Differential-Dynamic-Logic

Rose Bohrer

March 17, 2025

Abstract

We formalize differential dynamic logic, a logic for proving properties of hybrid systems. The proof calculus in this formalization is based on the uniform substitution principle. We show it is sound with respect to our denotational semantics, which provides increased confidence in the correctness of the KeYmaera X theorem prover based on this calculus. As an application, we include a proof term checker embedded in Isabelle/HOL with several example proofs.

Published in [1]

We present a formalization of a uniform substitution calculus for differential dynamic logic (dL). In this calculus, the soundness of dL proofs is reduced to the soundness of a finite number of axioms, standard propositional rules and a central *uniform substitution* rule for combining axioms. We present a formal definition for the denotational semantics of dL and prove the uniform substitution calculus sound by showing that all inference rules are sound with respect to the denotational semantics, and all axioms valid (true in every state and interpretation).

This work is published in [1] along with a Coq formalization. It is based on prior non-mechanized proofs [3, 2].

Contents

theory *Ids*
imports *Complex-Main*
begin

1 Identifier locale

The differential dynamic logic formalization is parameterized by the type of identifiers. The identifier type(s) must be finite and have at least 3-4 distinct elements. Distinctness is required for soundness of some axioms.

locale *ids* =
  fixes *vid1* :: ('*sz*::{*finite*,*linorder*})
  fixes *vid2* :: '*sz*
  fixes *vid3* :: '*sz*
  fixes *fid1* :: ('*sf*::*finite*)
  fixes *fid2* :: '*sf*
  fixes *fid3* :: '*sf*
  fixes *pid1* :: ('*sc*::*finite*)
  fixes *pid2* :: '*sc*
  fixes *pid3* :: '*sc*
  fixes *pid4* :: '*sc*
  assumes *vne12*:*vid1* ≠ *vid2*
  assumes *vne23*:*vid2* ≠ *vid3*
  assumes *vne13*:*vid1* ≠ *vid3*
  assumes *fne12*:*fid1* ≠ *fid2*
  assumes *fne23*:*fid2* ≠ *fid3*
  assumes *fne13*:*fid1* ≠ *fid3*
  assumes *pne12*:*pid1* ≠ *pid2*
  assumes *pne23*:*pid2* ≠ *pid3*
  assumes *pne13*:*pid1* ≠ *pid3*
  assumes *pne14*:*pid1* ≠ *pid4*
  assumes *pne24*:*pid2* ≠ *pid4*
  assumes *pne34*:*pid3* ≠ *pid4*
context *ids* **begin**
lemma *id-simps*:
  (*vid1* = *vid2*) = *False* (*vid2* = *vid3*) = *False* (*vid1* = *vid3*) = *False*
  (*fid1* = *fid2*) = *False* (*fid2* = *fid3*) = *False* (*fid1* = *fid3*) = *False*
  (*pid1* = *pid2*) = *False* (*pid2* = *pid3*) = *False* (*pid1* = *pid3*) = *False*
  (*pid1* = *pid4*) = *False* (*pid2* = *pid4*) = *False* (*pid3* = *pid4*) = *False*
  (*vid2* = *vid1*) = *False* (*vid3* = *vid2*) = *False* (*vid3* = *vid1*) = *False*

(*fid2* = *fid1*) = *False* (*fid3* = *fid2*) = *False* (*fid3* = *fid1*) = *False*
(*pid2* = *pid1*) = *False* (*pid3* = *pid2*) = *False* (*pid3* = *pid1*) = *False*
(*pid4* = *pid1*) = *False* (*pid4* = *pid2*) = *False* (*pid4* = *pid3*) = *False*
using *vne12 vne23 vne13 fne12 fne23 fne13 pne12 pne23 pne13 pne14 pne24 pne34* **by** *auto*
end
end
theory *Lib*
imports
Ordinary-Differential-Equations.ODE-Analysis
begin

2 Generic Mathematical Lemmas

General lemmas that don't have anything to do with dL specifically and could be fit for general-purpose libraries, mostly dealing with derivatives, ODEs and vectors.

