Derivatives of Logical Formulas

Dmitriy Traytel

March 17, 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* [1] by the author.

Contents

1 Equivalence Framework **1**
- 1.1 Abstract Deterministic Automaton 4
- 1.2 The overall procedure 6
- 1.3 Abstract Deterministic Finite Automaton 7

2 Derivatives of Abstract Formulas **9**
- 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

3 WS1S Interpretations **36**

[1] 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**
$[\![\mathfrak{o}\ L = \mathfrak{o}\ K; \bigwedge a\ b.\ R\ a\ b \Longrightarrow rel\text{-}language\ R\ (\mathfrak{d}\ L\ a)\ (\mathfrak{d}\ K\ b)]\!] \Longrightarrow rel\text{-}language\ R\ L\ K$

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

lemma *rel-language-alt*:
$rel\text{-}language\ R\ L\ K = rel\text{-}fun\ (list\text{-}all2\ R)\ (=)\ (in\text{-}language\ L)\ (in\text{-}language\ K)$
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 $in\text{-}language\ L\ xs = in\text{-}language\ K\ ys$
by (*induct xs ys arbitrary*: *L K*) (*auto del*: *iffI elim*: *rel-language.cases*)
next
assume $\forall xs\ ys.\ list\text{-}all2\ R\ xs\ ys \longrightarrow in\text{-}language\ L\ xs = in\text{-}language\ K\ ys$
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*)+
(*auto intro*: *box-equals*[*OF - to-language-in-language to-language-in-language*])

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

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

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

lemma *in-language-to-language*[*simp*]: $in\text{-}language\ (to\text{-}language\ L)\ w \longleftrightarrow w \in L$

by (*metis in-language-to-language mem-Collect-eq*)

lemma *rtrancl-fold-product*:
shows $\{((r, s), (f\ a\ r, g\ b\ s)) \mid a\ b\ r\ s.\ a \in A \wedge b \in B \wedge R\ a\ b\}\widehat{\ }* =$
$\{((r, s), (\mathit{fold}\ f\ w1\ r, \mathit{fold}\ g\ w2\ s)) \mid w1\ w2\ r\ s.\ w1 \in \mathit{lists}\ A \wedge w2 \in \mathit{lists}\ B \wedge \mathit{list\text{-}all2}\ R\ w1\ w2\}$
(**is** $?L = ?R$)
proof−
{ fix $r\ s\ r'\ s'$
have $((r, s), (r', s')) : ?L \Longrightarrow ((r, s), (r', s')) \in ?R$
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 $\bigwedge x.\ x \in ?L \Longrightarrow x \in ?R$ **by** *force*
moreover
{ fix $r\ s\ r'\ s'$
{ fix *w1 w2* **assume** *list-all2 R w1 w2* $w1 \in \mathit{lists}\ A$ $w2 \in \mathit{lists}\ B$
then have $((r, s), \mathit{fold}\ f\ w1\ r, \mathit{fold}\ g\ w2\ s) \in ?L$
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 $((r, s), (r', s')) \in ?R \Longrightarrow ((r, s), (r', s')) \in ?L$ **by** *auto*
}
hence $\bigwedge x.\ x \in ?R \Longrightarrow x \in ?L$ **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

lemma *rtrancl-fold-product1*:
shows $\{(r, s).\ \exists a \in A.\ s = f\ a\ r\}\widehat{\ }* = \{(r, s).\ \exists a \in \mathit{lists}\ A.\ s = \mathit{fold}\ f\ a\ r\}$ (**is** $?L = ?R$)
proof−
{ fix $r\ s$
have $(r, s) \in ?L \Longrightarrow (r, s) \in ?R$
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** $w \in \mathit{lists}\ A$
then have $(r, \mathit{fold}\ f\ w\ r) \in ?L$
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* $\Longrightarrow (r, s) \in$ *?L* **by** *auto*
 } **ultimately show** *?thesis* **by** (*auto 10 0*)
qed

lemma *lang-eq-ext-Nil-fold-Deriv*:
 fixes $K\ L\ A\ R$
 assumes
  $\bigwedge w.$ *in-language* $K\ w \Longrightarrow w \in$ *lists* A
  $\bigwedge w.$ *in-language* $L\ w \Longrightarrow w \in$ *lists* B
  $\bigwedge a\ b.\ R\ a\ b \Longrightarrow a \in A \longleftrightarrow b \in B$
 defines $\mathfrak{B} \equiv \{(\mathfrak{d}s\ w1\ K, \mathfrak{d}s\ w2\ L) \mid w1\ w2.\ w1 \in lists\ A \wedge w2 \in lists\ B \wedge list\text{-}all2\ R\ w1\ w2\}$
 shows *rel-language* $R\ K\ L \longleftrightarrow (\forall (K, L) \in \mathfrak{B}.\ \mathfrak{o}\ K \longleftrightarrow \mathfrak{o}\ L)$
proof
 assume $\forall (K, L)\in\mathfrak{B}.\ \mathfrak{o}\ K = \mathfrak{o}\ L$
 then show *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*: $\bigwedge K'\ L'.\ \exists w1\ w2.$
   $K' = \mathfrak{d}s\ w1\ K \wedge L' = \mathfrak{d}s\ w2\ L \wedge w1 \in lists\ A \wedge w2 \in lists\ B \wedge list\text{-}all2\ R\ w1\ w2 \Longrightarrow \mathfrak{o}\ K' = \mathfrak{o}\ L'$ **and**
   [*dest*]: $\bigwedge w.$ *in-language* $K\ w \Longrightarrow w \in$ *lists* A $\bigwedge w.$ *in-language* $L\ w \Longrightarrow w \in$ *lists* B
   by *blast+*
  show *?case* **unfolding** *ex-simps simp-thms*
  proof (*safe del*: *iffI*)
   show $\mathfrak{o}\ K = \mathfrak{o}\ L$ **by** (*intro CIH*[*OF exI*[**where** $x = []$]]) *simp*
  next
   fix $x\ y\ w1\ w2$ **assume** $\forall x\in set\ w1.\ x \in A$ $\forall x\in set\ w2.\ x \in B$ *list-all2* $R\ w1\ w2$ $R\ x\ y$
   then show $\mathfrak{o}\ (\mathfrak{d}s\ w1\ (\mathfrak{d}\ K\ x)) = \mathfrak{o}\ (\mathfrak{d}s\ w2\ (\mathfrak{d}\ L\ y))$
   proof (*cases* $x \in A \wedge y \in B$)
    assume $\neg\ (x \in A \wedge y \in B)$
    with *assms*(*3*)[*OF* ‹$R\ x\ y$›] **show** *?thesis*
      by (*auto simp*: *in-language-*$\mathfrak{d}$*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* $\mathfrak{o}$-$\mathfrak{d}$*s*)

1.1 Abstract Deterministic Automaton

locale *DA* =
fixes *alphabet* :: *'a list*
fixes *init* :: *'t* ⇒ *'s*
fixes *delta* :: *'a* ⇒ *'s* ⇒ *'s*
fixes *accept* :: *'s* ⇒ *bool*
fixes *wellformed* :: *'s* ⇒ *bool*
fixes *Language* :: *'s* ⇒ *'a language*
fixes *wf* :: *'t* ⇒ *bool*
fixes *Lang* :: *'t* ⇒ *'a language*
assumes *distinct-alphabet*: *distinct alphabet*
assumes *Language-init*: *wf t* ⟹ *Language* (*init t*) = *Lang t*
assumes *wellformed-init*: *wf t* ⟹ *wellformed* (*init t*)
assumes *Language-alphabet*: ⟦*wellformed s*; *in-language* (*Language s*) *w*⟧ ⟹ *w* ∈ *lists* (*set alphabet*)
assumes *wellformed-delta*: ⟦*wellformed s*; *a* ∈ *set alphabet*⟧ ⟹ *wellformed* (*delta a s*)
assumes *Language-delta*: ⟦*wellformed s*; *a* ∈ *set alphabet*⟧ ⟹ *Language* (*delta a s*) = $\mathfrak{d}$ (*Language s*) *a*
assumes *accept-Language*: *wellformed s* ⟹ *accept s* ⟷ $\mathfrak{o}$ (*Language s*)
begin

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

lemma *wellformed-deltas*:
⟦*wellformed s*; *w* ∈ *lists* (*set alphabet*)⟧ ⟹ *wellformed* (*fold delta w s*)
by (*induction w arbitrary*: *s*) (*auto simp add*: *Language-delta wellformed-delta*)

lemma *Language-deltas*:
⟦*wellformed s*; *w* ∈ *lists* (*set alphabet*)⟧ ⟹ *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* ⇒ *'s* ⇒ *'s set* **where**
reachable as s = *snd* (*the* (*rtrancl-while* (λ-. *True*) (λ*s*. *map* (λ*a*. *delta a s*) *as*) *s*))

definition *automaton* :: *'a list* ⇒ *'s* ⇒ ((*'s* ∗ *'a*) ∗ *'s*) *set* **where**
automaton as s =
snd (*the*
(*let start* = (([*s*], {*s*}), {});
test = λ((*ws*, *Z*), *A*). *ws* ≠ [];
step = λ((*ws*, *Z*), *A*).
(*let s* = *hd ws*;
new-edges = *map* (λ*a*. ((*s*, *a*), *delta a s*)) *as*;
new = *remdups* (*filter* (λ*ss*. *ss* ∉ *Z*) (*map snd new-edges*))
in ((*new* @ *tl ws*, *set new* ∪ *Z*), *set new-edges* ∪ *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-𝔇*s* **..**

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* = 𝔇*s w1* (*Language1 ?r*) ∧ *b* = 𝔇*s 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*), (*delta1 a r*, *delta2 b s*)) | *a b r s*.
a $\in$ *set alphabet1* $\wedge$ *b* $\in$ *set alphabet2* $\wedge$ *letter-eq a b*}^* ‘‘ {(*?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* $\in$ *set* (*step x*)} =
{((*r*, *s*), (*delta1 a r*, *delta2 b s*)) | *a b r s*. *a* $\in$ *set alphabet1* $\wedge$ *b* $\in$ *set alphabet2* $\wedge$ *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* $\in$ *set alphabet*. *s* = *delta a r*}^* ‘‘ {*r*})
unfolding *rtrancl-fold-product1* **by** (*auto intro*: *finite-subset*[*OF - fin*])

lemma
assumes *wellformed r set as* $\subseteq$ *set alphabet*
shows *reachable*: *reachable as r* = {*fold delta w r* | *w*. *w* $\in$ *lists* (*set as*)}
and *finite-reachable*: *finite* (*reachable as r*)
proof −
obtain *wsZ* **where** ∗: *rtrancl-while* (λ-. *True*) (λ*s*. *map* (λ*a*. *delta a s*) *as*) *r* = *Some wsZ*
using *assms* **by** (*atomize-elim*, *intro rtrancl-while-finite-Some Image-mono rtrancl-mono*
finite-subset[*OF - finite-rtrancl-delta-Image1*[*of r*]]) *auto*
then show *reachable as r* = {*fold delta w r* | *w*. *w* $\in$ *lists* (*set as*)}
unfolding *reachable-def* **by** (*cases wsZ*)
(*auto dest*!: *rtrancl-while-Some split*: *if-splits simp*: *rtrancl-fold-product1 image-iff*)
then show *finite* (*reachable as r*) **using** *assms* **by** (*force intro*: *finite-subset*[*OF - fin*])
qed

end

locale *DFAs* =
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* ⇒ *'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* $\in$ *set alphabet1* ⟷ *b* $\in$ *set alphabet2*
begin

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

lemma *finite-rtrancl-delta-Image*:
⟦*wellformed1 r*; *wellformed2 s*⟧ ⟹
finite ({((*r*, *s*), (*delta1 a r*, *delta2 b s*))| *a b r s*.
a $\in$ *set alphabet1* ∧ *b* $\in$ *set alphabet2* ∧ *R a b*}^∗ `` {(*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 ∃ *st*. *closure* (*r*, *s*) = *Some st* (**is** ∃ -. *closure ?i* = -)
proof (*rule rtrancl-while-finite-Some*)
show *finite* ({(*x*, *st*). *st* $\in$ *set* (*step x*)}* `` {*?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** *termination*
    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* (λ*a*. (*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 $x \notin$ *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 $x \notin$ *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*]: $0 \notin A \Longrightarrow i \in (\lambda x.\ x - Suc\ 0)$ ` $A \longleftrightarrow Suc\ i \in A$

by (*cases i*) (*fastforce simp*: *image-iff le-Suc-eq* *elim*: *contrapos-np*)+

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

lemma *funpow-cycle*: $(f\ \hat{}\hat{}\ k)\ x = x \Longrightarrow (f\ \hat{}\hat{}\ l)\ x = (f\ \hat{}\hat{}\ (l\ mod\ k))\ x$
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 $(f\ \hat{}\hat{}\ k)\ x = (f\ \hat{}\hat{}\ i)\ x\ \ i \le k\ \ i \le l$
shows $(f\ \hat{}\hat{}\ l)\ x = (f\ \hat{}\hat{}\ ((l - i)\ mod\ (k - i) + i))\ x$
proof −
from *assms* **have**
$(f\ \hat{}\hat{}\ (k - i))\ ((f\ \hat{}\hat{}\ i)\ x) = ((f\ \hat{}\hat{}\ i)\ x)$
$(f\ \hat{}\hat{}\ l)\ x = (f\ \hat{}\hat{}\ (l - i))\ ((f\ \hat{}\hat{}\ i)\ x)$
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*: $x \in set\ (tl\ xs) \Longrightarrow x \in set\ xs$
by (*cases xs*) *auto*

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

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* φ ψ) = *disjuncts* φ ∪ *disjuncts* ψ
| *disjuncts* φ = {φ}

fun *conjuncts* **where**
conjuncts (*FAnd* φ ψ) = *conjuncts* φ ∪ *conjuncts* ψ
| *conjuncts* φ = {φ}

fun *disjuncts-list* **where**
disjuncts-list (*FOr* φ ψ) = *disjuncts-list* φ @ *disjuncts-list* ψ
| *disjuncts-list* φ = [φ]

fun *conjuncts-list* **where**
conjuncts-list (*FAnd* φ ψ) = *conjuncts-list* φ @ *conjuncts-list* ψ
| *conjuncts-list* φ = [φ]

lemma *finite-juncts*[*simp*]: *finite* (*disjuncts* φ) *finite* (*conjuncts* φ)
and *nonempty-juncts*[*simp*]: *disjuncts* φ ≠ {} *conjuncts* φ ≠ {}
by (*induct* φ) *auto*

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

lemma *notin-juncts*:
⟦ψ ∈ *disjuncts* φ; *is-FOr* ψ⟧ ⟹ *False*
⟦ψ ∈ *conjuncts* φ; *is-FAnd* ψ⟧ ⟹ *False*
by (*induct* φ) *auto*

lemma *juncts-list-singleton*:
¬ *is-FOr* φ ⟹ *disjuncts-list* φ = [φ]
¬ *is-FAnd* φ ⟹ *conjuncts-list* φ = [φ]
by (*induct* φ) *auto*

lemma *juncts-singleton*:
¬ *is-FOr* φ ⟹ *disjuncts* φ = {φ}
¬ *is-FAnd* φ ⟹ *conjuncts* φ = {φ}
by (*induct* φ) *auto*

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

primrec *norm-ACI* (‹⟨-⟩›) **where**
⟨*FBool b*⟩ = *FBool b*
| ⟨*FBase a*⟩ = *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 $(FOr\ \psi 1\ \psi 2) = (\neg\ is\text{-}FOr\ \psi 1 \wedge (let\ \varphi s = \psi 1\ \#\ disjuncts\text{-}list\ \psi 2\ in$
$sorted\ \varphi s \wedge distinct\ \varphi s \wedge nf\text{-}ACI\ \psi 1 \wedge nf\text{-}ACI\ \psi 2))$
| *nf-ACI* $(FAnd\ \psi 1\ \psi 2) = (\neg\ is\text{-}FAnd\ \psi 1 \wedge (let\ \varphi s = \psi 1\ \#\ conjuncts\text{-}list\ \psi 2\ in$
$sorted\ \varphi s \wedge distinct\ \varphi s \wedge nf\text{-}ACI\ \psi 1 \wedge nf\text{-}ACI\ \psi 2))$
| *nf-ACI* $(FNot\ \varphi) = nf\text{-}ACI\ \varphi$
| *nf-ACI* $(FEx\ k\ \varphi) = nf\text{-}ACI\ \varphi$
| *nf-ACI* $(FAll\ k\ \varphi) = nf\text{-}ACI\ \varphi$
| *nf-ACI* $\varphi = True$

lemma *nf-ACI-D*:
$nf\text{-}ACI\ \varphi \Longrightarrow sorted\ (disjuncts\text{-}list\ \varphi)$
$nf\text{-}ACI\ \varphi \Longrightarrow sorted\ (conjuncts\text{-}list\ \varphi)$
$nf\text{-}ACI\ \varphi \Longrightarrow distinct\ (disjuncts\text{-}list\ \varphi)$
$nf\text{-}ACI\ \varphi \Longrightarrow distinct\ (conjuncts\text{-}list\ \varphi)$
$nf\text{-}ACI\ \varphi \Longrightarrow list\text{-}all\ nf\text{-}ACI\ (disjuncts\text{-}list\ \varphi)$
$nf\text{-}ACI\ \varphi \Longrightarrow list\text{-}all\ nf\text{-}ACI\ (conjuncts\text{-}list\ \varphi)$
by (*induct* φ) (*auto simp*: *juncts-list-singleton*)

lemma *disjuncts-list-nFOR*:
$[\![list\text{-}all\ (\lambda x.\ \neg\ is\text{-}FOr\ x)\ \varphi s;\ \varphi s \neq []]\!] \Longrightarrow disjuncts\text{-}list\ (nFOR\ \varphi s) = \varphi s$
by (*induct* φs *rule*: *nFOR.induct*) (*auto simp*: *juncts-list-singleton*)

lemma *conjuncts-list-nFAND*:
$[\![list\text{-}all\ (\lambda x.\ \neg\ is\text{-}FAnd\ x)\ \varphi s;\ \varphi s \neq []]\!] \Longrightarrow conjuncts\text{-}list\ (nFAND\ \varphi s) = \varphi s$
by (*induct* φs *rule*: *nFAND.induct*) (*auto simp*: *juncts-list-singleton*)

lemma *disjuncts-NFOR*:
$[\![finite\ X;\ X \neq \{\};\ \forall x \in X.\ \neg\ is\text{-}FOr\ x]\!] \Longrightarrow disjuncts\ (NFOR\ X) = X$
unfolding *NFOR-def* **by** (*auto simp*: *juncts-eq-set-juncts-list list-all-iff disjuncts-list-nFOR*)

lemma *conjuncts-NFAND*:
$[\![finite\ X;\ X \neq \{\};\ \forall x \in X.\ \neg\ is\text{-}FAnd\ x]\!] \Longrightarrow conjuncts\ (NFAND\ X) = X$
unfolding *NFAND-def* **by** (*auto simp*: *juncts-eq-set-juncts-list list-all-iff conjuncts-list-nFAND*)

lemma *nf-ACI-nFOR*:
$[\![sorted\ \varphi s;\ distinct\ \varphi s;\ list\text{-}all\ nf\text{-}ACI\ \varphi s;\ list\text{-}all\ (\lambda x.\ \neg\ is\text{-}FOr\ x)\ \varphi s]\!] \Longrightarrow nf\text{-}ACI\ (nFOR\ \varphi s)$
by (*induct* φs *rule*: *nFOR.induct*)
(*auto simp*: *juncts-list-singleton disjuncts-list-nFOR nf-ACI-D*)

lemma *nf-ACI-nFAND*:

$[\![$*sorted* φs; *distinct* φs; *list-all nf-ACI* φs; *list-all* $(\lambda x.\ \neg$ *is-FAnd* $x)$ $\varphi s]\!] \Longrightarrow$
nf-ACI (*nFAND* φs)
by (*induct* φs *rule*: *nFAND.induct*)
(*auto simp*: *juncts-list-singleton conjuncts-list-nFAND nf-ACI-D*)

lemma *nf-ACI-juncts*:
$[\![\psi \in$ *disjuncts* φ; *nf-ACI* $\varphi]\!] \Longrightarrow$ *nf-ACI* ψ
$[\![\psi \in$ *conjuncts* φ; *nf-ACI* $\varphi]\!] \Longrightarrow$ *nf-ACI* ψ
by (*induct* φ) *auto*

lemma *nf-ACI-norm-ACI*: *nf-ACI* $\langle\varphi\rangle$
by (*induct* φ)
(*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*: *nFOR* $(x \# xs) =$ (*if* $xs = []$ *then* x *else* *FOr* x (*nFOR* xs))
by (*cases* xs) *simp-all*

