:

$\llbracket \forall x \in set\ (fos\ \varphi). I1\ x = J1\ x; \forall X \in set\ (sos\ \varphi). I2\ X = J2\ X \rrbracket \implies$   
*satisfies*  $I1\ I2\ \varphi \longleftrightarrow$  *satisfies*  $J1\ J2\ \varphi$

**proof** (*induct*  $\varphi$  *arbitrary*:  $I1\ I2\ J1\ J2$ )

**case** ( $Or\ \varphi\ \psi$ ) **from**  $Or.hyps[of\ I1\ J1\ I2\ J2]$   $Or.prem$ s **show** *?case* **by** *auto*

**next**

**case** ( $And\ \varphi\ \psi$ ) **from**  $And.hyps[of\ I1\ J1\ I2\ J2]$   $And.prem$ s **show** *?case* **by** *auto*

**next**

**case** ( $Ex1\ x\ \varphi$ ) **with**  $Ex1.hyps[of\ I1\ (x := y)\ J1\ (x := y)\ I2\ J2\ for\ y,\ cong]$  **show** *?case* **by** *auto*

**next**

**case** ( $Ex2\ X\ \varphi$ ) **with**  $Ex2.hyps[of\ I1\ J1\ I2(X := Y)\ J2(X := Y)\ for\ Y,\ cong]$  **show** *?case* **by** *auto*

**next**

**case** ( $All1\ x\ \varphi$ ) **with**  $All1.hyps[of\ I1\ (x := y)\ J1\ (x := y)\ I2\ J2\ for\ y,\ cong]$  **show** *?case* **by** *auto*

**next**

**case** ( $All2\ X\ \varphi$ ) **with**  $All2.hyps[of\ I1\ J1\ I2(X := Y)\ J2(X := Y)\ for\ Y,\ cong]$  **show** *?case* **by** *auto*

**qed** *simp-all*

**lemma** *dec-I-mk-I-satisfies-cong*:

$\llbracket set\ (fos\ \varphi) \subseteq set\ bs1; set\ (sos\ \varphi) \subseteq set\ bs2; \mathfrak{A} = mk-I\ I1\ I2\ bs1\ bs2 \rrbracket \implies$   
*satisfies* ( $dec-I1\ \mathfrak{A}\ bs1$ ) ( $dec-I2\ \mathfrak{A}\ bs2$ )  $\varphi \longleftrightarrow$  *satisfies*  $I1\ I2\ \varphi$   
**by** (*rule* *satisfies-cong*) (*auto* *simp*:  $dec-I1-def\ dec-I2-def$ )

**definition** *ok-I*  $\mathfrak{A}\ fs = (\forall x \in set\ fs.\ index\ fs\ x^{\mathfrak{A}} FO \neq \{\})$

**lemma** *ok-I-mk-I[simp]*: *ok-I* ( $mk-I\ I1\ I2\ bs1\ bs2$ )  $bs1$

**unfolding** *ok-I-def* **by** *transfer (auto simp: Let-def)*

**lemma** *in-FV-εD[dest]*:  $\llbracket v \in FV\ (\varepsilon\ bs1\ bs2\ \varphi)\ FO;$

$set\ (fos\ \varphi) \subseteq set\ bs1; set\ (sos\ \varphi) \subseteq set\ bs2 \rrbracket \implies$

$(\exists y \in set\ bs1.\ v = index\ bs1\ y)$

**proof** (*induct*  $\varphi$  *arbitrary*:  $bs1\ bs2\ v$ )

**case** ( $Ex1\ x\ \varphi$ ) **from**  $Ex1.hyps[of\ Suc\ v\ x\ \# bs1\ bs2]$   $Ex1.prem$ s **show** *?case*  
**by** (*auto* *simp*: *Diff-subset-conv split: if-splits*)

**next**

**case** ( $Ex2\ X\ \varphi$ ) **from**  $Ex2.hyps[of\ v\ bs1\ X\ \# bs2]$   $Ex2.prem$ s **show** *?case*  
**by** (*auto* *simp*: *Diff-subset-conv split: if-splits*)

**next**

**case** ( $All1\ x\ \varphi$ ) **from**  $All1.hyps[of\ Suc\ v\ x\ \# bs1\ bs2]$   $All1.prem$ s **show** *?case*  
**by** (*auto* *simp*: *Diff-subset-conv split: if-splits*)