lemma *vec-extensionality*:$(\bigwedge i.\ v\$i = w\$i) \Longrightarrow (v = w)$
by (*simp add*: *vec-eq-iff*)

lemma *norm-axis*: *norm* (*axis i x*) = *norm x*
unfolding *axis-def norm-vec-def*
unfolding *L2-set-def*
by(*clarsimp simp add*: *if-distrib*[**where** *f*=*norm*] *if-distrib*[**where** *f*=$\lambda x.\ x^2$] *sum.If-cases*)

lemma *bounded-linear-axis*: *bounded-linear* (*axis i*)
proof
show *axis i* (*x* + *y*) = *axis i x* + *axis i y axis i* (*r* $*_R$ *x*) = *r* $*_R$ *axis i x* **for** *x y* :: *'a* **and** *r*
by (*auto simp*: *vec-eq-iff axis-def*)
show $\exists K.\ \forall x::'a.$ *norm* (*axis i x*) $\leq$ *norm x* $*$ *K*
by (*auto simp add*: *norm-axis intro*!: *exI*[*of - 1*])
qed

lemma *bounded-linear-vec*:
fixes *f*::('*a*::*finite*) $\Rightarrow$ '*b*::*real-normed-vector* $\Rightarrow$ '*c*::*real-normed-vector*
assumes *bounds*:$\bigwedge i.$ *bounded-linear* (*f i*)
shows *bounded-linear* ($\lambda x.\ \chi\ i.\ f\ i\ x$)
proof *unfold-locales*
fix *r x y*
interpret *bounded-linear f i* **for** *i* **by** *fact*
show ($\chi\ i.\ f\ i$ (*x* + *y*)) = ($\chi\ i.\ f\ i\ x$) + ($\chi\ i.\ f\ i\ y$)
by (*vector add*)
show ($\chi\ i.\ f\ i$ (*r* $*_R$ *x*)) = *r* $*_R$ ($\chi\ i.\ f\ i\ x$)
by (*vector scaleR*)
obtain *K* **where** *norm* (*f i x*) $\leq$ *norm x* $*$ *K i* **for** *x i*
using *bounded* **by** *metis*

then have *norm* (χ *i. f i x*) $\leq$ *norm x* $*$ ($\sum$ *i*$\in$*UNIV. K i*) (**is** *?lhs* $\leq$ *?rhs*) **for** *x*
unfolding *sum-distrib-left*
unfolding *norm-vec-def*
by (*auto intro*!: *L2-set-le-sum-abs*[*THEN order-trans*] *sum-mono simp*: *abs-mult*)
then show $\exists$ *K*. $\forall$ *x. norm* (χ *i. f i x*) $\leq$ *norm x* $*$ *K*
by *blast*
qed

lift-definition *blinfun-vec*::('*a*::*finite* $\Rightarrow$ '*b*::*real-normed-vector* $\Rightarrow_L$ *real*) $\Rightarrow$ '*b* $\Rightarrow_L$ (*real* ^ '*a*) **is** (λ(*f*::('*a* $\Rightarrow$ '*b* $\Rightarrow$ *real*)) (*x*::'*b*). χ (*i*::'*a*). *f i x*)
by(*rule bounded-linear-vec*, *simp*)

lemmas *blinfun-vec-simps*[*simp*] = *blinfun-vec.rep-eq*

lemma *continuous-blinfun-vec*:($\bigwedge$*i. continuous-on UNIV* (*blinfun-apply* (*g i*))) $\Longrightarrow$ *continuous-on UNIV* (*blinfun-vec g*)
by (*simp add*: *continuous-on-vec-lambda*)

lemma *blinfun-elim*:$\bigwedge$*g*. (*blinfun-apply* (*blinfun-vec g*)) = (λ*x*. χ *i. g i x*)
using *blinfun-vec.rep-eq* **by** *auto*

lemma *sup-plus*:
fixes *f g*::('*b*::*metric-space*) $\Rightarrow$ *real*
assumes *nonempty*:*R* $\neq$ {}
assumes *bddf*:*bdd-above* (*f* ‘ *R*)
assumes *bddg*:*bdd-above* (*g* ‘ *R*)
shows (*SUP x*$\in$*R. f x* + *g x*) $\leq$ (*SUP x*$\in$*R. f x*) + (*SUP x*$\in$*R. g x*)
proof −
have *bddfg*:*bdd-above*((λ*x. f x* + *g x*) ‘ *R*)
using *bddf bddg* **apply** (*auto simp add*: *bdd-above-def*)
using *add-mono-thms-linordered-semiring*(*1*) **by** *blast*
have *eq*:(*SUP x*$\in$*R. f x* + *g x*) $\leq$ (*SUP x*$\in$*R. f x*) + (*SUP x*$\in$*R. g x*)
$\longleftrightarrow$ ($\forall$ *x*$\in$*R.* (*f x* + *g x*) $\leq$ (*SUP x*$\in$*R. f x*) + (*SUP x*$\in$*R. g x*))
apply(*rule cSUP-le-iff*)
subgoal by (*rule nonempty*)
subgoal by (*rule bddfg*)
done
have *fs*:$\bigwedge$*x. x* $\in$ *R* $\Longrightarrow$ *f x* $\leq$ (*SUP x*$\in$*R. f x*)
using *bddf*
by (*simp add*: *cSUP-upper*)
have *gs*:$\bigwedge$*x. x* $\in$ *R* $\Longrightarrow$ *g x* $\leq$ (*SUP x*$\in$*R. g x*)
using *bddg*
by (*simp add*: *cSUP-upper*)
have ($\forall$ *x*$\in$*R.* (*f x* + *g x*) $\leq$ (*SUP x*$\in$*R. f x*) + (*SUP x*$\in$*R. g x*))
apply *auto*
subgoal for *x* **using** *fs*[*of x*] *gs*[*of x*] **by** *auto*
done
then show *?thesis* **by** (*auto simp add*: *eq*)