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

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

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

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

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

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

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

lemma *norm-ACI-juncts*:
nf-ACI $\varphi \Longrightarrow$ *norm-ACI* ‘ *disjuncts* $\varphi =$ *disjuncts* φ
nf-ACI $\varphi \Longrightarrow$ *norm-ACI* ‘ *conjuncts* $\varphi =$ *conjuncts* φ
by (*drule nf-ACI-D*(*5*,*6*), *force simp*: *list-all-iff juncts-eq-set-juncts-list norm-ACI-if-nf-ACI*)+

lemma
norm-ACI-NFOR: *nf-ACI* $\varphi \Longrightarrow \varphi$ = *NFOR* (*norm-ACI* ‘ *disjuncts* φ) **and**
norm-ACI-NFAND: *nf-ACI* $\varphi \Longrightarrow \varphi$ = *NFAND* (*norm-ACI* ‘ *conjuncts* φ)
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 (‹-ˉ-› [*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**
$FV\ (FBool\ \text{-})\ k = \{\}$
$|\ FV\ (FBase\ a)\ k = FV0\ k\ a$
$|\ FV\ (FNot\ \varphi)\ k = FV\ \varphi\ k$
$|\ FV\ (FOr\ \varphi\ \psi)\ k = FV\ \varphi\ k \cup FV\ \psi\ k$
$|\ FV\ (FAnd\ \varphi\ \psi)\ k = FV\ \varphi\ k \cup FV\ \psi\ k$
$|\ FV\ (FEx\ k'\ \varphi)\ k = (if\ k' = k\ then\ (\lambda x.\ x - 1)\ `\ (FV\ \varphi\ k - \{0\})\ else\ FV\ \varphi\ k)$
$|\ FV\ (FAll\ k'\ \varphi)\ k = (if\ k' = k\ then\ (\lambda x.\ x - 1)\ `\ (FV\ \varphi\ k - \{0\})\ else\ FV\ \varphi\ k)$

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

primrec *wf* :: $'n \Rightarrow ('a, 'k)$ *aformula* $\Rightarrow$ *bool* **where**
$wf\ n\ (FBool\ \text{-}) = True$
$|\ wf\ n\ (FBase\ a) = wf0\ n\ a$
$|\ wf\ n\ (FNot\ \varphi) = wf\ n\ \varphi$
$|\ wf\ n\ (FOr\ \varphi\ \psi) = (wf\ n\ \varphi \wedge wf\ n\ \psi)$
$|\ wf\ n\ (FAnd\ \varphi\ \psi) = (wf\ n\ \varphi \wedge wf\ n\ \psi)$
$|\ wf\ n\ (FEx\ k\ \varphi) = wf\ (SUC\ k\ n)\ \varphi$
$|\ wf\ n\ (FAll\ k\ \varphi) = wf\ (SUC\ k\ n)\ \varphi$

primrec *lformula* :: $('a, 'k)$ *aformula* $\Rightarrow$ *bool* **where**
$lformula\ (FBool\ \text{-}) = True$
$|\ lformula\ (FBase\ a) = lformula0\ a$
$|\ lformula\ (FNot\ \varphi) = lformula\ \varphi$
$|\ lformula\ (FOr\ \varphi\ \psi) = (lformula\ \varphi \wedge lformula\ \psi)$
$|\ lformula\ (FAnd\ \varphi\ \psi) = (lformula\ \varphi \wedge lformula\ \psi)$
$|\ lformula\ (FEx\ k\ \varphi) = lformula\ \varphi$
$|\ lformula\ (FAll\ k\ \varphi) = lformula\ \varphi$

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* :: $('k \Rightarrow 'v \Rightarrow nat \Rightarrow bool) \Rightarrow 'i \Rightarrow ('a, 'k)$ *aformula* $\Rightarrow$ *bool*
where
$\quad$ *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* φ) = $(\neg$ *satisfies-gen* r $\mathfrak{A}$ $\varphi)$
| *satisfies-gen* r $\mathfrak{A}$ (*FOr* φ_1 φ_2) = (*satisfies-gen* r $\mathfrak{A}$ φ_1 $\vee$ *satisfies-gen* r $\mathfrak{A}$ φ_2)
| *satisfies-gen* r $\mathfrak{A}$ (*FAnd* φ_1 φ_2) = (*satisfies-gen* r $\mathfrak{A}$ φ_1 $\wedge$ *satisfies-gen* r $\mathfrak{A}$ φ_2)
| *satisfies-gen* r $\mathfrak{A}$ (*FEx* k φ) = ($\exists P$. r k P (*Length* $\mathfrak{A}$) $\wedge$ *satisfies-gen* r (*Extend* k *0* $\mathfrak{A}$ P) φ)
| *satisfies-gen* r $\mathfrak{A}$ (*FAll* k φ) = ($\forall P$. r k P (*Length* $\mathfrak{A}$) $\longrightarrow$ *satisfies-gen* r (*Extend* k *0* $\mathfrak{A}$ P) φ)

abbreviation *satisfies* (**infix** ‹$\models$› *50*) **where**
$\quad \mathfrak{A} \models \varphi \equiv$ *satisfies-gen* (λ- - -. *True*) $\mathfrak{A}$ φ

abbreviation *satisfies-bounded* (**infix** ‹$\models_b$› *50*) **where**
$\quad \mathfrak{A} \models_b \varphi \equiv$ *satisfies-gen* (λ- P n. *len* $P \leq n$) $\mathfrak{A}$ φ

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

definition *sat* **where**
$\quad$ *sat* $\mathfrak{A}$ $\varphi \equiv$ *sat-vars-gen* (λ- - -. *True*) *UNIV* $\mathfrak{A}$ φ

definition sat_b **where**
$\quad sat_b$ $\mathfrak{A}$ $\varphi \equiv$ *sat-vars-gen* (λ- P n. *len* $P \leq n$) *UNIV* $\mathfrak{A}$ φ

fun *RESTR* **where**
$\quad$ *RESTR* (*FOr* φ ψ) = *FOr* (*RESTR* φ) (*RESTR* ψ)
| *RESTR* (*FAnd* φ ψ) = *FAnd* (*RESTR* φ) (*RESTR* ψ)
| *RESTR* (*FNot* φ) = *FNot* (*RESTR* φ)
| *RESTR* (*FEx* k φ) = *FEx* k (*FAnd* (*Restrict* k *0*) (*RESTR* φ))
| *RESTR* (*FAll* k φ) = *FAll* k (*FOr* (*FNot* (*Restrict* k *0*)) (*RESTR* φ))
| *RESTR* φ = φ

abbreviation *RESTRICT-VARS* **where**
$\quad$ *RESTRICT-VARS* *ks* V $\varphi \equiv$
$\quad\quad$ *foldr* (%k φ. *foldr* (λx φ. *FAnd* (*Restrict* k x) φ) (V k) φ) *ks* (*RESTR* φ)

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

primrec *nullable* :: $('a, 'k)$ *aformula* $\Rightarrow$ *bool* **where**
$\quad$ *nullable* (*FBool* b) = b
| *nullable* (*FBase* a) = *nullable0* a
| *nullable* (*FNot* φ) = ($\neg$ *nullable* φ)

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

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

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

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

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

fun $nFNot$ :: $('a,\ 'k)\ aformula \Rightarrow ('a,\ 'k)\ aformula$ **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 $lderiv \equiv deriv\ lderiv0$
abbreviation $rderiv \equiv deriv\ rderiv0$

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

lemma *fold-deriv-FNot*:
  $fold\ (deriv\ d0)\ xs\ (FNot\ \varphi) = FNot\ (fold\ (deriv\ d0)\ xs\ \varphi)$
  by $(induct\ xs\ arbitrary{:}\ \varphi)\ auto$

lemma *fold-deriv-FOr*:
  $fold\ (deriv\ d0)\ xs\ (FOr\ \varphi\ \psi) = FOr\ (fold\ (deriv\ d0)\ xs\ \varphi)\ (fold\ (deriv\ d0)\ xs\ \psi)$
  by $(induct\ xs\ arbitrary{:}\ \varphi\ \psi)\ auto$

lemma *fold-deriv-FAnd*:
  $fold\ (deriv\ d0)\ xs\ (FAnd\ \varphi\ \psi) = FAnd\ (fold\ (deriv\ d0)\ xs\ \varphi)\ (fold\ (deriv\ d0)\ xs\ \psi)$
  by $(induct\ xs\ arbitrary{:}\ \varphi\ \psi)\ auto$

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

$\varphi\rangle)\}$
proof $-$
  { fix *xs*
    have $\exists\,\psi.\ \langle fold\ (deriv\ d0)\ xs\ (FEx\ k\ \varphi)\rangle = FEx\ k\ \psi\ \wedge$
      $nf\text{-}ACI\ \psi \wedge disjuncts\ \psi \subseteq (\bigcup xs.\ disjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi\rangle)$
    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* $?\varphi$] **obtain** ψ **where** $\langle fold\ (deriv\ d0)\ xs\ (FEx\ k\ ?\varphi)\rangle = FEx\ k$
ψ
        $nf\text{-}ACI\ \psi$ **and** $*$: $disjuncts\ \psi \subseteq (\bigcup xs.\ disjuncts\ \langle fold\ (deriv\ d0)\ xs\ ?\varphi\rangle)$ **by**
blast+
      then show *?case*
      proof (*intro exI conjI*)
        have $(\bigcup xs.\ disjuncts\ \langle fold\ (deriv\ d0)\ xs\ ?\varphi\rangle) \subseteq$
          $(\bigcup xs.\ disjuncts\ \langle fold\ (Formula\text{-}Operations.deriv\ extend\ d0)\ xs\ \varphi\rangle)$
        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** $disjuncts\ \psi \subseteq \ldots$ **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*:
  $\{\langle fold\ (deriv\ d0)\ xs\ (FAll\ k\ \varphi)\rangle \mid xs.\ True\} \subseteq$
    $\{FAll\ k\ \psi \mid \psi.\ nf\text{-}ACI\ \psi \wedge conjuncts\ \psi \subseteq (\bigcup xs.\ conjuncts\ \langle fold\ (deriv\ d0)\ xs$
$\varphi\rangle)\}$
proof $-$
  { fix *xs*
    have $\exists\,\psi.\ \langle fold\ (deriv\ d0)\ xs\ (FAll\ k\ \varphi)\rangle = FAll\ k\ \psi\ \wedge$
      $nf\text{-}ACI\ \psi \wedge conjuncts\ \psi \subseteq (\bigcup xs.\ conjuncts\ \langle fold\ (deriv\ d0)\ xs\ \varphi\rangle)$
    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* $?\varphi$] **obtain** ψ **where** $\langle fold\ (deriv\ d0)\ xs\ (FAll\ k\ ?\varphi)\rangle = FAll\ k$
ψ
        $nf\text{-}ACI\ \psi$ **and** $*$: $conjuncts\ \psi \subseteq (\bigcup xs.\ conjuncts\ \langle fold\ (deriv\ d0)\ xs\ ?\varphi\rangle)$
by *blast+*
      then show *?case*
      proof (*intro exI conjI*)
        have $(\bigcup xs.\ conjuncts\ \langle fold\ (deriv\ d0)\ xs\ ?\varphi\rangle) \subseteq$
          $(\bigcup xs.\ conjuncts\ \langle fold\ (Formula\text{-}Operations.deriv\ extend\ d0)\ xs\ \varphi\rangle)$
        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* $\psi \subseteq \ldots$ **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* $\Longrightarrow$ *finite* $\{f\ \varphi \mid \varphi.\ \textit{nf-ACI}\ \varphi \wedge \textit{disjuncts}\ \varphi \subseteq B\}$
finite B $\Longrightarrow$ *finite* $\{f\ \varphi \mid \varphi.\ \textit{nf-ACI}\ \varphi \wedge \textit{conjuncts}\ \varphi \subseteq B\}$
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* × *'i* × *'k* :: {*linorder*, *enum*} × *'n* × *'x* × *'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*
$k \neq k' \Longrightarrow$ *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*: $[\![$*Length* $\mathfrak{A}$ = *0*; *Length* $\mathfrak{B}$ = *0*; $\#_V\ \mathfrak{A} = \#_V\ \mathfrak{B}]\!] \Longrightarrow \mathfrak{A} = \mathfrak{B}$
and *ex-Length-0*: $\exists\mathfrak{A}$. *Length* $\mathfrak{A}$ = *0* $\wedge$ $\#_V\ \mathfrak{A}$ = *idx*
and *Extend-commute-safe*: $[\![j \leq i$; *LEQ k i* ($\#_V\ \mathfrak{A}$)$]\!] \Longrightarrow$
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*: $k \neq k' \Longrightarrow$
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* ($\#_V\ \mathfrak{A}$) $\Longrightarrow$
$m^{\textit{Extend } k\ i\ \mathfrak{A}\ P}k' = (\textit{if } k = k' \textit{ then } (\textit{if } m = i \textit{ then } P \textit{ else dec } i\ m^{\mathfrak{A}}k) \textit{ else } m^{\mathfrak{A}}k')$
and *assigns-SNOC-zero*: *LESS k m* ($\#_V\ \mathfrak{A}$) $\Longrightarrow$ $m^{\textit{SNOC } (\textit{zero } (\#_V\ \mathfrak{A}))\ \mathfrak{A}}k = m^{\mathfrak{A}}k$
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*: $\#_V\ \mathfrak{A}$ = *size x* $\Longrightarrow$ *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*: $\#_V\ \mathfrak{A}$ = *size x* $\Longrightarrow$ *len P* $\leq$ *Length* (*SNOC x* $\mathfrak{A}$) $\Longrightarrow$

$Extend\ k\ 0\ (SNOC\ x\ \mathfrak{A})\ P =$
$SNOC\ (extend\ k\ (if\ eval\ P\ (Length\ \mathfrak{A})\ then\ True\ else\ False)\ x)\ (Extend\ k\ 0\ \mathfrak{A}\ (cut\ (Length\ \mathfrak{A})\ P))$
and $CONS\text{-}inj$: $size\ x = \#_V\ \mathfrak{A} \Longrightarrow size\ y = \#_V\ \mathfrak{A} \Longrightarrow \#_V\ \mathfrak{A} = \#_V\ \mathfrak{B} \Longrightarrow$
$CONS\ x\ \mathfrak{A} = CONS\ y\ \mathfrak{B} \longleftrightarrow (x = y \wedge \mathfrak{A} = \mathfrak{B})$
and $CONS\text{-}surj$: $Length\ \mathfrak{A} \neq 0 \Longrightarrow \#_V\ \mathfrak{A} = idx \Longrightarrow$
$\exists\, x\ \mathfrak{B}.\ \mathfrak{A} = CONS\ x\ \mathfrak{B} \wedge \#_V\ \mathfrak{B} = idx \wedge size\ x = idx$

and $size\text{-}zero$: $size\ (zero\ idx) = idx$
and $size\text{-}extend$: $size\ (extend\ k\ b\ x) = SUC\ k\ (size\ x)$
and $distinct\text{-}alphabet$: $distinct\ (alphabet\ idx)$
and $alphabet\text{-}size$: $x \in set\ (alphabet\ idx) \longleftrightarrow size\ x = idx$

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

and $wf0\text{-}decr0$: $[\![wf0\ (SUC\ k\ idx)\ a;\ LESS\ k\ l\ (SUC\ k\ idx);\ \neg\ find0\ k\ l\ a]\!] \Longrightarrow wf0\ idx\ (decr0\ k\ l\ a)$
and $lformula0\text{-}decr0$: $lformula0\ \varphi \Longrightarrow lformula0\ (decr0\ k\ l\ \varphi)$
and $Extend\text{-}satisfies0$: $[\![\neg\ find0\ k\ i\ a;\ LESS\ k\ i\ (SUC\ k\ (\#_V\ \mathfrak{A}));\ lformula0\ a \vee len\ P \leq Length\ \mathfrak{A}]\!] \Longrightarrow$
$Extend\ k\ i\ \mathfrak{A}\ P \models_0 a \longleftrightarrow \mathfrak{A} \models_0 decr0\ k\ i\ a$
and $nullable0\text{-}satisfies0$: $Length\ \mathfrak{A} = 0 \Longrightarrow nullable0\ a \longleftrightarrow \mathfrak{A} \models_0 a$
and $satisfies0\text{-}eqI$: $wf0\ (\#_V\ \mathfrak{B})\ a \Longrightarrow \#_V\ \mathfrak{A} = \#_V\ \mathfrak{B} \Longrightarrow lformula0\ a \Longrightarrow$
$(\bigwedge m\ k.\ LESS\ k\ m\ (\#_V\ \mathfrak{B}) \Longrightarrow m^{\mathfrak{A}}k = m^{\mathfrak{B}}k) \Longrightarrow \mathfrak{A} \models_0 a \longleftrightarrow \mathfrak{B} \models_0 a$
and $wf\text{-}lderiv0$: $[\![wf0\ idx\ a;\ lformula0\ a]\!] \Longrightarrow wf\ idx\ (lderiv0\ x\ a)$
and $lformula\text{-}lderiv0$: $lformula0\ a \Longrightarrow lformula\ (lderiv0\ x\ a)$
and $wf\text{-}rderiv0$: $wf0\ idx\ a \Longrightarrow wf\ idx\ (rderiv0\ x\ a)$
and $satisfies\text{-}lderiv0$:
$[\![wf0\ (\#_V\ \mathfrak{A})\ a;\ \#_V\ \mathfrak{A} = size\ x;\ lformula0\ a]\!] \Longrightarrow \mathfrak{A} \models lderiv0\ x\ a \longleftrightarrow CONS\ x\ \mathfrak{A} \models_0 a$
and $satisfies\text{-}bounded\text{-}lderiv0$:
$[\![wf0\ (\#_V\ \mathfrak{A})\ a;\ \#_V\ \mathfrak{A} = size\ x;\ lformula0\ a]\!] \Longrightarrow \mathfrak{A} \models_b lderiv0\ x\ a \longleftrightarrow CONS\ x\ \mathfrak{A} \models_0 a$
and $satisfies\text{-}bounded\text{-}rderiv0$:
$[\![wf0\ (\#_V\ \mathfrak{A})\ a;\ \#_V\ \mathfrak{A} = size\ x]\!] \Longrightarrow \mathfrak{A} \models_b rderiv0\ x\ a \longleftrightarrow SNOC\ x\ \mathfrak{A} \models_0 a$
and $find0\text{-}FV0$: $[\![wf0\ idx\ a;\ LESS\ k\ l\ idx]\!] \Longrightarrow find0\ k\ l\ a \longleftrightarrow l \in FV0\ k\ a$

and *finite-FV0*: *finite* (*FV0 k a*)
and *wf0-FV0-LESS*: $[\![$*wf0 idx a*; $v \in$ *FV0 k a*$]\!] \Longrightarrow$ *LESS k v idx*
and *restrict-Restrict*: $i^{\mathfrak{A}}k = P \Longrightarrow$ *restrict k P* $\longleftrightarrow$ *satisfies-gen r* $\mathfrak{A}$ (*Restrict k i*)
and *wf-Restrict*: *LESS k i idx* $\Longrightarrow$ *wf idx* (*Restrict k i*)
and *lformula-Restrict*: *lformula* (*Restrict k i*)
and *finite-lderiv0*: *lformula0 a* $\Longrightarrow$ *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*:
$[\![$*wf* ($\#_V$ $\mathfrak{A}$) φ; $\#_V$ $\mathfrak{A}$ = $\#_V$ $\mathfrak{B}$; $\bigwedge m\ k.$ *LESS k m* ($\#_V$ $\mathfrak{A}$) $\Longrightarrow m^{\mathfrak{A}}k = m^{\mathfrak{B}}k$; *lformula* $\varphi]\!] \Longrightarrow$
$\mathfrak{A} \models \varphi \longleftrightarrow \mathfrak{B} \models \varphi$
proof (*induct* φ *arbitrary*: $\mathfrak{A}$ $\mathfrak{B}$)
case (*FEx k* φ)
from *FEx.prems* **have** $\bigwedge P.$ (*Extend k 0* $\mathfrak{A}$ *P* $\models \varphi$) $\longleftrightarrow$ (*Extend k 0* $\mathfrak{B}$ *P* $\models \varphi$)
by (*intro FEx.hyps*) (*auto simp*: *nvars-Extend assigns-Extend dec-def gr0-conv-Suc LEQ-0 LESS-SUC*)
then show *?case* **by** *simp*
next
case (*FAll k* φ)
from *FAll.prems* **have** $\bigwedge P.$ (*Extend k 0* $\mathfrak{A}$ *P* $\models \varphi$) $\longleftrightarrow$ (*Extend k 0* $\mathfrak{B}$ *P* $\models \varphi$)
by (*intro FAll.hyps*) (*auto simp*: *nvars-Extend assigns-Extend dec-def gr0-conv-Suc LEQ-0 LESS-SUC*)
then show *?case* **by** *simp*
next
case (*FNot* φ)
from *FNot.prems* **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*:
$[\![$*wf* (*SUC k idx*) φ; *LEQ k l idx*; $\neg$ *find k l* $\varphi]\!] \Longrightarrow$ *wf idx* (*decr k l* φ)
by (*induct* φ *arbitrary*: *idx l*) (*auto simp*: *wf0-decr0 LESS-SUC SUC-SUC*)

lemma *lformula-decr*:
lformula $\varphi \Longrightarrow$ *lformula* (*decr k l* φ)
by (*induct* φ *arbitrary*: *l*) (*auto simp*: *lformula0-decr0*)