**next**

```

    case (All2 X  $\varphi$ ) from All2.hyps[of v bs1 X # bs2] All2.premis show ?case
      by (auto simp: Diff-subset-conv split: if-splits)
qed (auto split: if-splits)

lemma dec-I1-Extend-FO[simp]:
  dec-I1 (Extend FO 0  $\mathfrak{A}$  P) (x # bs1) = (dec-I1  $\mathfrak{A}$  bs1)(x := fMin P)
  by (auto simp: dec-I1-def)

lemma dec-I1-Extend-SO[simp]: dec-I1 (Extend SO i  $\mathfrak{A}$  P) bs1 = dec-I1  $\mathfrak{A}$  bs1
  by (auto simp: dec-I1-def)

lemma dec-I2-Extend-FO[simp]: dec-I2 (Extend FO i  $\mathfrak{A}$  P) bs2 = dec-I2  $\mathfrak{A}$  bs2
  by (auto simp: dec-I2-def)

lemma dec-I2-Extend-SO[simp]:
  dec-I2 (Extend SO 0  $\mathfrak{A}$  P) (X # bs2) = (dec-I2  $\mathfrak{A}$  bs2)(X := P)
  by (auto simp: dec-I2-def fun-eq-iff)

lemma sat- $\varepsilon$ :  $\llbracket \text{set } (fos \ \varphi) \subseteq \text{set } bs1; \text{set } (sos \ \varphi) \subseteq \text{set } bs2; ok-I \ \mathfrak{A} \ bs1 \rrbracket \implies$ 
  WS1S.sat  $\mathfrak{A}$  ( $\varepsilon$  bs1 bs2  $\varphi$ )  $\longleftrightarrow$ 
  satisfies (dec-I1  $\mathfrak{A}$  bs1) (dec-I2  $\mathfrak{A}$  bs2)  $\varphi$ 
proof (induct  $\varphi$  arbitrary:  $\mathfrak{A}$  bs1 bs2)
  case (Or  $\varphi$   $\psi$ ) from Or.hyps[of bs1 bs2  $\mathfrak{A}$ ] Or.premis show ?case
    unfolding WS1S.sat-def
    by (auto simp: restrict-def ok-I-def split: order.splits)
  next
  case (And  $\varphi$   $\psi$ ) from And.hyps[of bs1 bs2  $\mathfrak{A}$ ] And.premis show ?case
    unfolding WS1S.sat-def
    by (auto simp: restrict-def ok-I-def split: order.splits)
  next
  case (Not  $\varphi$ ) from Not.hyps[of bs1 bs2  $\mathfrak{A}$ ] Not.premis show ?case
    unfolding WS1S.sat-def
    by (auto simp: restrict-def ok-I-def split: order.splits)
  next
  case (Ex1 x  $\varphi$ )
    { fix P :: nat fset assume P  $\neq$   $\{\}\}$ 
      with Ex1.premis have WS1S.sat (Extend FO 0  $\mathfrak{A}$  P) ( $\varepsilon$  (x # bs1) bs2  $\varphi$ )  $\longleftrightarrow$ 
        satisfies (dec-I1 (Extend FO 0  $\mathfrak{A}$  P) (x # bs1)) (dec-I2 (Extend FO 0  $\mathfrak{A}$  P)
bs2)  $\varphi$ 
      by (intro Ex1.hyps) (auto simp: ok-I-def)
    } note IH = this
  show ?case
proof
  assume WS1S.sat  $\mathfrak{A}$  ( $\varepsilon$  bs1 bs2 (Ex1 x  $\varphi$ ))
  then obtain P where P  $\neq$   $\{\}\}$  WS1S.sat (Extend FO 0  $\mathfrak{A}$  P) ( $\varepsilon$  (x # bs1)
bs2)  $\varphi$ 
  unfolding WS1S.sat-def by (auto simp: restrict-def split: order.splits)
  then have satisfies (dec-I1 (Extend FO 0  $\mathfrak{A}$  P) (x # bs1)) (dec-I2 (Extend
FO 0  $\mathfrak{A}$  P) bs2)  $\varphi$ 