qed

lemma *continuous-blinfun-vec'*:
fixes f::$'a$::{*finite*,*linorder*} $\Rightarrow$ $'b$::{*metric-space*, *real-normed-vector*,*abs*} $\Rightarrow$ $'b$ $\Rightarrow_L$ *real*
fixes S::$'b$ *set*
assumes *conts*:$\bigwedge i$. *continuous-on UNIV* (f i)
shows *continuous-on UNIV* (λx. *blinfun-vec* (λ i. f i x))
proof (*auto simp add*: *LIM-def continuous-on-def*)
fix *x1* **and** ε::*real*
assume ε:$0 < \varepsilon$
let *?n* = *card* (*UNIV*::$'a$ *set*)
have *conts'*: $\bigwedge i$ *x1* ε. $0 < \varepsilon \Longrightarrow \exists \delta{>}0.\ \forall$ *x2*. *x2* $\neq$ *x1* $\wedge$ *dist x2 x1* $< \delta \longrightarrow$ *dist* (*f i x2*) (*f i x1*) $< \varepsilon$
using *conts* **by**(*auto simp add*: *LIM-def continuous-on-def*)
have *conts''*:$\bigwedge i$. $\exists \delta{>}0.\ \forall$ *x2*. *x2* $\neq$ *x1* $\wedge$ *dist x2 x1* $< \delta \longrightarrow$ *dist* (*f i x2*) (*f i x1*) $< (\varepsilon / ?n)$
subgoal for *i* **using** *conts'*[*of* ε / *?n x1 i*] ε **by** *auto* **done**
let *?f* = (λx. *blinfun-vec* (λ i. f i x))
let *?P*δ = (λ i δ. ($\delta{>}0$ $\wedge$ ($\forall$ *x2*. *x2* $\neq$ *x1* $\wedge$ *dist x2 x1* $< \delta \longrightarrow$ *dist* (*f i x2*) (*f i x1*) $< (\varepsilon / ?n)$)))
let *?*δi = (λi. *SOME* δ. *?P*δ i δ)
have *Ps*:$\bigwedge i$. $\exists \delta$. *?P*δ i δ **using** *conts''* **by** *auto*
have *P*δi:$\bigwedge i$. *?P*δ i (*?*δi i)
subgoal for *i* **using** *someI*[*of ?P*δ i] *Ps*[*of i*] **by** *auto* **done**
have *finU*:*finite* (*UNIV*::$'a$ *set*) **by** *auto*
let *?*δ = *linorder-class.Min* (*?*δi ‘ *UNIV*)
have $\delta 0$*s*:$\bigwedge i$. *?*δi $i > 0$ **using** *P*δi **by** *blast*
then have $\delta 0$*s'*:$\bigwedge i$. $0 <$ *?*δi i **by** *auto*
have *bounds*:*bdd-below* (*?*δi ‘ *UNIV*)
unfolding *bdd-below-def*
using $\delta 0$*s less-eq-real-def* **by** *blast*
have δ*s*:$\bigwedge i$. *?*$\delta \leq$ *?*δi i
using *bounds cINF-lower*[*of ?*δi] **by** *auto*
have *finite*:*finite* ((*?*δi ‘ *UNIV*)) **by** *auto*
have *nonempty*:((*?*δi ‘ *UNIV*)) $\neq$ {} **by** *auto*
have δ:*?*$\delta > 0$ **using** *Min-gr-iff*[*OF finite nonempty*] $\delta 0$*s'*
by *blast*
have *conts'''*:$\bigwedge i$ *x2*. *x2* $\neq$ *x1* $\Longrightarrow$ *dist x2 x1* $<$ *?*δi $i \Longrightarrow$ *dist* (*f i x2*) (*f i x1*) $< (\varepsilon / ?n)$