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

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

lemma *Extend-satisfies-bounded-decr*:
  $[\![\neg$ *find k i* φ; *LEQ k i* ($\#_V$ $\mathfrak{A}$); *len P* $\leq$ *Length* $\mathfrak{A}]\!] \Longrightarrow$
   Extend k i $\mathfrak{A}$ *P* $\models_b \varphi \longleftrightarrow \mathfrak{A} \models_b$ *decr k i* φ
proof (*induct* φ *arbitrary*: *i* $\mathfrak{A}$ *P*)
  case (*FEx k'* φ)
  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'* φ) **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 idx (*FOr* φ ψ) $\Longrightarrow$ *wf idx* (*nFOr* φ ψ)
  by (*induct* φ ψ *rule*: *nFOr.induct*) (*simp-all add*: *Let-def*)

lemma *wf-nFAnd*:
  wf idx (*FAnd* φ ψ) $\Longrightarrow$ *wf idx* (*nFAnd* φ ψ)
  by (*induct* φ ψ *rule*: *nFAnd.induct*) (*simp-all add*: *Let-def*)

lemma *wf-nFEx*:
wf idx (*FEx b* φ) $\Longrightarrow$ *wf idx* (*nFEx b* φ)
by (*induct* φ *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* φ) $\Longrightarrow$ *wf idx* (*nFAll b* φ)
by (*induct* φ *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* φ) $\Longrightarrow$ *wf idx* (*nFNot* φ)
by (*induct* φ *arbitrary*: *idx rule*: *nFNot.induct*) (*auto simp*: *wf-nFOr wf-nFAnd wf-nFEx wf-nFAll*)

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

lemma *lformula-nFOr*:
lformula (*FOr* φ ψ) $\Longrightarrow$ *lformula* (*nFOr* φ ψ)
by (*induct* φ ψ *rule*: *nFOr.induct*) (*simp-all add*: *Let-def*)

lemma *lformula-nFAnd*:
lformula (*FAnd* φ ψ) $\Longrightarrow$ *lformula* (*nFAnd* φ ψ)
by (*induct* φ ψ *rule*: *nFAnd.induct*) (*simp-all add*: *Let-def*)

lemma *lformula-nFEx*:
lformula (*FEx b* φ) $\Longrightarrow$ *lformula* (*nFEx b* φ)
by (*induct* φ *rule*: *nFEx.induct*)
(*auto simp*: *lformula-nFOr lformula0-decr0 lformula-decr*)

lemma *lformula-nFAll*:
lformula (*FAll b* φ) $\Longrightarrow$ *lformula* (*nFAll b* φ)
by (*induct* φ *rule*: *nFAll.induct*)
(*auto simp*: *lformula-nFAnd lformula0-decr0 lformula-decr*)

lemma *lformula-nFNot*:
lformula (*FNot* φ) $\Longrightarrow$ *lformula* (*nFNot* φ)
by (*induct* φ *rule*: *nFNot.induct*) (*auto simp*: *lformula-nFOr lformula-nFAnd lformula-nFEx lformula-nFAll*)

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

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

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

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

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

lemma *satisfies-bounded-nFNot*:
$\mathfrak{A} \models_b nFNot\ \varphi \longleftrightarrow \mathfrak{A} \models_b FNot\ \varphi$
by (*induct* φ *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* φ *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-lderiv*:
$[\![wf\ idx\ \varphi;\ lformula\ \varphi]\!] \Longrightarrow wf\ idx\ (lderiv\ x\ \varphi)$
by (*induct* φ *arbitrary*: *x idx*) (*auto simp*: *wf-lderiv0*)

lemma *lformula-lderiv*:
$lformula\ \varphi \Longrightarrow lformula\ (lderiv\ x\ \varphi)$
by (*induct* φ *arbitrary*: *x*) (*auto simp*: *lformula-lderiv0*)

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

theorem *satisfies-lderiv*:
$[\![wf\ (\#_V\ \mathfrak{A})\ \varphi;\ \#_V\ \mathfrak{A} = size\ x;\ lformula\ \varphi]\!] \Longrightarrow \mathfrak{A} \models lderiv\ x\ \varphi \longleftrightarrow CONS\ x\ \mathfrak{A} \models \varphi$
proof (*induct* φ *arbitrary*: $x\ \mathfrak{A}$)
case (*FEx k* φ)
from *FEx.prems 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* φ)
from *FAll.prems 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*:
  $[\![wf\ (\#_V\ \mathfrak{A})\ \varphi;\ \#_V\ \mathfrak{A} = size\ x;\ lformula\ \varphi]\!] \Longrightarrow \mathfrak{A} \models_b lderiv\ x\ \varphi \longleftrightarrow CONS\ x\ \mathfrak{A} \models_b \varphi$
proof (*induct* φ *arbitrary*: *x* $\mathfrak{A}$)
  case (*FEx k* φ)
  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.prems 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* φ)
  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.prems 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*:
  $[\![wf\ (\#_V\ \mathfrak{A})\ \varphi;\ \#_V\ \mathfrak{A} = size\ x]\!] \Longrightarrow \mathfrak{A} \models_b rderiv\ x\ \varphi \longleftrightarrow SNOC\ x\ \mathfrak{A} \models_b \varphi$
proof (*induct* φ *arbitrary*: *x* $\mathfrak{A}$)
  case (*FEx k* φ)
  from *FEx.prems 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* φ)
  from *FAll.prems 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*: $wf\ idx\ \varphi \Longrightarrow wf\ idx\ (((norm \circ rderiv\ (zero\ idx))\ \hat{}\hat{}\ k)\ \varphi)$
  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-lderiv0*]]
      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* $(\bigcup\ (disjuncts\ `\ \{\langle fold\ (deriv\ d0)\ xs\ \varphi\rangle \mid xs\ .\ True\}))$ **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* $(\bigcup\ (conjuncts\ `\ \{\langle fold\ (deriv\ d0)\ xs\ \varphi\rangle \mid xs\ .\ True\}))$ **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*: *finite* $\Phi \Longrightarrow$ *lformula* (*NFOR* Φ) = ($\forall \varphi \in \Phi$. *lformula* φ)
unfolding *NFOR-def o-apply lformula-nFOR* **by** *simp*

lemma *lformula-NFAND*: *finite* $\Phi \Longrightarrow$ *lformula* (*NFAND* Φ) = ($\forall \varphi \in \Phi$. *lformula* φ)
unfolding *NFAND-def o-apply lformula-nFAND* **by** *simp*

lemma *lformula-disjuncts*: ($\forall \psi \in$ *disjuncts* φ. *lformula* ψ) = *lformula* φ
by (*induct* φ *rule*: *disjuncts.induct*) *fastforce+*

lemma *lformula-conjuncts*: ($\forall \psi \in$ *conjuncts* φ. *lformula* ψ) = *lformula* φ
by (*induct* φ *rule*: *conjuncts.induct*) *fastforce+*

lemma *lformula-norm-ACI*: *lformula* $\langle\varphi\rangle$ = *lformula* φ
by (*induct* φ) (*simp-all add*: *ball-Un*
lformula-NFOR lformula-disjuncts lformula-NFAND lformula-conjuncts)

theorem
finite-fold-lderiv: *lformula* $\varphi \Longrightarrow$ *finite* $\{\langle$*fold lderiv xs* $\langle\varphi\rangle\rangle \mid$ *xs. True*$\}$ **and**
finite-fold-rderiv: *finite* $\{\langle$*fold rderiv xs* $\langle\varphi\rangle\rangle \mid$ *xs. True*$\}$
by (*subst* (*asm*) *lformula-norm-ACI*[*symmetric*]) (*blast intro*: *nf-ACI-norm-ACI finite-fold-deriv*)+

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

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

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

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

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

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

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

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

lemma *wf-juncts*:
  $wf\ idx\ \varphi \longleftrightarrow (\forall \psi \in disjuncts\ \varphi.\ wf\ idx\ \psi)$
  $wf\ idx\ \varphi \longleftrightarrow (\forall \psi \in conjuncts\ \varphi.\ wf\ idx\ \psi)$
  by $(induct\ \varphi)\ auto$

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

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 $(rule\ sym,\ induct\ \varphi\ arbitrary{:}\ \mathfrak{A})$
    $(auto\ simp{:}\ satisfies\text{-}bounded\text{-}NFOR\ satisfies\text{-}bounded\text{-}NFAND$
    $intro{:}\ iffD2[OF\ satisfies\text{-}bounded\text{-}disjuncts]\ iffD2[OF\ satisfies\text{-}bounded\text{-}conjuncts]$
    $dest{:}\ iffD1[OF\ satisfies\text{-}bounded\text{-}disjuncts]\ iffD1[OF\ satisfies\text{-}bounded\text{-}conjuncts])$

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

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

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

2.6 Emptiness Check

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *fut-invariant*: *fut-invariant* $\varphi\Phi \Longrightarrow$ *fut-test* $\varphi\Phi \Longrightarrow$ *fut-invariant* (*fut-step* $\varphi\Phi$)

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

lemma *fut-terminate*: *fut-invariant* $\varphi\Phi \Longrightarrow \neg$ *fut-test* $\varphi\Phi \Longrightarrow$ *fut-spec* $\varphi\Phi$
proof (*induct* $\varphi\Phi$, *unfold prod.case not-not*)
fix φ Φ **assume** *fut-invariant* (φ, Φ) $\varphi \in$ *set* Φ
then obtain i k **where** $i < k$ **and** φ-*def*: φ = *fut-derivs* i ψ
and Φ-*def*: Φ = *map* (λi. *fut-derivs* i ψ) (*rev* [$0..<k$])
and $*$: *fut-derivs* k ψ = *fut-derivs* i ψ **unfolding** *fut-invariant-def* **by** *auto*
have *set* Φ = {*fut-derivs* k ψ | k . *True*}
unfolding Φ-*def set-map set-rev set-upt* **proof** *safe*
fix j
show *fut-derivs* j $\psi \in$ (λi. *fut-derivs* i ψ) ' {$0..<k$}
proof (*cases* $j < k$)
case *False*
with $*$ ‹$i < k$› **have** *fut-derivs* j ψ = *fut-derivs* $((j - i)$ *mod* $(k - i) + i)$ ψ
unfolding *fut-derivs-def* **by** (*auto intro*: *funpow-cycle-offset*)
then show *?thesis* **using** ‹$i < k$› ‹$\neg$ $j < k$›
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* (φ, Φ)
unfolding *fut-spec-def* **using** *satisfies-bounded-fut-derivs* **by** (*auto simp*: *wf-fut-derivs*)
qed

lemma *fut-spec-while-default*:
fut-spec (*while-default fut-default fut-test fut-step* (ψ, []))
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* $\longleftrightarrow$
(*if* b *then* ($\exists k$. (*SNOC* (*zero idx*) $\widehat{}\,\widehat{}$ k) $\mathfrak{A} \models_b \psi$) *else* ($\forall k$. (*SNOC* (*zero idx*) $\widehat{}\,\widehat{}$ 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 φ) = *fut True idx* (*nFEx* k (*finalize* (*SUC* k *idx*) φ))
| *finalize idx* (*FAll* k φ) = *fut False idx* (*nFAll* k (*finalize* (*SUC* k *idx*) φ))
| *finalize idx* (*FOr* φ ψ) = *FOr* (*finalize idx* φ) (*finalize idx* ψ)

| *finalize idx* (*FAnd* φ ψ) = *FAnd* (*finalize idx* φ) (*finalize idx* ψ)
| *finalize idx* (*FNot* φ) = *FNot* (*finalize idx* φ)
| *finalize idx* φ = φ

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

lemma *wf-finalize*: *wf idx* φ $\Longrightarrow$ *wf idx* (*finalize idx* φ)
by (*induct* φ *arbitrary*: *idx*) (*auto simp*: *wf-fut wf-nFEx wf-nFAll*)

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

lemma *assigns-SNOCs-zero*:
$[\![$*LESS k m* ($\#_V$ $\mathfrak{A}$); $\#_V$ $\mathfrak{A}$ = *idx*$]\!]$ $\Longrightarrow$ $m^{(SNOC\ (zero\ idx)\ \hat{}\hat{}\ i)\ \mathfrak{A}}k = m^{\mathfrak{A}}k$
by (*induct i arbitrary*: $\mathfrak{A}$) (*auto simp*: *assigns-SNOC-zero nvars-SNOC fun-pow-swap1*)

lemma *Extend-SNOCs-zero-satisfies*: $[\![$*wf* (*SUC k idx*) φ; $\#_V$ $\mathfrak{A}$ = *idx*; *lformula* $\varphi$$]\!]$ $\Longrightarrow$
Extend k 0 ((*SNOC* (*zero* ($\#_V$ $\mathfrak{A}$)) ^^ *i*) $\mathfrak{A}$) *P* $\models$ φ $\longleftrightarrow$ *Extend k 0* $\mathfrak{A}$ *P* $\models$ φ
by (*rule satisfies-eqI*)
(*auto simp*: *nvars-Extend nvars-SNOCs assigns-Extend assigns-SNOCs-zero LEQ-0 LESS-SUC*
dec-def gr0-conv-Suc)

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

lemma *Extend-empty-satisfies0*:
$[\![$*Length* $\mathfrak{A}$ = *0*; *len P* = *0*$]\!]$ $\Longrightarrow$ *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*:
$[\![$*Length* $\mathfrak{A}$ = *0*; *len P* = *0*$]\!]$ $\Longrightarrow$ *Extend k 0* $\mathfrak{A}$ *P* $\models_b$ φ $\longleftrightarrow$ $\mathfrak{A}$ $\models_b$ φ
by (*induct* φ *arbitrary*: *k* $\mathfrak{A}$ *P*)
(*auto simp*: *Extend-empty-satisfies0 Length-Extend split*: *bool.splits*)

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

lemma *final-satisfies*:
$[\![$*wf idx* $\varphi \wedge$ *lformula* φ; *Length* $\mathfrak{A} = 0$; $\#_V\ \mathfrak{A} =$ *idx*$]\!] \Longrightarrow$ *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* φ)
by (*induct* φ *arbitrary*: $\mathfrak{A}$) (*auto simp*: *restrict-Restrict*[*symmetric*] *assigns-Extend LEQ-0*)

lemma *finite-FV*: *finite* (*FV* φ *k*)
by (*induct* φ) (*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* (λx. *FAnd* (*Restrict* $k\ x$)) $V\ \varphi$)
by (*induct* V *arbitrary*: φ) (*auto simp*: *restrict-Restrict*[*symmetric*])

lemma *sat-vars-RESTRICT-VARS*:
fixes φ
defines *vs* $\equiv$ *sorted-list-of-set o FV* φ
assumes $\forall k \in$ *set ks*. *finite* (*FV* φ *k*)
shows *sat-vars-gen r* (*set ks*) $\mathfrak{A}\ \varphi \longleftrightarrow$ *satisfies-gen* $r\ \mathfrak{A}$ (*RESTRICT-VARS ks vs* φ)
using *assms* **proof** (*induct ks*)
case (*Cons k ks*)
with *satisfies-gen-restrict*[*of r* $\mathfrak{A}$ (*RESTRICT-VARS ks vs* φ) *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* φ
unfolding *sat-def RESTRICT-def* **using** *sat-vars-RESTRICT-VARS*[*of Enum.enum*, *symmetric*]
by (*auto simp*: *finite-FV enum-UNIV*)

lemma sat_b*-RESTRICT*: $sat_b\ \mathfrak{A}\ \varphi \longleftrightarrow \mathfrak{A} \models_b$ *RESTRICT* φ
unfolding sat_b*-def RESTRICT-def* **using** *sat-vars-RESTRICT-VARS*[*of Enum.enum*, *symmetric*]
by (*auto simp*: *finite-FV enum-UNIV*)

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

lemma *wf-RESTRICT-VARS*: $[\![$*wf idx* φ; $\forall k \in$ *set ks*. $\forall v \in$ *set* (*vs k*). *LESS k v idx*$]\!] \Longrightarrow$
wf idx (*RESTRICT-VARS ks vs* φ)
proof (*induct ks*)

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

lemma *wf-FV-LESS*: $[\![$*wf idx* φ; $v \in$ *FV* φ $k]\!] \Longrightarrow$ *LESS k v idx*
by (*induct* φ *arbitrary*: *idx v*)
(*force simp*: *wf0-FV0-LESS LESS-SUC split*: *if-splits*)+

lemma *wf-RESTRICT*: *wf idx* $\varphi \Longrightarrow$ *wf idx* (*RESTRICT* φ)
unfolding *RESTRICT-def* **by** (*rule wf-RESTRICT-VARS*) (*auto simp*: *list-all-iff wf-FV-LESS finite-FV*)

lemma *lformula-RESTR*: *lformula* $\varphi \Longrightarrow$ *lformula* (*RESTR* φ)
by (*induct* φ) (*auto simp*: *lformula-Restrict*)

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

lemma *lformula-RESTRICT*: *lformula* $\varphi \Longrightarrow$ *lformula* (*RESTRICT* φ)
unfolding *RESTRICT-def* **by** (*rule lformula-RESTRICT-VARS*)

lemma *ex-fold-CONS*: $\exists\, xs\ \mathfrak{B}.\ \mathfrak{A} =$ *fold CONS xs* $\mathfrak{B} \wedge$ *Length* $\mathfrak{B} = 0 \wedge$ *Length* $\mathfrak{A} =$ *length xs* $\wedge$
$\#_V\ \mathfrak{B} = \#_V\ \mathfrak{A} \wedge (\forall\, x \in$ *set xs. 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** $\mathfrak{A} =$ *CONS a* $\mathfrak{B}$ $\#_V\ \mathfrak{B} = \#_V\ \mathfrak{A}$ *size a* $= \#_V\ \mathfrak{A}$ **by** *force*
moreover with *Suc*(*2*) **have** *Length* $\mathfrak{B} = m$ **by** (*simp add*: *Length-CONS*)
with *Suc*(*1*)[*of* $\mathfrak{B}$] **obtain** *xs* $\mathfrak{C}$ **where** $\mathfrak{B} =$ *fold CONS xs* $\mathfrak{C}$ *Length* $\mathfrak{C} = 0$ *Length* $\mathfrak{B} =$ *length xs*
$\#_V\ \mathfrak{C} = \#_V\ \mathfrak{B}$ $\forall\, x \in$ *set xs. 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**
$L\ idx\ I = Lang\ (\exists\mathfrak{A}.\ Length\ \mathfrak{A} = 0 \wedge \#_V\ \mathfrak{A} = idx \wedge \mathfrak{A} \in I)$
$(\lambda a.\ if\ size\ a = idx\ then\ L\ idx\ \{\mathfrak{B}.\ CONS\ a\ \mathfrak{B} \in I\}\ else\ Zero)$