```

```

    by (auto simp: IH)
  then show satisfies (dec-I1  $\mathfrak{A}$  bs1) (dec-I2  $\mathfrak{A}$  bs2) (Ex1 x  $\varphi$ ) by auto
next
  assume satisfies (dec-I1  $\mathfrak{A}$  bs1) (dec-I2  $\mathfrak{A}$  bs2) (Ex1 x  $\varphi$ )
  then obtain n where satisfies (dec-I1 (Extend FO 0  $\mathfrak{A}$   $\{|n|\}$ ) (x # bs1))
    (dec-I2 (Extend FO 0  $\mathfrak{A}$   $\{|n|\}$ ) bs2)  $\varphi$  by auto
  then have WS1S.sat (Extend FO 0  $\mathfrak{A}$   $\{|n|\}$ ) ( $\varepsilon$  (x # bs1) bs2  $\varphi$ )
    by (auto simp: IH)
  then show WS1S.sat  $\mathfrak{A}$  ( $\varepsilon$  bs1 bs2 (Ex1 x  $\varphi$ )) unfolding WS1S.sat-def
    by (auto simp: restrict-def split: order.splits)
qed
next
  case (Ex2 X  $\varphi$ )
  { fix P :: nat fset
    from Ex2.prem1 have WS1S.sat (Extend SO 0  $\mathfrak{A}$  P) ( $\varepsilon$  bs1 (X # bs2)  $\varphi$ )  $\longleftrightarrow$ 
      satisfies (dec-I1 (Extend SO 0  $\mathfrak{A}$  P) bs1) (dec-I2 (Extend SO 0  $\mathfrak{A}$  P) (X #
bs2))  $\varphi$ 
    by (intro Ex2.hyps) (auto simp: ok-I-def)
  } note IH = this
  show ?case
  proof -
    have WS1S.sat  $\mathfrak{A}$  ( $\varepsilon$  bs1 bs2 (Ex2 X  $\varphi$ ))  $\longleftrightarrow$ 
      ( $\exists P$ . WS1S.sat (Extend SO 0  $\mathfrak{A}$  P) ( $\varepsilon$  bs1 (X # bs2)  $\varphi$ ))
    unfolding WS1S.sat-def by (auto simp: restrict-def split: order.splits)
    also have ...  $\longleftrightarrow$  ( $\exists P$ . satisfies (dec-I1 (Extend SO 0  $\mathfrak{A}$  P) bs1)
      (dec-I2 (Extend SO 0  $\mathfrak{A}$  P) (X # bs2))  $\varphi$ ) by (auto simp: IH)
    also have ...  $\longleftrightarrow$  satisfies (dec-I1  $\mathfrak{A}$  bs1) (dec-I2  $\mathfrak{A}$  bs2) (Ex2 X  $\varphi$ ) by simp
    finally show ?thesis .
  qed
next
  case (All1 x  $\varphi$ )
  { fix P :: nat fset assume P  $\neq$   $\{|\}$ 
    with All1.prem1 have WS1S.sat (Extend FO 0  $\mathfrak{A}$  P) ( $\varepsilon$  (x # bs1) bs2  $\varphi$ )  $\longleftrightarrow$ 
      satisfies (dec-I1 (Extend FO 0  $\mathfrak{A}$  P) (x # bs1)) (dec-I2 (Extend FO 0  $\mathfrak{A}$  P)
bs2)  $\varphi$ 
    by (intro All1.hyps) (auto simp: ok-I-def)
  } note IH = this
  show ?case
  proof
    assume L: WS1S.sat  $\mathfrak{A}$  ( $\varepsilon$  bs1 bs2 (All1 x  $\varphi$ ))
    { fix n :: nat
      from L have WS1S.sat (Extend FO 0  $\mathfrak{A}$   $\{|n|\}$ ) ( $\varepsilon$  (x # bs1) bs2  $\varphi$ )
        unfolding WS1S.sat-def by (auto simp: restrict-def split: order.splits)
      then have satisfies (dec-I1 (Extend FO 0  $\mathfrak{A}$   $\{|n|\}$ ) (x # bs1))
        (dec-I2 (Extend FO 0  $\mathfrak{A}$   $\{|n|\}$ ) bs2)  $\varphi$  by (auto simp: IH)
    }
    then show satisfies (dec-I1  $\mathfrak{A}$  bs1) (dec-I2  $\mathfrak{A}$  bs2) (All1 x  $\varphi$ ) by simp
  next
    assume R: satisfies (dec-I1  $\mathfrak{A}$  bs1) (dec-I2  $\mathfrak{A}$  bs2) (All1 x  $\varphi$ )