subgoal for *i x2*
using *conts''*[*of i*] **apply** *auto*
subgoal for δ
apply(*erule allE*[**where** *x*=*x2*])
using *P*δi δ*s*[*of i*] **apply** (*auto simp add*: δ*s*[*of i*])
done
done
done
have $\bigwedge$*x2*. *x2* $\neq$ *x1* $\wedge$ *dist x2 x1* $<$ *?*$\delta \Longrightarrow$ *dist* (*blinfun-vec* (λi. f i *x2*))

```
(blinfun-vec (λi. f i x1)) < ε
  proof (auto)
    fix x2
    assume ne:x2 ≠ x1
    assume dist:∀ i. dist x2 x1 < ?δi i
    have dists:⋀i. dist x2 x1 < ?δi i
      subgoal for i using dist δs[of i] by auto done
    have euclid:⋀y. norm(?f x1 y − ?f x2 y) = (L2-set (λi. norm(f i x1 y − f i x2
y)) UNIV)
      by (simp add: norm-vec-def)
    have finite:finite (UNIV::'a set) by auto
    have nonempty: (UNIV::'a set) ≠ {} by auto
    have nonemptyB: (UNIV::'b set) ≠ {} by auto
    have nonemptyR: (UNIV::real set) ≠ {} by auto
     have SUP-leq:⋀f::('b ⇒ real). ⋀ g::('b ⇒ real). ⋀ S::'b set. S ≠ {} ⟹
bdd-above (g ‘ S) ⟹ (⋀x. x ∈ (S::'b set) ⟹ ((f x)::real) ≤ ((g x)::real)) ⟹
(SUP x∈S. f x) ≤ (SUP x∈S. g x)
      by(rule cSup-mono, auto)
    have SUP-sum-comm':⋀R S f . finite (S::'a set) ⟹ (R::'d::metric-space set)
≠ {} ⟹
      (⋀i x. ((f i x)::real) ≥ 0) ⟹
      (⋀i. bdd-above (f i ‘ R)) ⟹
      (SUP x∈R . (∑ i ∈ S. f i x)) ≤ (∑ i ∈ S. (SUP x∈R. f i x))
    proof −
      fix R::'d set and S ::('a)set and f ::'a ⇒ 'd ⇒ real
      assume non:R ≠ {}
      assume fin:finite S
      assume every:(⋀i x. 0 ≤ f i x)
      assume bddF:⋀i. bdd-above (f i ‘ R)
      then have bddF':⋀i. ∃ M. ∀ x ∈R. f i x ≤ M
        unfolding bdd-above-def by auto
      let ?boundP = (λi M. ∀ x ∈R. f i x ≤ M)
      let ?bound = (λi::'a. SOME M. ∀ x ∈R. f i x ≤ M)
      have ⋀i. ∃ M. ?boundP i M using bddF' by auto
      then have each-bound:⋀i. ?boundP i (?bound i)
        subgoal for i using someI[of ?boundP i] by blast done
      let ?bigBound = (λF. ∑ i∈F. (?bound i))
      have bddG:⋀i::'a. ⋀F. bdd-above ((λx. ∑ i∈F. f i x) ‘ R)
        subgoal for i F
          using bddF[of i] unfolding bdd-above-def apply auto
          apply(rule exI[where x=?bigBound F])
          subgoal for M
            apply auto
            subgoal for x
              using each-bound by (simp add: sum-mono)
            done
          done
        done
      show ?thesis R S f using fin assms
```