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

lemma *L-alt*: $L\ idx\ I =$
$to\text{-}language\ \{xs.\ \exists\mathfrak{A} \in I.\ \exists\mathfrak{B}.\ \mathfrak{A} = fold\ CONS\ (rev\ xs)\ \mathfrak{B} \wedge Length\ \mathfrak{B} = 0 \wedge$
$\#_V\ \mathfrak{B} = idx \wedge (\forall x \in set\ xs.\ size\ x = idx)\}$
by (*coinduction arbitrary*: *I*)
(*auto 0 4 simp*: *L-empty intro*: *exI*[*of* - {}] *arg-cong*[*of* - - *to-language*])

definition $lang\ idx\ \varphi = L\ idx\ \{\mathfrak{A}.\ \mathfrak{A} \models \varphi \wedge \#_V\ \mathfrak{A} = idx\}$
definition $lang_b\ idx\ \varphi = L\ idx\ \{\mathfrak{A}.\ \mathfrak{A} \models_b \varphi \wedge \#_V\ \mathfrak{A} = idx\}$
definition $language\ idx\ \varphi = L\ idx\ \{\mathfrak{A}.\ sat\ \mathfrak{A}\ \varphi \wedge \#_V\ \mathfrak{A} = idx\}$
definition $language_b\ idx\ \varphi = L\ idx\ \{\mathfrak{A}.\ sat_b\ \mathfrak{A}\ \varphi \wedge \#_V\ \mathfrak{A} = idx\}$

lemma $lformula\ \varphi \Longrightarrow lang\ n\ (norm\ \varphi) = lang\ n\ \varphi$
unfolding *lang-def* **using** *satisfies-norm* **by** *auto*

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

lemma *in-language-L-size*: *in-language* $(L\ idx\ I)\ w \Longrightarrow x \in set\ w \Longrightarrow size\ x =$
idx
by (*induct w arbitrary*: *x I*) (*auto split*: *if-splits*)

end

sublocale *Formula* <
bounded: *DA alphabet idx* $\lambda\varphi.\ norm\ (RESTRICT\ \varphi)\ \lambda a\ \varphi.\ norm\ (lderiv\ a\ \varphi)$
nullable
$\lambda\varphi.\ wf\ idx\ \varphi \wedge lformula\ \varphi\ lang_b\ idx$
$\lambda\varphi.\ wf\ idx\ \varphi \wedge lformula\ \varphi\ language_b\ idx$ **for** *idx*
using *ex-Length-0*[*of idx*]
by *unfold-locales*
(*auto simp*: *lformula-norm lformula-lderiv distinct-alphabet alphabet-size wf-norm*
wf-lderiv
$lang_b$-*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 *Formula* <
unbounded?: *DA alphabet idx* $\lambda\varphi.\ norm\ (RESTRICT\ \varphi)\ \lambda a\ \varphi.\ norm\ (lderiv\ a\ \varphi)$
final idx
$\lambda\varphi.\ wf\ idx\ \varphi \wedge lformula\ \varphi\ lang\ idx$

$\lambda\varphi.$ *wf idx* $\varphi \wedge$ *lformula* φ *language idx* **for** *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-eqv-soundness*:
$[\![\#_V\ \mathfrak{A} = idx;\ check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow sat\ \mathfrak{A}\ \varphi \longleftrightarrow sat\ \mathfrak{A}\ \psi$
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-eqv-soundness*:
$[\![\#_V\ \mathfrak{A} = idx;\ bounded.check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow sat_b\ \mathfrak{A}\ \varphi \longleftrightarrow sat_b\ \mathfrak{A}\ \psi$
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* |∈| *P*)
definition *downshift P* = ($\lambda x.\ x - Suc\ 0$) |‘| (*P* |−| {|*0*|})
definition *upshift P* = *Suc* |‘| *P*
definition *lift bs i P* = (*if bs* ! *i then finsert 0* (*upshift P*) *else upshift P*)
definition *snoc n bs i P* = (*if bs* ! *i then finsert n P else P*)
definition *cut n P* = *ffilter* ($\lambda i.\ i < n$) *P*
definition *len P* = (*if P* = {||} *then 0 else Suc* (*fMax P*))

datatype *order* = *FO* | *SO*
derive *linorder order*
instantiation *order* :: *enum* **begin**
definition *enum-order* = [*FO*, *SO*]
definition *enum-all-order P* = (*P FO* ∧ *P SO*)
definition *enum-ex-order P* = (*P FO* ∨ *P 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* × *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* = {||} ∨ *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*) ≤ *n*}
by *auto*

setup-lifting *type-definition-interp*

lift-definition *assigns* :: *nat* ⇒ *interp* ⇒ *order* ⇒ *nat fset* (‹-¯-› [*900*, *999*, *999*] *999*)
is λ*n* (-, *I1*, *I2*) *ord*. *case ord of FO* ⇒ *if n* < *length I1 then I1* ! *n else* {||}
| *SO* ⇒ *if n* < *length I2 then I2* ! *n else* {||} **.**
lift-definition *nvars* :: *interp* ⇒ *idx* (‹$\#_V$ -› [*1000*] *900*)
is λ(-, *I1*, *I2*). (*length I1*, *length I2*) **.**
lift-definition *Length* :: *interp* ⇒ *nat*
is λ(*n*, -, -). *n* **.**
lift-definition *Extend* :: *order* ⇒ *nat* ⇒ *interp* ⇒ *nat fset* ⇒ *interp*
is λ*ord i* (*n*, *I1*, *I2*) *P*. *case ord of*
FO ⇒ (*max n* (*if P* = {||} *then 0 else Suc* (*fMax P*)), *insert-nth i P I1*, *I2*)
| *SO* ⇒ (*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* × *bool list*) ⇒ *interp* ⇒ *interp*
is λ(*bs1*, *bs2*) (*n*, *I1*, *I2*).
(*Suc n*, *map-index* (*lift bs1*) *I1*, *map-index* (*lift bs2*) *I2*)
by *auto*

lift-definition *SNOC* :: (*bool list* × *bool list*) ⇒ *interp* ⇒ *interp*
is λ(*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* × *bool list*

lift-definition *zero* :: *idx* ⇒ *atom*
is λ(*m*, *n*). (*replicate m False*, *replicate n False*) **.**

definition *extend ord b* ≡
λ(*bs1*, *bs2*). *case ord of FO* ⇒ (*b* # *bs1*, *bs2*) | *SO* ⇒ (*bs1*, *b* # *bs2*)
lift-definition *size-atom* :: *bool list* × *bool list* ⇒ *idx*
is λ(*bs1*, *bs2*). (*length bs1*, *length bs2*) **.**

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

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

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

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

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

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

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

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

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

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

lemma *Extend-commute-safe*[*simp*]:
$[\![j \le i;\ LEQ\ ord\ i\ (\#_V\ \mathfrak{A})]\!] \Longrightarrow$
$Extend\ ord\ j\ (Extend\ ord\ i\ \mathfrak{A}\ P1)\ P2 = Extend\ ord\ (Suc\ i)\ (Extend\ ord\ j\ \mathfrak{A}$

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

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

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

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

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

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

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

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

lemma *map-index'-eq-conv*[*simp*]:
  $map\text{-}index'\ i\ f\ xs = map\text{-}index'\ j\ g\ xs = (\forall k < length\ xs.\ f\ (i + k)\ (xs\ !\ k) = g\ (j + k)\ (xs\ !\ k))$
proof (*induct xs arbitrary*: *i j*)
  case *Cons* **from** *Cons*(*1*)[*of Suc i Suc j*] **show** *?case* **by** (*auto simp*: *nth-Cons split*: *nat.splits*)
qed *simp*

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

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

lemma *Suc-fMax-pred-fimage*[*simp*]:
assumes *Suc p* |∈| *P 0* |∉| *P*
shows *Suc* (*fMax* ((λ*x*. *x* − *Suc 0*) |`| *P*)) = *fMax P*
using *assms* **by** (*subst mono-fMax-commute*[*of Suc*, *unfolded mono-def*, *simplified*]) (*auto simp*: *o-def*)

lemma *Extend-CONS*[*simp*]: $\#_V$ 𝔄 = *size-atom x* ⟹ *Extend ord 0* (*CONS x* 𝔄) *P* =
CONS (*extend ord* (*eval P 0*) *x*) (*Extend ord 0* 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-atom (*extend ord b x*) = *SUC ord* (*size-atom x*)
unfolding *extend-def* **by** *transfer* (*simp split*: *prod.splits order.splits*)

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

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

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

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

lemma *MSB-eq-SucD*: *MSB Is* = *Suc x* ⟹ ∃ *P*∈*set Is*. *x* |∈| *P*
using *Max-in*[*of* ⋃*x*∈*set Is*. *fset x*]
unfolding *MSB-def* **by** (*force split*: *if-splits*)

lemma *append-replicate-inj*:
assumes *xs* ≠ [] ⟹ *last xs* ≠ *x* **and** *ys* ≠ [] ⟹ *last ys* ≠ *x*
shows *xs* @ *replicate m x* = *ys* @ *replicate n x* ⟷ (*xs* = *ys* ∧ *m* = *n*)
proof *safe*

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

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

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

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

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

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

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

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

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

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

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

lemma *in-fimage-Suc*[*simp*]: x |∈| *Suc* |‘| $A \longleftrightarrow (\exists\, y.\ y$ |∈| $A \wedge x = $ *Suc* y)
  by *blast*

lemma *fimage-Suc-inj*[*simp*]: *Suc* |‘| A = *Suc* |‘| $B \longleftrightarrow A = B$
  by *blast*

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

lemma *Suc-in-fimage-Suc*: *Suc* x |∈| *Suc* |‘| $X \longleftrightarrow x$ |∈| X
  by *auto*

lemma *Suc-in-fimage-Suc-o-Suc*[*simp*]: *Suc* x |∈| (*Suc* ∘ *Suc*) |‘| $X \longleftrightarrow x$ |∈| *Suc* |‘| X
  by *auto*

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

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

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

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

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

lemma *len-downshift-helper*:
$x \mid\in\mid P \Longrightarrow Suc\ (fMax\ ((\lambda x.\ x - Suc\ 0) \mid`\mid (P \mid-\mid \{|0|\}))) \neq fMax\ P \Longrightarrow xa \mid\in\mid P \Longrightarrow xa = 0$
proof −
assume *a1*: $xa \mid\in\mid P$
assume *a2*: $Suc\ (fMax\ ((\lambda x.\ x - Suc\ 0) \mid`\mid (P \mid-\mid \{|0|\}))) \neq fMax\ P$
have $xa \mid\in\mid \{|0|\} \longrightarrow xa = 0$ **by** *fastforce*
moreover
{ assume $xa \mid\notin\mid \{|0|\}$
hence $0 \mid\notin\mid P \mid-\mid \{|0|\} \wedge xa \mid\notin\mid \{|0|\}$ **by** *blast*
then obtain $esk1_1 :: nat \Rightarrow nat$ **where** $xa = 0$ **using** *a1 a2* **by** (*metis Suc-fMax-pred-fimage fMax-Diff-0 fminus-iff not0-implies-Suc*) **}**
ultimately show $xa = 0$ **by** *blast*
qed

lemma *CONS-inj*[*simp*]: $size\text{-}atom\ x = \#_V\ \mathfrak{A} \Longrightarrow size\text{-}atom\ y = \#_V\ \mathfrak{A} \Longrightarrow \#_V\ \mathfrak{A} = \#_V\ \mathfrak{B} \Longrightarrow$
$CONS\ x\ \mathfrak{A} = CONS\ y\ \mathfrak{B} \longleftrightarrow (x = y \wedge \mathfrak{A} = \mathfrak{B})$
by *transfer* (*auto simp*: *list-eq-iff-nth-eq lift-def upshift-def split*: *if-splits*; *blast*)

lemma *Suc-minus1*: $Suc\ (x - Suc\ 0) = (if\ x = 0\ then\ Suc\ 0\ else\ x)$
by *auto*

lemma *fset-eq-empty-iff*: $(fset\ X = \{\}) = (X = \{||\})$
by (*metis bot-fset.rep-eq fset-inverse*)

lemma *fset-le-singleton-iff*: $(fset\ X \subseteq \{x\}) = (X = \{||\} \vee X = \{|x|\})$
by (*metis finsert.rep-eq fset-eq-empty-iff fset-inject order-refl singleton-insert-inj-eq subset-singletonD*)

lemma *MSB-decreases*:
$MSB\ I \leq Suc\ m \Longrightarrow MSB\ (map\ (\lambda X.\ (\lambda I1.\ I1 - Suc\ 0) \mid`\mid (X \mid-\mid \{|0|\}))\ I) \leq m$
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 $\exists\, x\ \mathfrak{B}.\ \mathfrak{A} = \mathit{CONS}\ x\ \mathfrak{B} \wedge \#_V\ \mathfrak{B} = \#_V\ \mathfrak{A} \wedge \mathit{size\text{-}atom}\ x = \#_V\ \mathfrak{A}$
proof −
have $\mathit{MSB}\ I1 \le \mathit{Suc}\ m \Longrightarrow \mathit{MSB}\ I2 \le \mathit{Suc}\ m \Longrightarrow \exists\, a\ b\ ab\ ba.$
$(\exists\, I1.\ \mathit{length}\ ab = \mathit{length}\ I1 \wedge (\forall\, i{<}\mathit{length}\ ab.\ ab\ !\ i = I1\ !\ i) \wedge$
$(\exists\, I2.\ \mathit{length}\ ba = \mathit{length}\ I2 \wedge (\forall\, i{<}\mathit{length}\ ba.\ ba\ !\ i = I2\ !\ i) \wedge$
$\mathit{MSB}\ I1 \le m \wedge \mathit{MSB}\ I2 \le m)) \wedge$
$I1 = \mathit{map\text{-}index}\ (\mathit{lift}\ a)\ ab \wedge$
$I2 = \mathit{map\text{-}index}\ (\mathit{lift}\ b)\ ba \wedge$
$\mathit{length}\ ab = \mathit{length}\ I1 \wedge$
$\mathit{length}\ ba = \mathit{length}\ I2 \wedge$
$\mathit{length}\ a = \mathit{length}\ I1 \wedge$
$\mathit{length}\ b = \mathit{length}\ I2$
for *m I1 I2*
by (*auto simp*: *list-eq-iff-nth-eq lift-def upshift-def split*: *if-splits*
intro!: *exI*[*of - map* ($\lambda X.\ 0$ |∈| X) -] *exI*[*of - map* ($\lambda X.$ ($\lambda x.\ x - \mathit{Suc}\ 0$) |'|
(X |−| {|*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* ∧ (*case i of Some i* ⇒ *i* ≤ *n* | - ⇒ *True*))
| *wf0 idx* (*Less - m1 m2*) = (*LESS FO m1 idx* ∧ *LESS FO m2 idx*)
| *wf0 idx* (*Plus-FO i m1 m2 n*) =
(*LESS FO m1 idx* ∧ *LESS FO m2 idx* ∧ (*case i of Some i* ⇒ *i* ≤ *n* | - ⇒ *True*))
| *wf0 idx* (*Eq-FO - m1 m2*) = (*LESS FO m1 idx* ∧ *LESS FO m2 idx*)
| *wf0 idx* (*Eq-SO M1 M2*) = (*LESS SO M1 idx* ∧ *LESS SO M2 idx*)
| *wf0 idx* (*Suc-SO br bl M1 M2*) = (*LESS SO M1 idx* ∧ *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* ∧ *LESS SO M2 idx*)
| *wf0 idx* (*In - m M*) = (*LESS FO m idx* ∧ *LESS SO M idx*)
| *wf0 idx* (*Eq-Max - m M*) = (*LESS FO m idx* ∧ *LESS SO M idx*)
| *wf0 idx* (*Eq-Min - m M*) = (*LESS FO m idx* ∧ *LESS SO M idx*)
| *wf0 idx* (*Eq-Union M1 M2 M3*) = (*LESS SO M1 idx* ∧ *LESS SO M2 idx* ∧ *LESS SO M3 idx*)
| *wf0 idx* (*Eq-Inter M1 M2 M3*) = (*LESS SO M1 idx* ∧ *LESS SO M2 idx* ∧ *LESS SO M3 idx*)
| *wf0 idx* (*Eq-Diff M1 M2 M3*) = (*LESS SO M1 idx* ∧ *LESS SO M2 idx* ∧ *LESS SO M3 idx*)
| *wf0 idx* (*Disjoint M1 M2*) = (*LESS SO M1 idx* ∧ *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* ≡ *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* ∨ *i* = *m2*)
| *find0 FO i* (*Plus-FO - m1 m2 -*) = (*i* = *m1* ∨ *i* = *m2*)
| *find0 FO i* (*Eq-FO - m1 m2*) = (*i* = *m1* ∨ *i* = *m2*)
| *find0 SO i* (*Eq-SO M1 M2*) = (*i* = *M1* ∨ *i* = *M2*)
| *find0 SO i* (*Suc-SO - - M1 M2*) = (*i* = *M1* ∨ *i* = *M2*)
| *find0 SO i* (*Empty M*) = (*i* = *M*)
| *find0 SO i* (*Singleton M*) = (*i* = *M*)
| *find0 SO i* (*Subset M1 M2*) = (*i* = *M1* ∨ *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* ∨ *i* = *M2* ∨ *i* = *M3*)
| *find0 SO i* (*Eq-Inter M1 M2 M3*) = (*i* = *M1* ∨ *i* = *M2* ∨ *i* = *M3*)
| *find0 SO i* (*Eq-Diff M1 M2 M3*) = (*i* = *M1* ∨ *i* = *M2* ∨ *i* = *M3*)
| *find0 SO i* (*Disjoint M1 M2*) = (*i* = *M1* ∨ *i* = *M2*)
| *find0 SO i* (*Eq-Presb - M -*) = (*i* = *M*)
| *find0 - - -* = *False*

abbreviation *decr0 ord k* ≡ *map-atomic* (*case-order* (*dec k*) *id ord*) (*case-order id* (*dec k*) *ord*)

lemma *sum-pow2-image-Suc*:
  finite X ⟹ *sum* ((^) (*2* :: *nat*)) (*Suc* ‘ *X*) = *2* ∗ *sum* ((^) *2*) *X*
  by (*induct X rule*: *finite-induct*) (*auto intro*: *trans*[*OF sum.insert*])

lemma *sum-pow2-insert0*:
  ⟦*finite X*; *0* ∉ *X*⟧ ⟹ *sum* ((^) (*2* :: *nat*)) (*insert 0 X*) = *Suc* (*sum* ((^) *2*) *X*)
  by (*induct X rule*: *finite-induct*) (*auto intro*: *trans*[*OF sum.insert*])

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

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

lemma *sum-pow2-inj*:
⟦*finite X*; *finite Y*; ($\sum x{\in}X.\ 2 \,\widehat{}\, x$ :: *nat*) = ($\sum x{\in}Y.\ 2 \,\widehat{}\, x$)⟧ ⟹ $X = Y$
(**is** - ⟹ - ⟹ *?f X* = *?f Y* ⟹ -)
proof (*induct X arbitrary*: *Y rule*: *finite-linorder-max-induct*)
case (*insert x X*)
from *insert*(*2*) **have** *?f X* ≤ *?f* {*0* ..< *x*} **by** (*intro sum-mono2*) *auto*
also have … < *2* ^ *x* **by** (*induct x*) *simp-all*
finally have *?f X* < *2* ^ *x* **.**
moreover from *insert*(*1*,*2*) **have** ∗: *?f X* + *2* ^ *x* = *?f Y*
using *trans*[*OF sym*[*OF insert*(*5*)] *sum.insert*] **by** *auto*
ultimately have *?f Y* < *2* ^ *Suc x* **by** *simp*

have ∀ *y* ∈ *Y*. *y* ≤ *x*
proof (*rule ccontr*)
assume ¬ (∀ *y*∈*Y*. *y* ≤ *x*)
then obtain *y* **where** *y* ∈ *Y Suc x* ≤ *y* **by** *auto*
from *this*(*2*) **have** *2* ^ *Suc x* ≤ (*2* ^ *y* :: *nat*) **by** (*intro power-increasing*) *auto*
also from ‹*y* ∈ *Y*› *insert*(*4*) **have** … ≤ *?f Y* **by** (*metis order.refl sum.remove trans-le-add1*)
finally show *False* **using** ‹*?f Y* < *2* ^ *Suc x*› **by** *simp*
qed