```

```

{ fix P :: nat fset assume P ≠ {}
  with R have satisfies (dec-I1 (Extend FO 0  $\mathfrak{A}$  P) (x # bs1))
    (dec-I2 (Extend FO 0  $\mathfrak{A}$  P) bs2)  $\varphi$  by auto
  with  $\langle P \neq \{\} \rangle$  have WS1S.sat (Extend FO 0  $\mathfrak{A}$  P) ( $\varepsilon$  (x # bs1) bs2  $\varphi$ )
    by (auto simp: IH)
}
with All1.prems in-FV- $\varepsilon$ D[of - x # bs1 bs2  $\varphi$ ]
  show WS1S.sat  $\mathfrak{A}$  ( $\varepsilon$  bs1 bs2 (All1 x  $\varphi$ )) unfolding WS1S.sat-def
  by (auto 0 3 simp: restrict-def ok-I-def split: if-splits order.splits)
qed
next
case (All2 X  $\varphi$ )
{ fix P :: nat fset
  from All2.prems have WS1S.sat (Extend SO 0  $\mathfrak{A}$  P) ( $\varepsilon$  bs1 (X # bs2)  $\varphi$ )
 $\longleftrightarrow$ 
  satisfies (dec-I1 (Extend SO 0  $\mathfrak{A}$  P) bs1) (dec-I2 (Extend SO 0  $\mathfrak{A}$  P) (X #
bs2))  $\varphi$ 
  by (intro All2.hyps) (auto simp: ok-I-def)
} note IH = this
show ?case
proof -
  have WS1S.sat  $\mathfrak{A}$  ( $\varepsilon$  bs1 bs2 (All2 X  $\varphi$ ))  $\longleftrightarrow$ 
    ( $\forall P$ . WS1S.sat (Extend SO 0  $\mathfrak{A}$  P) ( $\varepsilon$  bs1 (X # bs2)  $\varphi$ ))
  unfolding WS1S.sat-def by (auto simp: restrict-def split: order.splits)
  also have ...  $\longleftrightarrow$  ( $\forall P$ . satisfies (dec-I1 (Extend SO 0  $\mathfrak{A}$  P) bs1)
    (dec-I2 (Extend SO 0  $\mathfrak{A}$  P) (X # bs2))  $\varphi$ ) by (auto simp: IH)
  also have ...  $\longleftrightarrow$  satisfies (dec-I1  $\mathfrak{A}$  bs1) (dec-I2  $\mathfrak{A}$  bs2) (All2 X  $\varphi$ ) by simp
  finally show ?thesis .
qed
qed (auto simp: WS1S.sat-def Let-def dec-I1-def dec-I2-def ok-I-def
  restrict-def split: if-splits order.splits)

```

**definition**  $eqv \varphi \psi =$   
 (let fs = List.union (fos  $\varphi$ ) (fos  $\psi$ ); ss = List.union (sos  $\varphi$ ) (sos  $\psi$ )  
 in check- $eqv$  (Abs-idx (length fs, length ss)) ( $\varepsilon$  fs ss  $\varphi$ ) ( $\varepsilon$  fs ss  $\psi$ ))

**lemma**  $eqv$ -sound:  $eqv \varphi \psi \implies$  satisfies I1 I2  $\varphi \longleftrightarrow$  satisfies I1 I2  $\psi$   
**unfolding**  $eqv$ -def Let-def  
**by** (drule check- $eqv$ -sound[rotated, of - - -  
mk-I I1 I2 (List.union (fos  $\varphi$ ) (fos  $\psi$ )) (List.union (sos  $\varphi$ ) (sos  $\psi$ ))])  
(auto simp: sat- $\varepsilon$  dec-I-mk-I-satisfies-cong)

**definition**  $Thm \varphi = eqv \varphi T$

**lemma**  $Thm \Phi \implies$  satisfies I1 I2  $\Phi$   
**unfolding**  $Thm$ -def **by** (drule  $eqv$ -sound[of - - I1 I2]) simp

**setup-lifting** type-definition-fo  
**setup-lifting** type-definition-so



```
instantiation fo :: equal
begin
  lift-definition equal-fo :: fo  $\Rightarrow$  fo  $\Rightarrow$  bool is (=) .
  instance by (standard, transfer) simp
end
```

```
instantiation so :: equal
begin
  lift-definition equal-so :: so  $\Rightarrow$  so  $\Rightarrow$  bool is (=) .
  instance by (standard, transfer) simp
end
```