proof (*induct*)
case *empty*
have ((*SUP* $x{\in}R.\ \sum i{\in}\{\}.\ f\ i\ x$)::*real*) $\leq$ ($\sum i{\in}\{\}.$ *SUP* $a{\in}R.\ f\ i\ a$) **by** (*simp add*: *non*)
then show *?case* **by** *auto*
next
case (*insert x F*)
have ((*SUP* $xa{\in}R.\ \sum i{\in}$*insert x F*$.\ f\ i\ xa$)::*real*) $\leq$ (*SUP* $xa{\in}R.\ f\ x\ xa$ + ($\sum i{\in}F.\ f\ i\ xa$))
using *insert.hyps*(*2*) **by** *auto*
moreover have ... $\leq$ (*SUP* $xa{\in}$ *R*. $f\ x\ xa$) + (*SUP* $xa{\in}R.$ ($\sum i{\in}F.\ f\ i\ xa$))
by(*rule sup-plus*, *rule non*, *rule bddF*, *rule bddG*)
moreover have ... $\leq$ (*SUP* $xa{\in}$ *R*. $f\ x\ xa$) + ($\sum i{\in}F.$ *SUP* $a{\in}R.\ f\ i\ a$)
using *add-le-cancel-left conts insert.hyps*(*3*) **by** *blast*
moreover have ... $\leq$ ($\sum i{\in}$(*insert x F*). *SUP* $a{\in}R.\ f\ i\ a$)
by (*simp add*: *insert.hyps*(*2*))
ultimately have ((*SUP* $xa{\in}R.\ \sum i{\in}$*insert x F*$.\ f\ i\ xa$)::*real*) $\leq$ ($\sum i{\in}$(*insert x F*). *SUP* $a{\in}R.\ f\ i\ a$)
by *linarith*
then show *?case* **by** *auto*
qed
qed
have *SUP-sum-comm*:$\bigwedge R\ S\ y1\ y2$. *finite* (*S*::*'a set*) $\Longrightarrow$ (*R*::*'b set*) $\neq$ {} $\Longrightarrow$ (*SUP* $x{\in}R$. ($\sum i \in S.$ *norm*($f\ i\ y1\ x - f\ i\ y2\ x$)/*norm*($x$))) $\leq$ ($\sum i \in S.$ (*SUP* $x{\in}R.$ *norm*($f\ i\ y1\ x - f\ i\ y2\ x$)/*norm*($x$)))
apply(*rule SUP-sum-comm'*)
apply(*auto*)[*3*]
proof (*unfold bdd-above-def*)
fix *R S y1 y2 i*
{ fix *rr* :: *real* $\Rightarrow$ *real*
obtain *bb* :: *real* $\Rightarrow$ (*'b* $\Rightarrow$ *real*) $\Rightarrow$ *'b set* $\Rightarrow$ *'b* **where**
ff1: $\bigwedge r\ f\ B.\ r \notin f$ ‘ $B \vee f\ (bb\ r\ f\ B) = r$
unfolding *image-iff* **by** *moura*
{ assume $\exists r.\ \neg$ *rr r* $\leq$ *norm* (*f i y1* $-$ *f i y2*)
then have $\exists r.$ *norm* (*blinfun-apply* (*f i y1*) (*bb* (*rr r*) ($\lambda b.$ *norm* (*blinfun-apply* (*f i y1*) *b* $-$ *blinfun-apply* (*f i y2*) *b*) / *norm b*) *R*) $-$ *blinfun-apply* (*f i y2*) (*bb* (*rr r*) ($\lambda b.$ *norm* (*blinfun-apply* (*f i y1*) *b* $-$ *blinfun-apply* (*f i y2*) *b*) / *norm b*) *R*)) / *norm* (*bb* (*rr r*) ($\lambda b.$ *norm* (*blinfun-apply* (*f i y1*) *b* $-$ *blinfun-apply* (*f i y2*) *b*) / *norm b*) *R*) $\neq$ *rr r*
by (*metis* (*no-types*) *le-norm-blinfun minus-blinfun.rep-eq*)
then have $\exists r.$ *rr r* $\leq r \vee$ *rr r* $\notin$ ($\lambda b.$ *norm* (*blinfun-apply* (*f i y1*) *b* $-$ *blinfun-apply* (*f i y2*) *b*) / *norm b*) ‘ *R*
using *ff1* **by** *meson* **}**
then have $\exists r.$ *rr r* $\leq r \vee$ *rr r* $\notin$ ($\lambda b.$ *norm* (*blinfun-apply* (*f i y1*) *b* $-$ *blinfun-apply* (*f i y2*) *b*) / *norm b*) ‘ *R*
by *blast* **}**
then show $\exists r.\ \forall ra{\in}$($\lambda b.$ *norm* (*blinfun-apply* (*f i y1*) *b* $-$ *blinfun-apply* (*f i y2*) *b*) / *norm b*) ‘ *R*. *ra* $\leq r$
by *meson*