{ assume *x* ∉ *Y*
with ‹∀ *y* ∈ *Y*. *y* ≤ *x*› **have** *?f Y* ≤ *?f* {*0* ..< *x*} **by** (*intro sum-mono2*) (*auto simp*: *le-less*)
also have … < *2* ^ *x* **by** (*induct x*) *simp-all*
finally have *?f Y* < *2* ^ *x* **.**
with ∗ **have** *False* **by** *auto*
}
then have *x* ∈ *Y* **by** *blast*

from *insert*(*4*) **have** *?f* (*Y* − {*x*}) + *2* ^ *x* = *?f* (*insert x* (*Y* − {*x*}))**by** (*subst sum.insert*) *auto*
also have … = *?f X* + *2* ^ *x* **unfolding** ∗ **using** ‹*x* ∈ *Y*› **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 ‹*x* ∈ *Y*› **show** *?case* **by** *auto*
qed *simp*

lemma *finite-pow2-eq*:
fixes *n* :: *nat*
shows *finite* {*i*. *2* ^ *i* = *n*}
proof −
have ((^) *2*) ‘ {*i*. *2* ^ *i* = *n*} ⊆ {*n*} **by** *auto*
then have *finite* (((^) (*2* :: *nat*)) ‘ {*i*. *2* ^ *i* = *n*}) **by** (*rule finite-subset*) *blast*
then show *finite* {*i*. *2* ^ *i* = *n*} **by** (*rule finite-imageD*) (*auto simp*: *inj-on-def*)

qed

lemma *finite-pow2-le*[*simp*]:
  fixes *n* :: *nat*
  shows *finite* $\{i.\ 2 \mathbin{\hat{}} i \leq n\}$
  by (*induct n*) (*auto simp*: *le-Suc-eq finite-pow2-eq*)

lemma *le-pow2*[*simp*]: $x \leq y \Longrightarrow x \leq 2 \mathbin{\hat{}} y$
  by (*induct x arbitrary*: *y*) (*force simp add*: *Suc-le-eq order.strict-iff-order*)+

lemma *ld-bounded*: *Max* $\{i.\ 2 \mathbin{\hat{}} i \leq Suc\ n\} \leq Suc\ n$ (**is** $?m \leq Suc\ n$)
proof −
  have $?m \leq 2 \mathbin{\hat{}} ?m$ **by** (*rule le-pow2*) *simp*
  moreover
  have $?m \in \{i.\ 2 \mathbin{\hat{}} i \leq Suc\ n\}$ **by** (*rule Max-in*) (*auto intro*: *exI*[*of - 0*])
  then have $2 \mathbin{\hat{}} ?m \leq Suc\ n$ **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* ⇒ *Length* $\mathfrak{A} = i$ | - ⇒ *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* ⇒ *False* | *Some True* ⇒ $P2 = \{||\}$ | *Some False* ⇒ $P1 \neq$
$\{||\}$)
   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* ⇒ $P1 = P2$ | *Some i* ⇒ $P2 \neq \{||\} \wedge$ *fMin P2* + *i* =
Length $\mathfrak{A}$ | - ⇒ *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* |'| $M2^{\mathfrak{A}}SO$))
| *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*) = ($M1^{\mathfrak{A}}SO$ |⊆| $M2^{\mathfrak{A}}SO$)
| *satisfies0* $\mathfrak{A}$ (*In b m M*) =
   (*let* $P = m^{\mathfrak{A}}FO$ *in if* $P = \{||\}$ *then b else fMin P* |∈| $M^{\mathfrak{A}}SO$)

| *satisfies0* $\mathfrak{A}$ (*Eq-Max b m M*) =
(*let P1* = $m^{\mathfrak{A}}$*FO*; *P2* = $M^{\mathfrak{A}}$*SO in if b then P1* = {||}
else if P1 = {||} ∨ *P2* = {||} *then False else fMin P1* = *fMax P2*)
| *satisfies0* $\mathfrak{A}$ (*Eq-Min b m M*) =
(*let P1* = $m^{\mathfrak{A}}$*FO*; *P2* = $M^{\mathfrak{A}}$*SO in if P1* = {||} ∨ *P2* = {||} *then b* ∧ *P1* = *P2*
else fMin P1 = *fMin P2*)
| *satisfies0* $\mathfrak{A}$ (*Eq-Union M1 M2 M3*) = (*M1*$^{\mathfrak{A}}$*SO* = *M2*$^{\mathfrak{A}}$*SO* |∪| *M3*$^{\mathfrak{A}}$*SO*)
| *satisfies0* $\mathfrak{A}$ (*Eq-Inter M1 M2 M3*) = (*M1*$^{\mathfrak{A}}$*SO* = *M2*$^{\mathfrak{A}}$*SO* |∩| *M3*$^{\mathfrak{A}}$*SO*)
| *satisfies0* $\mathfrak{A}$ (*Eq-Diff M1 M2 M3*) = (*M1*$^{\mathfrak{A}}$*SO* = *M2*$^{\mathfrak{A}}$*SO* |−| *M3*$^{\mathfrak{A}}$*SO*)
| *satisfies0* $\mathfrak{A}$ (*Disjoint M1 M2*) = (*M1*$^{\mathfrak{A}}$*SO* |∩| *M2*$^{\mathfrak{A}}$*SO* = {||})
| *satisfies0* $\mathfrak{A}$ (*Eq-Presb i M n*) = ((($\sum$ *x*∈*fset* ($M^{\mathfrak{A}}$*SO*). *2* ^ *x*) = *n*) ∧
(*case i of None* ⇒ *True* | *Some l* ⇒ *Length* $\mathfrak{A}$ = *l*))

fun *lderiv0* **where**
lderiv0 (*bs1*, *bs2*) (*Fo m*) = (*if bs1* ! *m then FBool True else FBase* (*Fo m*))
| *lderiv0* (*bs1*, *bs2*) (*Eq-Const None m n*) = (*if n* = *0* ∧ *bs1* ! *m then FBool True*
else if n = *0* ∨ *bs1* ! *m then FBool False else FBase* (*Eq-Const None m* (*n* −
1)))
| *lderiv0* (*bs1*, *bs2*) (*Less None m1 m2*) = (*case* (*bs1* ! *m1*, *bs1* ! *m2*) *of*
(*False*, *False*) ⇒ *FBase* (*Less None m1 m2*)
| (*True*, *False*) ⇒ *FBase* (*Fo m2*)
| - ⇒ *FBool False*)|
lderiv0 (*bs1*, *bs2*) (*Eq-FO False m1 m2*) = (*case* (*bs1* ! *m1*, *bs1* ! *m2*) *of*
(*False*, *False*) ⇒ *FBase* (*Eq-FO False m1 m2*)
| (*True*, *True*) ⇒ *FBool True*
| - ⇒ *FBool False*)
| *lderiv0* (*bs1*, *bs2*) (*Plus-FO None m1 m2 n*) = (*if n* = *0*
then
(*case* (*bs1* ! *m1*, *bs1* ! *m2*) *of*
(*False*, *False*) ⇒ *FBase* (*Plus-FO None m1 m2 n*)
| (*True*, *True*) ⇒ *FBool True*
| - ⇒ *FBool False*)
else
(*case* (*bs1* ! *m1*, *bs1* ! *m2*) *of*
(*False*, *False*) ⇒ *FBase* (*Plus-FO None m1 m2 n*)
| (*False*, *True*) ⇒ *FBase* (*Eq-Const None m1* (*n* − *1*))
| - ⇒ *FBool False*))
| *lderiv0* (*bs1*, *bs2*) (*Eq-SO M1 M2*) =
(*if bs2* ! *M1* = *bs2* ! *M2 then FBase* (*Eq-SO M1 M2*) *else FBool False*)
| *lderiv0* (*bs1*, *bs2*) (*Suc-SO False bl M1 M2*) = (*if bl* = *bs2* ! *M1*
then FBase (*Suc-SO False* (*bs2* ! *M2*) *M1 M2*) *else FBool False*)
| *lderiv0* (*bs1*, *bs2*) (*Empty M*) = (*case bs2* ! *M of*
True ⇒ *FBool False*
| *False* ⇒ *FBase* (*Empty M*))
| *lderiv0* (*bs1*, *bs2*) (*Singleton M*) = (*case bs2* ! *M of*
True ⇒ *FBase* (*Empty M*)
| *False* ⇒ *FBase* (*Singleton M*))
| *lderiv0* (*bs1*, *bs2*) (*Subset M1 M2*) = (*case* (*bs2* ! *M1*, *bs2* ! *M2*) *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 ⟹ ld n = Max {i. 2 ^ i ≤ n}
proof (safe intro!: Max-eqI[symmetric])
  assume n > 0 then show 2 ^ ld n ≤ n by (induct n rule: ld.induct) auto
next
  fix y
  assume 2 ^ y ≤ n
  then show y ≤ ld n
  proof (induct n arbitrary: y rule: ld.induct)
    case (3 z)
    then have y − 1 ≤ ld (Suc (Suc z) 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)
```

| - ⇒ *FBase* (*In b m M*))
| *rderiv0* (*bs1*, *bs2*) (*Eq-Max b m M*) = (*case* (*bs1* ! *m*, *bs2* ! *M*) *of*
(*True*, *True*) ⇒ *if b then FBool False else FBase* (*Eq-Max True m M*)
| (*True*, *False*) ⇒ *if b then FBool False else FBase* (*Eq-Max False m M*)
| (*False*, *True*) ⇒ *if b then FBase* (*Eq-Max True m M*) *else FBool False*
| (*False*, *False*) ⇒ *FBase* (*Eq-Max b m M*))
| *rderiv0* (*bs1*, *bs2*) (*Eq-Min b m M*) = (*case* (*bs1* ! *m*, *bs2* ! *M*) *of*
(*True*, *True*) ⇒ *FBase* (*Eq-Min True m M*)
| (*False*, *False*) ⇒ *FBase* (*Eq-Min b m M*)
| - ⇒ *FBase* (*Eq-Min False m M*))
| *rderiv0* (*bs1*, *bs2*) (*Eq-Union M1 M2 M3*) = (*if bs2* ! *M1* = (*bs2* ! *M2* ∨ *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* ∧ *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* ∧ ¬ *bs2* ! *M3*)
then FBase (*Eq-Diff M1 M2 M3*) *else FBool False*)
| *rderiv0* (*bs1*, *bs2*) (*Disjoint M1 M2*) =
(*if bs2* ! *M1* ∧ *bs2* ! *M2 then FBool False else FBase* (*Disjoint M1 M2*))
| *rderiv0* (*bs1*, *bs2*) (*Eq-Presb l M n*) = (*case l of*
None ⇒ *if bs2* ! *M then*
if n = *0 then FBool False*
else let l = *ld n in FBase* (*Eq-Presb* (*Some l*) *M* (*n* − *2* ^ *l*))
else FBase (*Eq-Presb l M n*)
| *Some 0* ⇒ *FBool False*
| *Some* (*Suc l*) ⇒ *if bs2* ! *M* ∧ *n* ≥ *2* ^ *l then FBase* (*Eq-Presb* (*Some l*) *M* (*n* − *2* ^ *l*))
else if ¬ *bs2* ! *M* ∧ *n* < *2* ^ *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* ⇒ *False* | *Some b* ⇒ *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* ∧ (*l* = *Some 0* ∨ *l* = *None*))

definition *restrict ord P* = (*case ord of FO* ⇒ *P* ≠ {||} | *SO* ⇒ *True*)
definition *Restrict ord i* = (*case ord of FO* ⇒ *FBase* (*Fo i*) | *SO* ⇒ *FBool True*)

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

global-interpretation *WS1S*: *Formula SUC LESS assigns nvars Extend CONS SNOC Length*
extend size-atom zero σ 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* ⇒ -
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 Restrict* :: - ⇒ *formula*
and *RESTRICT* = *Formula-Operations.RESTRICT Restrict 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 :: *idx* ⇒ *formula* ⇒ -
and *ws1s-wf* = *Formula-Operations.wf SUC* (*wf0* :: *idx* ⇒ *ws1s* ⇒ -)
and *ws1s-lformula* = *Formula-Operations.lformula lformula0* :: *formula* ⇒ -
and *check-eqv* = *λidx*. *DAs.check-eqv*
(*σ idx*) (*λφ*. *norm* (*RESTRICT φ*) :: (*ws1s*, *order*) *aformula*)
(*λa φ*. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) *φ*))
(*final idx*) (*λφ* :: *formula*. *ws1s-wf idx φ* ∧ *ws1s-lformula φ*)
(*σ idx*) (*λφ*. *norm* (*RESTRICT φ*) :: (*ws1s*, *order*) *aformula*)
(*λa φ*. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) *φ*))
(*final idx*) (*λφ* :: *formula*. *ws1s-wf idx φ* ∧ *ws1s-lformula φ*) (=)
and *bounded-check-eqv* = *λidx*. *DAs.check-eqv*
(*σ idx*) (*λφ*. *norm* (*RESTRICT φ*) :: (*ws1s*, *order*) *aformula*)
(*λa φ*. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) *φ*))
nullable (*λφ* :: *formula*. *ws1s-wf idx φ* ∧ *ws1s-lformula φ*)
(*σ idx*) (*λφ*. *norm* (*RESTRICT φ*) :: (*ws1s*, *order*) *aformula*)
(*λa φ*. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) *φ*))
nullable (*λφ* :: *formula*. *ws1s-wf idx φ* ∧ *ws1s-lformula φ*) (=)
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*)+)
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*: $lformula0\ a \lor len\ P \leq Length\ \mathfrak{A}$
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 rule*: *lformula0.induct*)
(*auto simp*: *dec-def split*: *if-splits order.split option.splits bool.splits*) — slow
next
note *dec-def*[*simp*]
assume $len\ P \leq Length\ \mathfrak{A}$
with $*$ **show** *?thesis*
proof (*induct a*)
case *Fo* **then show** *?case* **by** (*cases k*) *auto*
next
case *Eq-Const* **then show** *?case*
by (*cases k*) (*auto simp*: *Let-def len-def split*: *if-splits option.splits*)
next
case *Less* **then show** *?case* **by** (*cases k*) *auto*
next
case *Plus-FO* **then show** *?case*
by (*cases k*) (*auto simp*: *max-def len-def Let-def split*: *option.splits nat.splits*)
next
case *Eq-FO* **then show** *?case* **by** (*cases k*) *auto*
next
case *Eq-SO* **then show** *?case* **by** (*cases k*) *auto*
next
case (*Suc-SO br bl M1 M2*) **then show** *?case*
by (*cases k*) (*auto simp*: *max-def len-def*)
next
case *Empty* **then show** *?case* **by** (*cases k*) *auto*
next
case *Singleton* **then show** *?case* **by** (*cases k*) *auto*
next
case *Subset* **then show** *?case* **by** (*cases k*) *auto*
next
case *In* **then show** *?case* **by** (*cases k*) *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 $\#_V$ $\mathfrak{A}$ = *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 $?f = sum\ ((\widehat{\ })\ (2 :: nat))$
case (*Eq-Presb l M n*)
moreover
let $?M = fset\ (M^{\mathfrak{A}}SO)$ **and** $?L = Length\ \mathfrak{A}$
have $?f\ (insert\ ?L\ ?M) = 2\ \widehat{}\ ?L + ?f\ ?M$
by (*subst sum.insert*) *auto*
moreover have $n > 0 \Longrightarrow 2\ \widehat{}\ Max\ \{i.\ 2\ \widehat{}\ i \leq n\} \leq n$
using *Max-in*[*of* $\{i.\ 2\ \widehat{}\ i \leq n\}$, *simplified*, *OF exI*[*of* - *0*]] **by** *auto*
moreover
{ have $?f\ ?M \leq ?f\ \{0\ ..<\ ?L\}$ **by** (*rule sum-mono2*) *auto*
also have $\ldots = 2\ \widehat{}\ ?L - 1$ **by** (*rule sum-pow2-upto*)
also have $\ldots < 2\ \widehat{}\ ?L$ **by** *simp*
finally have $?f\ ?M < 2\ \widehat{}\ ?L$ **.**
}
moreover have $Max\ \{i.\ 2\ \widehat{}\ i \leq 2\ \widehat{}\ ?L + ?f\ ?M\} = ?L$
proof (*intro Max-eqI*, *safe*)
fix *y* **assume** $2\ \widehat{}\ y \leq 2\ \widehat{}\ ?L + ?f\ ?M$

{ **assume** $?L < y$
then have $(2 :: nat)$ ^ $?L + 2$ ^ $?L \le 2$ ^ y
by (*cases y*) (*auto simp*: *less-Suc-eq-le numeral-eq-Suc add-le-mono*)
also note ‹2 ^ $y \le 2$ ^ $?L + ?f\ ?M$›
finally have 2 ^ $?L \le ?f\ ?M$ **by** *simp*
with ‹$?f\ ?M < 2$ ^ $?L$› **have** *False* **by** *auto*
} **then show** $y \le ?L$ **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** $\mathfrak{A}$ $\mathfrak{B}$ :: *interp*
assume *wf0* ($\#_V$ $\mathfrak{B}$) *a* $\#_V\ \mathfrak{A} = \#_V\ \mathfrak{B}$ ($\bigwedge m\ k.$ *LESS k m* ($\#_V$ $\mathfrak{B}$) $\Longrightarrow m^{\mathfrak{A}}k = m^{\mathfrak{B}}k$) *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 Φ :: - $\Rightarrow$ (*ws1s*, *order*) *aformula set*
where Φ *a* =
(*case a of*
Fo m $\Rightarrow$ {*FBase* (*Fo m*), *FBool True*}
| *Eq-Const None m n* $\Rightarrow$ {*FBase* (*Eq-Const None m i*) | *i* . $i \le n$} $\cup$ {*FBool True*, *FBool False*}
| *Less None m1 m2* $\Rightarrow$ {*FBase* (*Less None m1 m2*), *FBase* (*Fo m2*), *FBool True*, *FBool False*}
| *Plus-FO None m1 m2 n* $\Rightarrow$ {*FBase* (*Eq-Const None m1 i*) | *i* . $i \le n$} $\cup$
{*FBase* (*Plus-FO None m1 m2 n*), *FBool True*, *FBool False*}
| *Eq-FO False m1 m2* $\Rightarrow$ {*FBase* (*Eq-FO False m1 m2*), *FBool True*, *FBool False*}
| *Eq-SO M1 M2* $\Rightarrow$ {*FBase* (*Eq-SO M1 M2*), *FBool False*}
| *Suc-SO False bl M1 M2* $\Rightarrow$ {*FBase* (*Suc-SO False True M1 M2*), *FBase* (*Suc-SO False False M1 M2*),
FBool False}
| *Empty M* $\Rightarrow$ {*FBase* (*Empty M*), *FBool False*}
| *Singleton M* $\Rightarrow$ {*FBase* (*Singleton M*), *FBase* (*Empty M*), *FBool False*}
| *Subset M1 M2* $\Rightarrow$ {*FBase* (*Subset M1 M2*), *FBool False*}
| *In False i I* $\Rightarrow$ {*FBase* (*In False i I*), *FBool True*, *FBool False*}
| *Eq-Max False m M* $\Rightarrow$ {*FBase* (*Eq-Max False m M*), *FBase* (*Empty M*), *FBool False*}
| *Eq-Min False m M* $\Rightarrow$ {*FBase* (*Eq-Min False m M*), *FBool True*, *FBool False*}
| *Eq-Union M1 M2 M3* $\Rightarrow$ {*FBase* (*Eq-Union M1 M2 M3*), *FBool False*}
| *Eq-Inter M1 M2 M3* $\Rightarrow$ {*FBase* (*Eq-Inter M1 M2 M3*), *FBool False*}
| *Eq-Diff M1 M2 M3* $\Rightarrow$ {*FBase* (*Eq-Diff M1 M2 M3*), *FBool False*}
| *Disjoint M1 M2* $\Rightarrow$ {*FBase* (*Disjoint M1 M2*), *FBool False*}
| *Eq-Presb None M n* $\Rightarrow$ {*FBase* (*Eq-Presb None M i*) | *i* . $i \le n$} $\cup$ {*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** *auto*
moreover have $\bigwedge x\ \varphi.\ \varphi \in \Phi\ a \Longrightarrow d\ x\ \varphi \in \Phi\ a$
by (*auto simp*: *d-def split*: *atomic.splits list.splits bool.splits if-splits option.splits*)
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*) ∈ 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*: *finite-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* .
$j \leq$ (*case i of Some i* ⇒ *max i n* | - ⇒ *n*) ∧ $l \leq$ (*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 $\bigwedge x\ \varphi.\ \varphi \in \Phi\ a \Longrightarrow d\ x\ \varphi \in \Phi\ a$
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 $\bigwedge\varphi.\ \varphi \in \Phi\ a \Longrightarrow$ *fold d xs* $\varphi \in \Phi\ a$
by (*induct xs*) *auto*
ultimately have *fold d xs* (*FBase a*) ∈ 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*: *finite-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* $\mathfrak{A}$ *k P r*
assume $i^{\mathfrak{A}}k = P$
then show *restrict k P* ⟷
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 dest!: *CONS-surj*
dest: *fMax-ge fMax-ffilter-less fMax-boundedD fsubset-fsingletonD*
split!: *order.splits if-splits*)

lemma *check-eqv-code*[*code*]: *check-eqv 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* (λ*a*. (*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-eqv-def WS1S.check-eqv-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-eqv-sound*:
$[\![\#_V\ \mathfrak{A} = idx;\ check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow (WS1S.sat\ \mathfrak{A}\ \varphi \longleftrightarrow WS1S.sat\ \mathfrak{A}\ \psi)$
unfolding *check-eqv-def* **by** (*rule WS1S.check-eqv-soundness*)

lemma *bounded-check-eqv-sound*:
$[\![\#_V\ \mathfrak{A} = idx;\ bounded\text{-}check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow (WS1S.sat_b\ \mathfrak{A}\ \varphi \longleftrightarrow WS1S.sat_b\ \mathfrak{A}\ \psi)$
unfolding *bounded-check-eqv-def* **by** (*rule WS1S.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 × 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* ∧ *LESS FO m2 idx*)
| *wf0 idx* (*In m M*) = (*LESS FO m idx* ∧ *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* ≡ *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* ∨ *i* = *m2*)
| *find0 FO i* (*In m -*) = (*i* = *m*)
| *find0 SO i* (*In - M*) = (*i* = *M*)
| *find0 - - -* = *False*

abbreviation *decr0 ord k* ≡ *map-atomic* (*case-order* (*dec k*) *id ord*) (*case-order id* (*dec k*) *ord*)

primrec *satisfies0* **where**
  satisfies0 $\mathfrak{A}$ (*Fo m*) = (∃ *x*. $m^{\mathfrak{A}}$*FO* = {|*x*|})
| *satisfies0* $\mathfrak{A}$ (*Z m*) = ($m^{\mathfrak{A}}$*FO* = {||})
| *satisfies0* $\mathfrak{A}$ (*Less m1 m2*) =
  (*let P1* = *m1*$^{\mathfrak{A}}$*FO*; *P2* = *m2*$^{\mathfrak{A}}$*FO in if* ¬(∃ *x*. *P1* = {|*x*|}) ∨ ¬(∃ *x*. *P2* = {|*x*|})
  then False
  else fthe-elem P1 < *fthe-elem P2*)
| *satisfies0* $\mathfrak{A}$ (*In m M*) =
  (*let P* = $m^{\mathfrak{A}}$*FO in if* ¬(∃ *x*. *P* = {|*x*|}) *then False else fMin P* |∈| $M^{\mathfrak{A}}$*SO*)

fun *lderiv0* **where**
  lderiv0 (*bs1*, *bs2*) (*Fo m*) = (*if bs1* ! *m then FBase* (*Z m*) *else FBase* (*Fo m*))
| *lderiv0* (*bs1*, *bs2*) (*Z m*) = (*if bs1* ! *m then FBool False else FBase* (*Z m*))
| *lderiv0* (*bs1*, *bs2*) (*Less m1 m2*) = (*case* (*bs1* ! *m1*, *bs1* ! *m2*) *of*
    (*False*, *False*) ⇒ *FBase* (*Less m1 m2*)
  | (*True*, *False*) ⇒ *FAnd* (*FBase* (*Z m1*)) (*FBase* (*Fo m2*))
  | - ⇒ *FBool False*)
| *lderiv0* (*bs1*, *bs2*) (*In m M*) = (*case* (*bs1* ! *m*, *bs2* ! *M*) *of*
    (*False*, -) ⇒ *FBase* (*In m M*)
  | (*True*, *True*) ⇒ *FBase* (*Z m*)
  | - ⇒ *FBool False*)

primrec *rev* **where**
  rev (*Fo m*) = *Fo m*
| *rev* (*Z m*) = *Z m*
| *rev* (*Less m1 m2*) = *Less m2 m1*
| *rev* (*In m M*) = *In m M*

abbreviation *rderiv0 v* ≡ *map-aformula rev id o lderiv0 v o rev*

primrec *nullable0* **where**
  nullable0 (*Fo m*) = *False*
| *nullable0* (*Z m*) = *True*
| *nullable0* (*Less m1 m2*) = *False*
| *nullable0* (*In m M*) = *False*

lemma *fimage-Suc-fsubset0*[*simp*]: *Suc* |\`| *A* |⊆| {|*0*|} ⟷ *A* = {||}
  by *blast*

lemma *fsubset-singleton-iff*: *A* |⊆| {|*x*|} ⟷ *A* = {||} ∨ *A* = {|*x*|}
  by *blast*

definition *restrict ord P* = (*case ord of FO* ⇒ ∃ *x*. *P* = {|*x*|} | *SO* ⇒ *True*)
definition *Restrict ord i* = (*case ord of FO* ⇒ *FBase* (*Fo i*) | *SO* ⇒ *FBool True*)

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

global-interpretation *WS1S-Alt*: *Formula SUC LESS assigns nvars Extend CONS SNOC Length*
extend size-atom zero σ *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* ⇒ -
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 Restrict* :: - ⇒ *formula*
and *RESTRICT* = *Formula-Operations.RESTRICT Restrict FV0*
and *deriv* = λ*d0* (*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 :: *idx* ⇒ *formula* ⇒ -
and *ws1s-wf* = *Formula-Operations.wf SUC* (*wf0* :: *idx* ⇒ *ws1s* ⇒ -)
and *ws1s-lformula* = *Formula-Operations.lformula lformula0* :: *formula* ⇒ -
and *check-eqv* = λ*idx*. *DAs.check-eqv*
(σ *idx*) (λφ. *norm* (*RESTRICT* φ) :: (*ws1s*, *order*) *aformula*)
(λ*a* φ. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
(*final idx*) (λφ :: *formula*. *ws1s-wf idx* φ ∧ *ws1s-lformula* φ)
(σ *idx*) (λφ. *norm* (*RESTRICT* φ) :: (*ws1s*, *order*) *aformula*)
(λ*a* φ. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
(*final idx*) (λφ :: *formula*. *ws1s-wf idx* φ ∧ *ws1s-lformula* φ) (=)
and *bounded-check-eqv* = λ*idx*. *DAs.check-eqv*
(σ *idx*) (λφ. *norm* (*RESTRICT* φ) :: (*ws1s*, *order*) *aformula*)
(λ*a* φ. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
nullable (λφ :: *formula*. *ws1s-wf idx* φ ∧ *ws1s-lformula* φ)
(σ *idx*) (λφ. *norm* (*RESTRICT* φ) :: (*ws1s*, *order*) *aformula*)
(λ*a* φ. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
nullable (λφ :: *formula*. *ws1s-wf idx* φ ∧ *ws1s-lformula* φ) (=)
and *automaton* = *DA.automaton*
(λ*a* φ. *norm* (*deriv lderiv0* (*a* :: *atom*) φ :: *formula*))
proof
fix *k idx* **and** *a* :: *ws1s* **and** *l* **assume** *wf0* (*SUC k idx*) *a LESS k l* (*SUC k idx*) ¬ *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* $\#_V$ $\mathfrak{A}$ = *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* $\#_V$ $\mathfrak{A}$ = *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* $\#_V$ $\mathfrak{A}$ = *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 finsertCI less-not-sym*)
apply *force*
apply (*metis assigns-less-Length finsertCI 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*. *LESS k m* ($\#_V$ $\mathfrak{B}$) $\Longrightarrow$ $m^{\mathfrak{A}}k$ = $m^{\mathfrak{B}}k$) *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 Φ :: - $\Rightarrow$ (*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 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* $\in \Phi$ *a* **by** (*cases a*) *auto*
moreover have $\bigwedge x\ \varphi.\ \varphi \in \Phi\ a \Longrightarrow d\ x\ \varphi \in \Phi\ a$
by (*auto simp*: *d-def split*: *atomic.splits list.splits bool.splits if-splits option.splits*)
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* (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*: *finite-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* $\in \Phi$ *a* **by** (*auto split*: *atomic.splits option.splits*)
moreover have $\bigwedge x\ \varphi.\ \varphi \in \Phi\ a \Longrightarrow d\ x\ \varphi \in \Phi\ a$
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 $\bigwedge\varphi.\ \varphi \in \Phi\ a \Longrightarrow$ *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* (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*: *finite-subset*)
next
fix *k l* **and** *a* :: *ws1s*
show *find0 k l a* $\longleftrightarrow$ *l* $\in$ *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* $\in$ *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* $\mathfrak{A}$ *k P r*
assume $i^{\mathfrak{A}}k = P$
then show *restrict k 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-ffilter-less fMax-boundedD fsubset-fsingletonD*
split: *order.splits if-splits*)

lemma *check-eqv-code*[*code*]: *check-eqv idx r s* =
((*ws1s-wf idx r* $\wedge$ *ws1s-lformula r*) $\wedge$ (*ws1s-wf idx s* $\wedge$ *ws1s-lformula s*) $\wedge$
(*case rtrancl-while* (λ(*p*, *q*). *final idx p* = *final idx q*)
(λ(*p*, *q*). *map* (λ*a*. (*norm* (*deriv lderiv0 a p*), *norm* (*deriv lderiv0 a q*))) (σ *idx*))
(*norm* (*RESTRICT r*), *norm* (*RESTRICT s*)) *of*
None $\Rightarrow$ *False*
| *Some* ([], *x*) $\Rightarrow$ *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* 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-eqv-sound*:
$[\![\#_V\ \mathfrak{A} = idx;\ check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow (WS1S\text{-}Alt.sat\ \mathfrak{A}\ \varphi \longleftrightarrow WS1S\text{-}Alt.sat\ \mathfrak{A}\ \psi)$
unfolding *check-eqv-def* **by** (*rule WS1S-Alt.check-eqv-soundness*)

lemma *bounded-check-eqv-sound*:
$[\![\#_V\ \mathfrak{A} = idx;\ bounded\text{-}check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow (WS1S\text{-}Alt.sat_b\ \mathfrak{A}\ \varphi \longleftrightarrow WS1S\text{-}Alt.sat_b\ \mathfrak{A}\ \psi)$
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 × 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* ⟷ *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* ≤ *l* ⟷ *x* < *2* ^ *l*
proof (*induct x arbitrary*: *l rule*: *len.induct*)
fix *l* **show** (*len* (*Suc 0*) ≤ *l*) = (*Suc 0* < *2* ^ *l*)
proof (*cases l*)
case *Suc* **then show** *?thesis* **using** *le-less* **by** *fastforce*
qed *simp*
next
fix *v l* **assume** ⋀*l*. (*len* (*Suc* (*Suc v*) *div 2*) ≤ *l*) = (*Suc* (*Suc v*) *div 2* < *2* ^ *l*)
then show (*len* (*Suc* (*Suc v*)) ≤ *l*) = (*Suc* (*Suc v*) < *2* ^ *l*)
by (*cases l*) (*simp-all*, *linarith*)
qed *simp*

lemma *len-pow2*[*simp*]: *len* (*2* ^ *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* ^ *len x*
by (*induct x rule*: *len.induct*) *auto*

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

lemma *len-mono*[*simp*]: $x \le y \Longrightarrow len\ x \le len\ y$
unfolding *len-le-iff* **using** *less-pow2-len*[*of y*] **by** *linarith*

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

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

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

abbreviation (*input*) *zero n* ≡ *replicate n False*
abbreviation (*input*) *SUC* ≡ λ-::*unit*. *Suc*
definition *test-bit m n* ≡ (*m* :: *nat*) *div 2* ^ *n mod 2* = *1*
lemma *test-bit-eq-bit*: ‹*test-bit* = *bit*›
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* ≡ (*m* :: *nat*) *div 2*
definition *upshift m* ≡ (*m* :: *nat*) ∗ *2*
lemma *set-bit-def*: *set-bit n m* ≡ *m* + (*if* ¬ *test-bit m n then 2* ^ *n else* (*0* :: *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* ≡ (*m* :: *nat*) *mod 2* ^ *n*
lemma *cut-bits-eq-take-bit*: ‹*cut-bits* = *take-bit*›
by (*simp add*: *fun-eq-iff cut-bits-def take-bit-eq-mod*)

typedef *interp* = {(*n* :: *nat*, *xs* :: *nat list*). ∀ *x* ∈ *set xs*. *len x* ≤ *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* :: *nat* ⇒ *interp* ⇒ *unit* ⇒ *value* (‹-ˉ-› [*900*, *999*, *999*] *999*) **is**
λ*n* (-, *I*) -. *if n* < *length I then I* ! *n else 0* .

lift-definition *nvars* :: *interp* ⇒ *nat* (‹$\#_V$ -› [*1000*] *900*) **is**

λ(-, I). *length I* **.**

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

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

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

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

definition *extend* :: *unit* $\Rightarrow$ *bool* $\Rightarrow$ *atom* $\Rightarrow$ *atom* **where**
extend - *b bs* $\equiv$ *b* # *bs*

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

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

lemma *FV0-code*[*code*]:
FV0 x (*Eq is i off*) = *Option.these* (*set* (*map-index* (λi x. *if* $x = 0$ *then None else Some i*) *is*))
unfolding *FV0-def* **by** (*force simp*: *Option.these-def image-iff*)

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

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

primrec *decr0* **where**
decr0 (-::*unit*) *k* (*Eq is i d*) = *Eq* (*take k is* @ *drop* (*Suc k*) *is*) *i d*

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

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

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

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

code-pred *lformula0* **.**

fun *lderiv0* :: *bool list* ⇒ *presb* ⇒ *formula* **where**
lderiv0 bs (*Eq is i d*) = (*if d* ≠ *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* ⇒ *presb* ⇒ *formula* **where**
rderiv0 bs (*Eq is i d*) =
(*let*
l = − *sum-list* [*i*. *i* ← *is*, *i* < *0*];
h = − *sum-list* [*i*. *i* ← *is*, *i* > *0*];
d′ = *scalar-product bs is* + *2* ∗ *d*
in if d′ ∈ {*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 *σ* :: *nat* ⇒ *atom list* **where**
σ n = *List.n-lists n* [*True*, *False*]

named-theorems *Presb-simps*

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

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

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

lemma *ex-Length0*[*Presb-simps*]: ∃𝔄. *Length* 𝔄 = *0* ∧ $\#_V$ 𝔄 = *idx*
by (*transfer fixing*: *idx*) (*auto intro*: *exI*[*of* - *replicate idx 0*])

lemma *Extend-commute-safe*[*Presb-simps*]: ⟦*j* ≤ *i*; *i* < *Suc* ($\#_V$ 𝔄)⟧ ⟹
Extend k j (*Extend k i* 𝔄 *P*) *Q* = *Extend k* (*Suc i*) (*Extend k j* 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 ≠ *k′* ⟹ *Extend k j* (*Extend k′ i* 𝔄 *P*) *Q* = *Extend k′ i* (*Extend k j* 𝔄 *Q*) *P*
by *transfer auto*

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

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

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

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

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

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

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

lemma *Extend-SNOC*[*Presb-simps*]: $[\![\#_V\ \mathfrak{A} = length\ x;\ len\ P \le Length\ (SNOC\ x\ \mathfrak{A})]\!] \Longrightarrow$
$Extend\ k\ 0\ (SNOC\ x\ \mathfrak{A})\ P =$
$SNOC\ (extend\ k\ (test\text{-}bit\ P\ (Length\ \mathfrak{A}))\ x)\ (Extend\ k\ 0\ \mathfrak{A}\ (cut\text{-}bits\ (Length\ \mathfrak{A})\ P))$
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*:
$Suc\ (2 * x) = 2 * y \longleftrightarrow False$
$2 * y = Suc\ (2 * x) \longleftrightarrow False$
by *presburger*+

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

lemma *mod-2-Suc-iff*:
  $x\ mod\ 2 = Suc\ 0 \longleftrightarrow x = Suc\ (2 * (x\ div\ 2))$
  by *presburger+*

lemma *CONS-surj*[*Presb-simps*]: $Length\ \mathfrak{A} \neq 0 \Longrightarrow$
  $\exists x\ \mathfrak{B}.\ \mathfrak{A} = CONS\ x\ \mathfrak{B} \wedge \#_V\ \mathfrak{B} = \#_V\ \mathfrak{A} \wedge size\ x = \#_V\ \mathfrak{A}$
  by *transfer*
    (*auto simp*: *gr0-conv-Suc list-eq-iff-nth-eq len-le-iff split*: *if-splits*
    intro!: *exI*[*of - map* ($\lambda n.\ n\ mod\ 2 \neq 0$) -] *exI*[*of - map* ($\lambda n.\ n\ div\ 2$) -];
    auto simp: *mod-2-Suc-iff*)

lemma [*Presb-simps*]:
  length (*extend k b x*) = *Suc* (*length x*)
  downshift (*upshift P*) = *P*
  downshift (*set-bit 0 P*) = *downshift P*
  test-bit (*set-bit n P*) *n*
  ¬ *test-bit* (*upshift P*) *0*
  len P ≤ *p* ⟹ ¬ *test-bit P p*
  len (*cut-bits n P*) ≤ *n*
  len P ≤ *n* ⟹ *cut-bits n P* = *P*
  len (*upshift P*) = (*case len P of 0* ⇒ *0* | *Suc n* ⇒ *Suc* (*Suc n*))
  len (*downshift P*) = (*case len P of 0* ⇒ *0* | *Suc n* ⇒ *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*: $Suc\ 0\ div\ 2\ \hat{}\ i = (if\ i = 0\ then\ 1\ else\ 0)$
  by (*induct i*) (*auto simp*: *div-mult2-eq*)

lemma *set-unset-bit-preserves-len*:
  assumes $x\ div\ 2\ \hat{}\ m = 2 * q\ m < len\ x$
  shows $x + 2\ \hat{}\ m < 2\ \hat{}\ len\ x$
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*]: *len* (*set-bit m P*) = *max* (*Suc m*) (*len P*)
proof (*rule antisym*)
  show *len* (*set-bit m P*) ≤ *max* (*Suc m*) (*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 $P < 2\ \hat{}\ len\ (P + 2\ \hat{}\ m)$ **by** (*rule order.strict-trans2*[*OF less-pow2-len*])
auto

moreover have *m* < *len* (*P* + *2* ^ *m*) **by** (*rule order.strict-trans2*[*OF* - *len-mono*[*of* *2* ^ *m*]]) *auto*
ultimately show *max* (*Suc m*) (*len P*) ≤ *len* (*set-bit m P*)
by (*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* :: *nat*
shows *m* < *n* ⟹ *p mod 2* ^ *n div 2* ^ *m* = *p div 2* ^ *m mod 2* ^ (*n* − *m*)
by (*induct m arbitrary*: *p n*) (*auto simp*: *div-mult2-eq mod-mult2-eq Suc-less-eq2*)

lemma *irrelevant-set-bit*[*simp*]:
fixes *p m n* :: *nat*
assumes *n* ≤ *m*
shows (*p* + *2* ^ *m*) *mod 2* ^ *n* = *p mod 2* ^ *n*
proof −
from *assms* **obtain** *q* :: *nat* **where** *2* ^ *m* = *q* ∗ *2* ^ *n*
by (*metis le-add-diff-inverse mult.commute power-add*)
then show *?thesis* **by** *simp*
qed

lemma *mod-lemma*: ⟦ (*0*::*nat*) < *c*; *r* < *b* ⟧ ⟹ *b* ∗ (*q mod c*) + *r* < *b* ∗ *c*
by (*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* :: *nat*
assumes *m* < *n p div 2* ^ *m* = *2* ∗ *q*
shows (*p* + *2* ^ *m*) *mod 2* ^ *n* = *p mod 2* ^ *n* + *2* ^ *m*
proof −
have *p mod 2* ^ *n* + *2* ^ *m* < *2* ^ *n*
using *assms* **proof** (*induct m arbitrary*: *p n*)
case *0* **then show** *?case*
by (*auto simp*: *gr0-conv-Suc*)
(*metis One-nat-def Suc-eq-plus1 lessI mod-lemma numeral-2-eq-2 zero-less-numeral zero-less-power*)
next
case (*Suc m*)
from *Suc*(*1*)[*of n* − *1 p div 2*] *Suc*(*2*,*3*) **show** *?case*
by (*auto simp*: *div-mult2-eq mod-mult2-eq Suc-less-eq2*)
qed
with ‹*m* < *n*› **show** *?thesis* **by** (*subst mod-add-eq* [*symmetric*]) *auto*
qed

lemma *cut-bits-set-bit*[*Presb-simps*]: *cut-bits n* (*set-bit m p*) =
(*if n* ≤ *m then cut-bits n p else set-bit m* (*cut-bits n p*))
unfolding *cut-bits-def set-bit-def test-bit-def*
by (*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* $\Longrightarrow$ *l* < *Suc idx* $\Longrightarrow$ ¬ *find0 k l a* $\Longrightarrow$ *wf0 idx* (*decr0 k l a*)
by (*induct a*) *auto*

lemma *lformula0-decr0*[*Presb-simps*]: *lformula0 a* $\Longrightarrow$ *lformula0* (*decr0 k l a*)
by (*induct a*) (*auto elim*: *lformula0.cases intro*: *lformula0.intros*)

abbreviation *sat0-syn* (**infix** ‹⊨*0*› *65*) **where**
sat0-syn ≡ *satisfies0*
abbreviation *sat-syn* (**infix** ‹⊨› *65*) **where**
sat-syn ≡ *Formula-Operations.satisfies Extend Length satisfies0*
abbreviation *sat-bounded-syn* (**infix** ‹$\models_b$› *65*) **where**
sat-bounded-syn ≡ *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* ⇒ *x* ∗ *y* + *scalar-product xs ys* | [] ⇒ *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 ¬ *find0 k i a i* < *Suc* ($\#_V$ 𝔄) *lformula0 a* ∨ *len P* ≤ *Length* 𝔄
shows *Extend k i* 𝔄 *P* ⊨*0* *a* = 𝔄 ⊨*0* *decr0 k i a*
proof −
{ **fix** *is* :: *int list*
assume *is* ! *i* = *0*
with *assms*(*1*,*2*) **have** *eval-tm* (*Extend k i* 𝔄 *P*) *is* = *eval-tm* 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*$\in$*set ns*. *c* = *0* $\Longrightarrow$ *scalar-product ns 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*]: *Length* $\mathfrak{A}$ = *0* $\Longrightarrow$ *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*: *wf0* ($\#_V$ $\mathfrak{B}$) *a* $\Longrightarrow$ $\#_V$ $\mathfrak{A}$ = $\#_V$ $\mathfrak{B}$ $\Longrightarrow$ *lformula0 a* $\Longrightarrow$
  ($\bigwedge$*m k*. *m* < $\#_V$ $\mathfrak{B}$ $\Longrightarrow$ $m^{\mathfrak{A}}k$ = $m^{\mathfrak{B}}k$) $\Longrightarrow$ $\mathfrak{A}$ $\models$*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-lderiv0*[*Presb-simps*]:
  wf0 idx a $\Longrightarrow$ *lformula0 a* $\Longrightarrow$ *Formula-Operations.wf* (λ-. *Suc*) *wf0 idx* (*lderiv0 x a*)
   by (*induct a*) (*auto elim*: *lformula0.cases simp*: *Formula-Operations.wf.simps Let-def*)

lemma *lformula-lderiv0*[*Presb-simps*]:
  lformula0 a $\Longrightarrow$ *Formula-Operations.lformula lformula0* (*lderiv0 x a*)
  by (*induct a*)
   (*auto elim*: *lformula0.cases intro*: *lformula0.intros simp*: *Let-def Formula-Operations.lformula.simps*)

lemma *wf-rderiv0*[*Presb-simps*]:
  wf0 idx a $\Longrightarrow$ *Formula-Operations.wf* (λ-. *Suc*) *wf0 idx* (*rderiv0 x a*)
   by (*induct a*) (*auto elim*: *lformula0.cases simp*: *Formula-Operations.wf.simps Let-def*)

lemma *find0-FV0*[*Presb-simps*]: $[\![$*wf0 idx a*; *l* < *idx*$]\!]$ $\Longrightarrow$ *find0 k l a* = (*l* $\in$ *FV0 k a*)
  by (*induct a*) (*auto simp*: *FV0-def*)

lemma *FV0-less*[*Presb-simps*]: *wf0 idx a* $\Longrightarrow$ *v* $\in$ *FV0 k a* $\Longrightarrow$ *v* < *idx*
  by (*induct a*) (*auto simp*: *FV0-def*)

lemma *finite-FV0*[*Presb-simps*]: *finite* (*FV0 k a*)
  by (*induct a*) (*auto simp*: *FV0-def*)

lemma *finite-lderiv0*[*Presb-simps*]:
assumes *lformula0 a*
shows *finite* {φ. ∃*xs*. φ = *fold* (*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* ← *is*, *i* < *0*] **for** *is* :: *int list*
define *h* **where** *h is* = *sum-list* [*i*. *i* ← *is*, *i* > *0*] **for** *is* :: *int list*
define Φ **where** Φ *a* = (*case a of*
Eq is n z ⇒ {*FBase* (*Eq is i 0*) | *i* . *i* ∈ {*min* (− *h is*) *n* .. *max* (− *l is*) *n*}} ∪
{*FBool False* :: *formula*}) **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 simp*: *elim*!: *lformula0.cases*)
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* :: *int* **and** φ :: *formula*
assume *i* ∈ {*min* (− *h is*) *n*..*max* (− *l is*) *n*} φ = *FBase* (*presb.Eq is i 0*)
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 i 0*) |*i*. *i* ∈ {*min* (− *h is*) *n*..*max* (− *l is*) *n*}} ∪ {*FBool False*}
by (*auto simp*: *d-def Let-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 *finite-rderiv0*[*Presb-simps*]:
finite {φ. ∃*xs*. φ = *fold* (*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* ← *is*, *i* < *0*]**for** *is* :: *int list*
define *h* **where** *h is* = *sum-list* [*i*. *i* ← *is*, *i* > *0*]**for** *is* :: *int list*
define Φ **where** Φ *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 $\bigwedge x\ \varphi.\ \varphi \in \Phi\ a \Longrightarrow d\ x\ \varphi \in \Phi\ a$
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 $\bigwedge\varphi.\ \varphi \in \Phi\ a \Longrightarrow$ *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*: *length xs* = *length* (*bs* :: *bool list*) ⟹
scalar-product (*map-index* (λ*i n*. *2* ∗ *n* + *bs* ! *i*) *xs*) *is* =
scalar-product bs is + *2* ∗ *scalar-product xs is*
by (*induct is arbitrary*: *bs xs*) (*auto split*: *list.splits simp*: *algebra-simps*)

lemma *eval-tm-CONS*[*simp*]:
$[\![$*length is* ≤ $\#_V\ \mathfrak{A}$; $\#_V\ \mathfrak{A}$ = *length x*$]\!]$ ⟹
eval-tm (*CONS x* 𝔄) *is* = *scalar-product x is* + *2* ∗ *eval-tm* 𝔄 *is*
by *transfer* (*auto simp*: *scalar-product-CONS*[*symmetric*]
intro!: *arg-cong2*[*of - - - - scalar-product*] *nth-equalityI*)

lemma *satisfies-lderiv0*[*Presb-simps*]:
$[\![$*wf0* ($\#_V\ \mathfrak{A}$) *a*; $\#_V\ \mathfrak{A}$ = *length x*; *lformula0 a*$]\!]$ ⟹ 𝔄 ⊨ *lderiv0 x a* ⟷ *CONS x* 𝔄 ⊨*0 a*

by (*auto simp*: *Let-def Formula-Operations.satisfies-gen.simps*
split: *if-splits elim*!: *lformula0.cases*)

lemma *satisfies-bounded-lderiv0*[*Presb-simps*]:
$[\![wf0\ (\#_V\ \mathfrak{A})\ a;\ \#_V\ \mathfrak{A} = length\ x;\ lformula0\ a]\!] \Longrightarrow \mathfrak{A} \models_b lderiv0\ x\ a \longleftrightarrow CONS\ x\ \mathfrak{A} \models 0\ a$
by (*auto simp*: *Let-def Formula-Operations.satisfies-gen.simps*
split: *if-splits elim*!: *lformula0.cases*)

lemma *scalar-product-SNOC*: $length\ xs = length\ (bs :: bool\ list) \Longrightarrow$
$scalar\text{-}product\ (map\text{-}index\ (\lambda i\ m.\ m + 2\ \widehat{}\ a * bs\ !\ i)\ xs)\ is =$
$scalar\text{-}product\ xs\ is + 2\ \widehat{}\ a * scalar\text{-}product\ bs\ is$
by (*induct is arbitrary*: *bs xs*) (*auto split*: *list.splits simp*: *algebra-simps*)

lemma *eval-tm-SNOC*[*simp*]:
$[\![length\ is \le \#_V\ \mathfrak{A};\ \#_V\ \mathfrak{A} = length\ x]\!] \Longrightarrow$
$eval\text{-}tm\ (SNOC\ x\ \mathfrak{A})\ is = eval\text{-}tm\ \mathfrak{A}\ is + 2\ \widehat{}\ Length\ \mathfrak{A} * scalar\text{-}product\ x\ is$
by *transfer* (*auto simp*: *scalar-product-SNOC*[*symmetric*]
intro!: *arg-cong2*[*of - - - - scalar-product*] *nth-equalityI*)

lemma *Length-eq0-eval-tm-eq0*[*simp*]: $Length\ \mathfrak{A} = 0 \Longrightarrow eval\text{-}tm\ \mathfrak{A}\ is = 0$
by *transfer* (*auto simp*: *len-eq0-iff scalar-product-eq0*)

lemma *less-pow2*: $x < 2\ \widehat{}\ a \Longrightarrow int\ x < 2\ \widehat{}\ a$
by (*metis of-nat-less-iff of-nat-numeral of-nat-power* [*symmetric*])

lemma *scalar-product-upper-bound*: $\forall x{\in}set\ b.\ len\ x \le a \Longrightarrow$
$scalar\text{-}product\ b\ is \le (2\ \widehat{}\ a - 1) * sum\text{-}list\ [i.\ i \leftarrow is,\ i > 0]$
proof (*induct is arbitrary*: *b*)
case (*Cons i is*)
then have $scalar\text{-}product\ (tl\ b)\ is \le (2\ \widehat{}\ a - 1) * sum\text{-}list\ [i.\ i \leftarrow is,\ i > 0]$
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* - $(2\ \widehat{}\ a - 1) * i$] *less-pow2*
intro: *add-mono elim*: *order-trans*[*OF add-mono*[*OF order-refl*]])
qed *simp*

lemma *scalar-product-lower-bound*: $\forall x{\in}set\ b.\ len\ x \le a \Longrightarrow$
$scalar\text{-}product\ b\ is \ge (2\ \widehat{}\ a - 1) * sum\text{-}list\ [i.\ i \leftarrow is,\ i < 0]$
proof (*induct is arbitrary*: *b*)
case (*Cons i is*)
then have $scalar\text{-}product\ (tl\ b)\ is \ge (2\ \widehat{}\ a - 1) * sum\text{-}list\ [i.\ i \leftarrow is,\ i < 0]$
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* - $(2\ \widehat{}\ a - 1) * i$] *less-pow2*
intro: *add-mono elim*: *order-trans*[*OF add-mono*[*OF order-refl*]] *order-trans*)
qed *simp*

lemma *eval-tm-upper-bound*: *eval-tm* $\mathfrak{A}$ *is* $\leq$ (2 ^ *Length* $\mathfrak{A}$ $-$ 1) $*$ *sum-list* [*i*. *i* $\leftarrow$ *is*, *i* $>$ 0]
  by *transfer* (*auto simp*: *scalar-product-upper-bound*)

lemma *eval-tm-lower-bound*: *eval-tm* $\mathfrak{A}$ *is* $\geq$ (2 ^ *Length* $\mathfrak{A}$ $-$ 1) $*$ *sum-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*$\rrbracket$ $\Longrightarrow$ $\mathfrak{A}$ $\models_b$ *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-product x is* + 2 $*$ *d*
  define *l* **where** *l* = *sum-list* [*i*. *i* $\leftarrow$ *is*, *i* $<$ 0]
  define *h* **where** *h* = *sum-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-tm* $\mathfrak{A}$ *is* + 2 ^ *?l* $*$ *scalar-product x is* = *n* $-$ 2 ^ *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 ^ *?l* $*$ (*h* + *d'*) 2 ^ *?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.RESTRICT* (λ- -. *FBool True*) *FV0*
and *deriv* = λ*d0* (*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* = λ*idx*. *DAs.check-eqv*
(σ *idx*) (λφ. *norm* (*RESTRICT* φ) :: *formula*)
(λ*a* 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*)

($\lambda a\ \varphi.$ *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
(*final idx*) ($\lambda\varphi$ :: *formula. presb-wf idx* φ ∧ *presb-lformula* φ)
(=)
and *bounded-check-eqv* = λ*idx. DAs.check-eqv*
(σ *idx*) ($\lambda\varphi$. *norm* (*RESTRICT* φ) :: *formula*)
($\lambda a\ \varphi.$ *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
nullable ($\lambda\varphi$ :: *formula. presb-wf idx* φ ∧ *presb-lformula* φ)
(σ *idx*) ($\lambda\varphi$. *norm* (*RESTRICT* φ) :: *formula*)
($\lambda a\ \varphi.$ *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
nullable ($\lambda\varphi$ :: *formula. presb-wf idx* φ ∧ *presb-lformula* φ)
(=)
and *automaton* = *DA.automaton*
($\lambda a\ \varphi.$ *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 Formula-Operations.wf.simps*
dest: *satisfies0-cong split*: *presb.splits if-splits*)
done

lemma *check-eqv-code*[*code*]: *check-eqv 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-eqv-def Presb.check-eqv-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-eqv-sound*:
$[\![\#_V\ \mathfrak{A} = idx;\ check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow (Presb.sat\ \mathfrak{A}\ \varphi \longleftrightarrow Presb.sat\ \mathfrak{A}\ \psi)$
unfolding *check-eqv-def* **by** (*rule Presb.check-eqv-soundness*)

lemma *bounded-check-eqv-sound*:
$[\![\#_V\ \mathfrak{A} = idx;\ bounded\text{-}check\text{-}eqv\ idx\ \varphi\ \psi]\!] \Longrightarrow (Presb.sat_b\ \mathfrak{A}\ \varphi \longleftrightarrow Presb.sat_b\ \mathfrak{A}\ \psi)$
unfolding *bounded-check-eqv-def* **by** (*rule Presb.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 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**
λ(*m1*,*m2*) *n* (*bs1*, *bs2*) *bs*. *m1* = *0* ∧ *m2* = *n* ∧ *bs1* = [] ∧ *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*)
(λ*a* 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))
(λ*a* φ. *Presburger-Formula.norm* (*Presburger-Formula.deriv Presburger-Formula.lderiv0 a* φ))
(*Presburger-Formula.final n*)
(λφ. *presb-wf n* φ ∧ *presb-lformula* φ)

(λφ. *Formula.lang Presburger-Formula.nvars*
Presburger-Formula.Extend Presburger-Formula.CONS Presburger-Formula.Length
Presburger-Formula.size-atom (⊨*0*) *n* φ)
(λφ. *presb-wf n* φ ∧ *presb-lformula* φ)
(λφ. *Formula.language Presburger-Formula.assigns*
Presburger-Formula.nvars Presburger-Formula.Extend Presburger-Formula.CONS
Presburger-Formula.Length Presburger-Formula.size-atom (λ- -. *True*)
Presburger-Formula.FV0 (⊨*0*)
n φ)
letter-eq idx n
defines *check-eqv* = λ*idx n*. *DAs.check-eqv*
(σ *idx*) (λφ. *norm* (*RESTRICT* φ) :: (*ws1s*, *order*) *aformula*)
(λ*a* φ. *norm* (*deriv* (*lderiv0* :: - ⇒ - ⇒ *formula*) (*a* :: *atom*) φ))
(*final idx*) (λφ :: *formula*. *ws1s-wf idx* φ ∧ *ws1s-lformula* φ)
(*Presburger-Formula*.σ *n*) (λφ. *Presburger-Formula.norm* (*Presburger-Formula.RESTRICT*
φ))
(λ*a* φ. *Presburger-Formula.norm* (*Presburger-Formula.deriv Presburger-Formula.lderiv0*
a φ))
(*Presburger-Formula.final n*) (λφ. *presb-wf n* φ ∧ *presb-lformula* φ) (*letter-eq*
idx n)
by *unfold-locales auto*

lemma *check-eqv-code*[*code*]: *check-eqv idx n r s* ⟷
((*ws1s-wf idx r* ∧ *ws1s-lformula r*) ∧ (*presb-wf n s* ∧ *presb-lformula s*) ∧
(*case rtrancl-while* (λ(*p*, *q*). *final idx p* = *Presburger-Formula.final n q*)
(λ(*p*, *q*).
map (λ(*a*, *b*). (*norm* (*deriv lderiv0 a p*),
Presburger-Formula.norm (*Presburger-Formula.deriv Presburger-Formula.lderiv0*
b q)))
[(*x*, *y*)←*List.product* (σ *idx*) (*Presburger-Formula*.σ *n*). *letter-eq idx n x y*])
(*norm* (*RESTRICT r*), *Presburger-Formula.norm* (*Presburger-Formula.RESTRICT*
s)) *of*
None ⇒ *False*
| *Some* ([], *x*) ⇒ *True*
| *Some* (*a* # *list*, *x*) ⇒ *False*))
unfolding *check-eqv-def WS1S-Presb.check-eqv-def* **..**

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: *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**

$\quad satisfies\ I1\ I2\ T = True$
$|\ satisfies\ I1\ I2\ F = False$
$|\ satisfies\ I1\ I2\ (Or\ \varphi\ \psi) = (satisfies\ I1\ I2\ \varphi \vee satisfies\ I1\ I2\ \psi)$
$|\ satisfies\ I1\ I2\ (And\ \varphi\ \psi) = (satisfies\ I1\ I2\ \varphi \wedge satisfies\ I1\ I2\ \psi)$
$|\ satisfies\ I1\ I2\ (Not\ \varphi) = (\neg\ satisfies\ I1\ I2\ \varphi)$
$|\ satisfies\ I1\ I2\ (Ex1\ x\ \varphi) = (\exists\, n.\ satisfies\ (I1(x := n))\ I2\ \varphi)$
$|\ satisfies\ I1\ I2\ (Ex2\ X\ \varphi) = (\exists\, N.\ satisfies\ I1\ (I2(X := N))\ \varphi)$
$|\ satisfies\ I1\ I2\ (All1\ x\ \varphi) = (\forall\, n.\ satisfies\ (I1(x := n))\ I2\ \varphi)$
$|\ satisfies\ I1\ I2\ (All2\ X\ \varphi) = (\forall\, N.\ satisfies\ I1\ (I2(X := N))\ \varphi)$

$|\ satisfies\ I1\ I2\ (Lt\ x\ y) = (I1\ x < I1\ y)$
$|\ satisfies\ I1\ I2\ (In\ x\ X) = (I1\ x\ |{\in}|\ I2\ X)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}Const\ x\ n) = (I1\ x = n)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}Presb\ X\ n) = ((\sum x \in fset\ (I2\ X).\ 2\ \widehat{}\ x) = n)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}FO\ x\ y) = (I1\ x = I1\ y)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}FO\text{-}Offset\ x\ y\ n) = (I1\ x = I1\ y + n)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}SO\ X\ Y) = (I2\ X = I2\ Y)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}SO\text{-}Inc\ X\ Y) = (I2\ X = Suc\ |`|\ I2\ Y)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}Max\ x\ X) = (let\ Z = I2\ X\ in\ Z \neq \{||\} \wedge I1\ x = fMax\ Z)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}Min\ x\ X) = (let\ Z = I2\ X\ in\ Z \neq \{||\} \wedge I1\ x = fMin\ Z)$
$|\ satisfies\ I1\ I2\ (Empty\ X) = (I2\ X = \{||\})$
$|\ satisfies\ I1\ I2\ (Singleton\ X) = (\exists\, x.\ I2\ X = \{|x|\})$
$|\ satisfies\ I1\ I2\ (Subset\ X\ Y) = (I2\ X\ |{\subseteq}|\ I2\ Y)$
$|\ satisfies\ I1\ I2\ (Disjoint\ X\ Y) = (I2\ X\ |{\cap}|\ I2\ Y = \{||\})$
$|\ satisfies\ I1\ I2\ (Eq\text{-}Union\ X\ Y\ Z) = (I2\ X = I2\ Y\ |{\cup}|\ I2\ Z)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}Inter\ X\ Y\ Z) = (I2\ X = I2\ Y\ |{\cap}|\ I2\ Z)$
$|\ satisfies\ I1\ I2\ (Eq\text{-}Diff\ X\ Y\ Z) = (I2\ X = I2\ Y\ |{-}|\ I2\ Z)$

primrec *fos* **where**

$\quad fos\ T = []$
$|\ fos\ F = []$
$|\ fos\ (Or\ \varphi\ \psi) = List.union\ (fos\ \varphi)\ (fos\ \psi)$
$|\ fos\ (And\ \varphi\ \psi) = List.union\ (fos\ \varphi)\ (fos\ \psi)$
$|\ fos\ (Not\ \varphi) = fos\ \varphi$
$|\ fos\ (Ex1\ x\ \varphi) = List.remove1\ x\ (fos\ \varphi)$
$|\ fos\ (Ex2\ X\ \varphi) = fos\ \varphi$
$|\ fos\ (All1\ x\ \varphi) = List.remove1\ x\ (fos\ \varphi)$
$|\ fos\ (All2\ X\ \varphi) = fos\ \varphi$
$|\ fos\ (Lt\ x\ y) = remdups\ [x,\ y]$
$|\ fos\ (In\ x\ X) = [x]$
$|\ fos\ (Eq\text{-}Const\ x\ n) = [x]$
$|\ fos\ (Eq\text{-}Presb\ X\ n) = []$
$|\ fos\ (Eq\text{-}FO\ x\ y) = remdups\ [x,\ y]$
$|\ fos\ (Eq\text{-}FO\text{-}Offset\ x\ y\ n) = remdups\ [x,\ y]$
$|\ fos\ (Eq\text{-}SO\ X\ Y) = []$
$|\ fos\ (Eq\text{-}SO\text{-}Inc\ X\ Y) = []$
$|\ fos\ (Eq\text{-}Max\ x\ X) = [x]$
$|\ fos\ (Eq\text{-}Min\ x\ X) = [x]$

| *fos* (*Empty* *X*) = []
| *fos* (*Singleton* *X*) = []
| *fos* (*Subset* *X* *Y*) = []
| *fos* (*Disjoint* *X* *Y*) = []
| *fos* (*Eq-Union* *X* *Y* *Z*) = []
| *fos* (*Eq-Inter* *X* *Y* *Z*) = []
| *fos* (*Eq-Diff* *X* *Y* *Z*) = []

primrec *sos* **where**
  sos *T* = []
| *sos* *F* = []
| *sos* (*Or* φ ψ) = *List.union* (*sos* φ) (*sos* ψ)
| *sos* (*And* φ ψ) = *List.union* (*sos* φ) (*sos* ψ)
| *sos* (*Not* φ) = *sos* φ
| *sos* (*Ex1* *x* φ) = *sos* φ
| *sos* (*Ex2* *X* φ) = *List.remove1* *X* (*sos* φ)
| *sos* (*All1* *x* φ) = *sos* φ
| *sos* (*All2* *X* φ) = *List.remove1* *X* (*sos* φ)
| *sos* (*Lt* *x* *y*) = []
| *sos* (*In* *x* *X*) = [*X*]
| *sos* (*Eq-Const* *x* *n*) = []
| *sos* (*Eq-Presb* *X* *n*) = [*X*]
| *sos* (*Eq-FO* *x* *y*) = []
| *sos* (*Eq-FO-Offset* *x* *y* *n*) = []
| *sos* (*Eq-SO* *X* *Y*) = *remdups* [*X*, *Y*]
| *sos* (*Eq-SO-Inc* *X* *Y*) = *remdups* [*X*, *Y*]
| *sos* (*Eq-Max* *x* *X*) = [*X*]
| *sos* (*Eq-Min* *x* *X*) = [*X*]
| *sos* (*Empty* *X*) = [*X*]
| *sos* (*Singleton* *X*) = [*X*]
| *sos* (*Subset* *X* *Y*) = *remdups* [*X*, *Y*]
| *sos* (*Disjoint* *X* *Y*) = *remdups* [*X*, *Y*]
| *sos* (*Eq-Union* *X* *Y* *Z*) = *remdups* [*X*, *Y*, *Z*]
| *sos* (*Eq-Inter* *X* *Y* *Z*) = *remdups* [*X*, *Y*, *Z*]
| *sos* (*Eq-Diff* *X* *Y* *Z*) = *remdups* [*X*, *Y*, *Z*]

lemma *distinct-fos*[*simp*]: *distinct* (*fos* φ) **by** (*induct* φ) *auto*
lemma *distinct-sos*[*simp*]: *distinct* (*sos* φ) **by** (*induct* φ) *auto*

primrec ε **where**
  ε *bs1* *bs2* *T* = *FBool* *True*
| ε *bs1* *bs2* *F* = *FBool* *False*
| ε *bs1* *bs2* (*Or* φ ψ) = *FOr* (ε *bs1* *bs2* φ) (ε *bs1* *bs2* ψ)
| ε *bs1* *bs2* (*And* φ ψ) = *FAnd* (ε *bs1* *bs2* φ) (ε *bs1* *bs2* ψ)
| ε *bs1* *bs2* (*Not* φ) = *FNot* (ε *bs1* *bs2* φ)
| ε *bs1* *bs2* (*Ex1* *x* φ) = *FEx* *FO* (ε (*x* # *bs1*) *bs2* φ)
| ε *bs1* *bs2* (*Ex2* *X* φ) = *FEx* *SO* (ε *bs1* (*X* # *bs2*) φ)
| ε *bs1* *bs2* (*All1* *x* φ) = *FAll* *FO* (ε (*x* # *bs1*) *bs2* φ)
| ε *bs1* *bs2* (*All2* *X* φ) = *FAll* *SO* (ε *bs1* (*X* # *bs2*) φ)

| ε *bs1 bs2* (*Lt x y*) = *FBase* (*Less None* (*index bs1 x*) (*index bs1 y*))
| ε *bs1 bs2* (*In x X*) = *FBase* (*WS1S-Formula.In False* (*index bs1 x*) (*index bs2 X*))
| ε *bs1 bs2* (*Eq-Const x n*) = *FBase* (*WS1S-Formula.Eq-Const None* (*index bs1 x*) *n*)
| ε *bs1 bs2* (*Eq-Presb X n*) = *FBase* (*WS1S-Formula.Eq-Presb None* (*index bs2 X*) *n*)
| ε *bs1 bs2* (*Eq-FO x y*) = *FBase* (*WS1S-Formula.Eq-FO False* (*index bs1 x*) (*index bs1 y*))
| ε *bs1 bs2* (*Eq-FO-Offset x y n*) = *FBase* (*WS1S-Formula.Plus-FO None* (*index bs1 x*) (*index bs1 y*) *n*)
| ε *bs1 bs2* (*Eq-SO X Y*) = *FBase* (*WS1S-Formula.Eq-SO* (*index bs2 X*) (*index bs2 Y*))
| ε *bs1 bs2* (*Eq-SO-Inc X Y*) = *FBase* (*WS1S-Formula.Suc-SO False False* (*index bs2 X*) (*index bs2 Y*))
| ε *bs1 bs2* (*Eq-Max x X*) = *FBase* (*WS1S-Formula.Eq-Max False* (*index bs1 x*) (*index bs2 X*))
| ε *bs1 bs2* (*Eq-Min x X*) = *FBase* (*WS1S-Formula.Eq-Min False* (*index bs1 x*) (*index bs2 X*))
| ε *bs1 bs2* (*Empty X*) = *FBase* (*WS1S-Formula.Empty* (*index bs2 X*))
| ε *bs1 bs2* (*Singleton X*) = *FBase* (*WS1S-Formula.Singleton* (*index bs2 X*))
| ε *bs1 bs2* (*Subset X Y*) = *FBase* (*WS1S-Formula.Subset* (*index bs2 X*) (*index bs2 Y*))
| ε *bs1 bs2* (*Disjoint X Y*) = *FBase* (*WS1S-Formula.Disjoint* (*index bs2 X*) (*index bs2 Y*))
| ε *bs1 bs2* (*Eq-Union X Y Z*) = *FBase* (*WS1S-Formula.Eq-Union* (*index bs2 X*) (*index bs2 Y*) (*index bs2 Z*))
| ε *bs1 bs2* (*Eq-Inter X Y Z*) = *FBase* (*WS1S-Formula.Eq-Inter* (*index bs2 X*) (*index bs2 Y*) (*index bs2 Z*))
| ε *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* ⇒ *nat*) ⇒ (*so* ⇒ *nat fset*) ⇒ *fo list* ⇒ *so list* ⇒ *interp* **is**
λ*I1 I2 fs ss. let I1s* = *map* (λ*x*. {|*I1 x*|}) *fs*; *I2s* = *map I2 ss in* (*MSB* (*I1s* @ *I2s*), *I1s*, *I2s*)
by (*auto simp*: *Let-def*)

definition *dec-I1* :: *interp* ⇒ *fo list* ⇒ *fo* ⇒ *nat* **where** *dec-I1* $\mathfrak{A}$ *fs x* = *fMin* (*index fs* $x^{\mathfrak{A}}$*FO*)
definition *dec-I2* :: *interp* ⇒ *so list* ⇒ *so* ⇒ *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*]:
$m^{mk\text{-}I\ I1\ I2\ bs1\ bs2}$*FO* = (*if m* < *length bs1 then* {|*I1* (*bs1* ! *m*)|} *else* {||})
by *transfer* (*auto simp*: *Let-def*)

lemma *assigns-mk-I-SO*[*simp*]:
$m^{mk\text{-}I\ I1\ I2\ bs1\ bs2}SO = (if\ m < length\ bs2\ then\ I2\ (bs2\ !\ m)\ else\ \{||\})$
by *transfer* (*auto simp*: *Let-def*)

lemma *satisfies-cong*:
$[\![\forall x \in set\ (fos\ \varphi).\ I1\ x = J1\ x;\ \forall X \in set\ (sos\ \varphi).\ I2\ X = J2\ X]\!] \Longrightarrow$
satisfies I1 I2 $\varphi \longleftrightarrow$ *satisfies J1 J2* φ
proof (*induct* φ *arbitrary*: *I1 I2 J1 J2*)
case (*Or* φ ψ) **from** *Or.hyps*[*of I1 J1 I2 J2*] *Or.prems* **show** *?case* **by** *auto*
next
case (*And* φ ψ) **from** *And.hyps*[*of I1 J1 I2 J2*] *And.prems* **show** *?case* **by** *auto*
next
case (*Ex1 x* φ) **with** *Ex1.hyps*[*of I1*$(x := y)$ *J1*$(x := y)$ *I2 J2* **for** *y*, *cong*] **show** *?case* **by** *auto*
next
case (*Ex2 X* φ) **with** *Ex2.hyps*[*of I1 J1 I2*$(X := Y)$ *J2*$(X := Y)$ **for** *Y*, *cong*] **show** *?case* **by** *auto*
next
case (*All1 x* φ) **with** *All1.hyps*[*of I1*$(x := y)$ *J1*$(x := y)$ *I2 J2* **for** *y*, *cong*] **show** *?case* **by** *auto*
next
case (*All2 X* φ) **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*:
$[\![set\ (fos\ \varphi) \subseteq set\ bs1;\ set\ (sos\ \varphi) \subseteq set\ bs2;\ \mathfrak{A} = mk\text{-}I\ I1\ I2\ bs1\ bs2]\!] \Longrightarrow$
satisfies (*dec-I1* $\mathfrak{A}$ *bs1*) (*dec-I2* $\mathfrak{A}$ *bs2*) $\varphi \longleftrightarrow$ *satisfies I1 I2* φ
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*]: $[\![v \in FV\ (\varepsilon\ bs1\ bs2\ \varphi)\ FO;$
$set\ (fos\ \varphi) \subseteq set\ bs1;\ set\ (sos\ \varphi) \subseteq set\ bs2]\!] \Longrightarrow$
$(\exists y \in set\ bs1.\ v = index\ bs1\ y)$
proof (*induct* φ *arbitrary*: *bs1 bs2 v*)
case (*Ex1 x* φ) **from** *Ex1.hyps*[*of Suc v x # bs1 bs2*] *Ex1.prems* **show** *?case*
by (*auto simp*: *Diff-subset-conv split*: *if-splits*)
next
case (*Ex2 X* φ) **from** *Ex2.hyps*[*of v bs1 X # bs2*] *Ex2.prems* **show** *?case*
by (*auto simp*: *Diff-subset-conv split*: *if-splits*)
next
case (*All1 x* φ) **from** *All1.hyps*[*of Suc v x # bs1 bs2*] *All1.prems* **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.prems* **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-*ε: $[\![$*set (fos* φ*)* $\subseteq$ *set bs1*; *set (sos* φ*)* $\subseteq$ *set bs2*; *ok-I* $\mathfrak{A}$ *bs1*$]\!] \Longrightarrow$
  WS1S.sat $\mathfrak{A}$ $(\varepsilon$ *bs1 bs2* $\varphi) \longleftrightarrow$
  satisfies (dec-I1 $\mathfrak{A}$ *bs1) (dec-I2* $\mathfrak{A}$ *bs2)* φ
proof *(induct* φ *arbitrary:* $\mathfrak{A}$ *bs1 bs2)*
  case *(Or* φ ψ*)* **from** *Or.hyps[of bs1 bs2* $\mathfrak{A}$*] Or.prems* **show** *?case*
    unfolding *WS1S.sat-def*
    by *(auto simp: restrict-def ok-I-def split: order.splits)*
next
  case *(And* φ ψ*)* **from** *And.hyps[of bs1 bs2* $\mathfrak{A}$*] And.prems* **show** *?case*
    unfolding *WS1S.sat-def*
    by *(auto simp: restrict-def ok-I-def split: order.splits)*
next
  case *(Not* φ*)* **from** *Not.hyps[of bs1 bs2* $\mathfrak{A}$*] Not.prems* **show** *?case*
    unfolding *WS1S.sat-def*
    by *(auto simp: restrict-def ok-I-def split: order.splits)*
next
  case *(Ex1 x* φ*)*
  { fix *P :: nat fset* **assume** $P \neq \{||\}$
   with *Ex1.prems* **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)* φ
     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* φ*))*
    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)* φ

by (*auto simp*: *IH*)
then show *satisfies* (*dec-I1* $\mathfrak{A}$ *bs1*) (*dec-I2* $\mathfrak{A}$ *bs2*) (*Ex1 x* φ) **by** *auto*
next
assume *satisfies* (*dec-I1* $\mathfrak{A}$ *bs1*) (*dec-I2* $\mathfrak{A}$ *bs2*) (*Ex1 x* φ)
then obtain *n* **where** *satisfies* (*dec-I1* (*Extend FO 0* $\mathfrak{A}$ $\{|n|\}$) (*x* # *bs1*))
(*dec-I2* (*Extend FO 0* $\mathfrak{A}$ $\{|n|\}$) *bs2*) φ **by** *auto*
then have *WS1S.sat* (*Extend FO 0* $\mathfrak{A}$ $\{|n|\}$) (ε (*x* # *bs1*) *bs2* φ)
by (*auto simp*: *IH*)
then show *WS1S.sat* $\mathfrak{A}$ (ε *bs1 bs2* (*Ex1 x* φ)) **unfolding** *WS1S.sat-def*
by (*auto simp*: *restrict-def split*: *order.splits*)
qed
next
case (*Ex2 X* φ)
{ fix *P* :: *nat fset*
from *Ex2.prems* **have** *WS1S.sat* (*Extend SO 0* $\mathfrak{A}$ *P*) (ε *bs1* (*X* # *bs2*) φ) $\longleftrightarrow$
satisfies (*dec-I1* (*Extend SO 0* $\mathfrak{A}$ *P*) *bs1*) (*dec-I2* (*Extend SO 0* $\mathfrak{A}$ *P*) (*X* #
bs2)) φ
by (*intro Ex2.hyps*) (*auto simp*: *ok-I-def*)
} **note** *IH* = *this*
show *?case*
proof −
have *WS1S.sat* $\mathfrak{A}$ (ε *bs1 bs2* (*Ex2 X* φ)) $\longleftrightarrow$
($\exists$ *P*. *WS1S.sat* (*Extend SO 0* $\mathfrak{A}$ *P*) (ε *bs1* (*X* # *bs2*) φ))
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*)) φ) **by** (*auto simp*: *IH*)
also have ... $\longleftrightarrow$ *satisfies* (*dec-I1* $\mathfrak{A}$ *bs1*) (*dec-I2* $\mathfrak{A}$ *bs2*) (*Ex2 X* φ) **by** *simp*
finally show *?thesis* **.**
qed
next
case (*All1 x* φ)
{ fix *P* :: *nat fset* **assume** $P \neq \{||\}$
with *All1.prems* **have** *WS1S.sat* (*Extend FO 0* $\mathfrak{A}$ *P*) (ε (*x* # *bs1*) *bs2* φ) $\longleftrightarrow$
satisfies (*dec-I1* (*Extend FO 0* $\mathfrak{A}$ *P*) (*x* # *bs1*)) (*dec-I2* (*Extend FO 0* $\mathfrak{A}$ *P*)
bs2) φ
by (*intro All1.hyps*) (*auto simp*: *ok-I-def*)
} **note** *IH* = *this*
show *?case*
proof
assume *L*: *WS1S.sat* $\mathfrak{A}$ (ε *bs1 bs2* (*All1 x* φ))
{ fix *n* :: *nat*
from *L* **have** *WS1S.sat* (*Extend FO 0* $\mathfrak{A}$ $\{|n|\}$) (ε (*x* # *bs1*) *bs2* φ)
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*) φ **by** (*auto simp*: *IH*)
}
then show *satisfies* (*dec-I1* $\mathfrak{A}$ *bs1*) (*dec-I2* $\mathfrak{A}$ *bs2*) (*All1 x* φ) **by** *simp*
next
assume *R*: *satisfies* (*dec-I1* $\mathfrak{A}$ *bs1*) (*dec-I2* $\mathfrak{A}$ *bs2*) (*All1 x* φ)

{ **fix** P :: *nat fset* **assume** $P \neq \{||\}$
with R **have** *satisfies* (*dec-I1* (*Extend FO 0* $\mathfrak{A}$ P) (x # *bs1*))
(*dec-I2* (*Extend FO 0* $\mathfrak{A}$ P) *bs2*) φ **by** *auto*
with ‹$P \neq \{||\}$› **have** *WS1S.sat* (*Extend FO 0* $\mathfrak{A}$ P) (ε (x # *bs1*) *bs2* φ)
by (*auto simp*: *IH*)
}
with *All1.prems in-FV-εD*[*of* - x # *bs1 bs2* φ]
show *WS1S.sat* $\mathfrak{A}$ (ε *bs1 bs2* (*All1* x φ)) **unfolding** *WS1S.sat-def*
by (*auto 0 3 simp*: *restrict-def ok-I-def split*: *if-splits order.splits*)
qed
next
case (*All2* X φ)
{ **fix** P :: *nat fset*
from *All2.prems* **have** *WS1S.sat* (*Extend SO 0* $\mathfrak{A}$ P) (ε *bs1* (X # *bs2*) φ)
$\longleftrightarrow$
satisfies (*dec-I1* (*Extend SO 0* $\mathfrak{A}$ P) *bs1*) (*dec-I2* (*Extend SO 0* $\mathfrak{A}$ P) (X #
bs2)) φ
by (*intro All2.hyps*) (*auto simp*: *ok-I-def*)
} **note** *IH* = *this*
show *?case*
proof −
have *WS1S.sat* $\mathfrak{A}$ (ε *bs1 bs2* (*All2* X φ)) $\longleftrightarrow$
($\forall P$. *WS1S.sat* (*Extend SO 0* $\mathfrak{A}$ P) (ε *bs1* (X # *bs2*) φ))
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*)) φ) **by** (*auto simp*: *IH*)
also have ... $\longleftrightarrow$ *satisfies* (*dec-I1* $\mathfrak{A}$ *bs1*) (*dec-I2* $\mathfrak{A}$ *bs2*) (*All2* X φ) **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* φ ψ =
(*let fs* = *List.union* (*fos* φ) (*fos* ψ); *ss* = *List.union* (*sos* φ) (*sos* ψ)
in check-eqv (*Abs-idx* (*length fs*, *length ss*)) (ε *fs ss* φ) (ε *fs ss* ψ))

lemma *eqv-sound*: *eqv* φ ψ $\Longrightarrow$ *satisfies I1 I2* φ $\longleftrightarrow$ *satisfies I1 I2* ψ
unfolding *eqv-def Let-def*
by (*drule check-eqv-sound*[*rotated*, *of* - - -
mk-I I1 I2 (*List.union* (*fos* φ) (*fos* ψ)) (*List.union* (*sos* φ) (*sos* ψ))])
(*auto simp*: *sat-ε dec-I-mk-I-satisfies-cong*)

definition *Thm* φ = *eqv* φ T

lemma *Thm* Φ $\Longrightarrow$ *satisfies I1 I2* Φ
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* ⇒ *fo* ⇒ *bool* **is** (=) **.**
  instance by (*standard*, *transfer*) *simp*
end

instantiation *so* :: *equal*
begin
  lift-definition *equal-so* :: *so* ⇒ *so* ⇒ *bool* **is** (=) **.**
  instance by (*standard*, *transfer*) *simp*
end