qed
have *SUM-leq*:$\bigwedge S$::('*a*) *set*. $\bigwedge$ *f g* ::('*a* ⇒ *real*). *S* ≠ {} ⟹ *finite S* ⟹ ($\bigwedge$*x*. *x* ∈ *S* ⟹ *f x* < *g x*) ⟹ ($\sum$ *x*∈*S*. *f x*) < ($\sum$ *x*∈*S*. *g x*)
by(*rule sum-strict-mono*, *auto*)
have *L2*:$\bigwedge$*f S*. *L2-set* (λ*x*. *norm*(*f x*)) *S* ≤ ($\sum$ *x* ∈ *S*. *norm*(*f x*))
using *L2-set-le-sum norm-ge-zero* **by** *metis*
have *L2'*:$\bigwedge$*y*. (*L2-set* (λ*i*. *norm*(*f i x1 y* − *f i x2 y*)) *UNIV*)/*norm*(*y*) ≤ ($\sum$ *i*∈*UNIV*. *norm*(*f i x1 y* − *f i x2 y*))/*norm*(*y*)
subgoal for *y*
using *L2*[*of* (λ *x*. *f x x1 y* − *f x x2 y*) *UNIV*]
by (*auto simp add*: *divide-right-mono*)
done
have $\bigwedge$*i*. (*SUP y*∈*UNIV*. *norm*((*f i x1* − *f i x2*) *y*)/*norm*(*y*)) = *norm*(*f i x1* − *f i x2*)
by (*simp add*: *onorm-def norm-blinfun.rep-eq*)
then have *each-norm*:$\bigwedge$*i*. (*SUP y*∈*UNIV*. *norm*(*f i x1 y* − *f i x2 y*)/*norm*(*y*)) = *norm*(*f i x1* − *f i x2*)
by (*metis* (*no-types*, *lifting*) *SUP-cong blinfun.diff-left*)
have *bounded-linear*:$\bigwedge$*i*. *bounded-linear* (λ*y*. *f i x1 y* − *f i x2 y*)
by (*simp add*: *blinfun.bounded-linear-right bounded-linear-sub*)
have *each-bound*:$\bigwedge$*i*. *bdd-above* ((λ*y*. *norm*(*f i x1 y* − *f i x2 y*)/*norm*(*y*)) ‘ *UNIV*)
using *bounded-linear* **unfolding** *bdd-above-def*
proof −
fix *i* :: '*a*
{ fix *rr* :: *real* ⇒ *real*
have $\bigwedge$*a r*. *norm* (*blinfun-apply* (*f a x1*) *r* − *blinfun-apply* (*f a x2*) *r*) / *norm r* ≤ *norm* (*f a x1* − *f a x2*)
by (*metis le-norm-blinfun minus-blinfun.rep-eq*)
then have $\bigwedge$*r R*. *r* ∉ (λ*r*. *norm* (*blinfun-apply* (*f i x1*) *r* − *blinfun-apply* (*f i x2*) *r*) / *norm r*) ‘ *R* ∨ *r* ≤ *norm* (*f i x1* − *f i x2*)
by *blast*
then have ∃ *r*. *rr r* ≤ *r* ∨ *rr r* ∉ *range* (λ*r*. *norm* (*blinfun-apply* (*f i x1*) *r* − *blinfun-apply* (*f i x2*) *r*) / *norm r*)
by *blast* **}**
then show ∃ *r*. ∀ *ra*∈*range* (λ*r*. *norm* (*blinfun-apply* (*f i x1*) *r* − *blinfun-apply* (*f i x2*) *r*) / *norm r*). *ra* ≤ *r*
by *meson*
qed
have *bdd-above*:(*bdd-above* ((λ*y*. ($\sum$ *i*∈*UNIV*. *norm*(*f i x1 y* − *f i x2 y*)/*norm*(*y*))) ‘ *UNIV*))
using *each-bound* **unfolding** *bdd-above-def* **apply** *auto*
proof −
assume *each*:($\bigwedge$*i*. ∃ *M*. ∀ *x*. |*blinfun-apply* (*f i x1*) *x* − *blinfun-apply* (*f i x2*) *x*| / *norm x* ≤ *M*)
let *?boundP* = (λ*i M*. ∀ *x*. |*blinfun-apply* (*f i x1*) *x* − *blinfun-apply* (*f i x2*) *x*| / *norm x* ≤ *M*)
let *?bound* = (λ*i*. *SOME x*. *?boundP i x*)
have *bounds*:$\bigwedge$*i*. *?boundP i* (*?bound i*)

subgoal for i **using** *each someI*[*of ?boundP i*] **by** *blast* **done**
let *?bigBound* = $\sum i{\in}(UNIV{::}'a\ set).$ *?bound i*
show $\exists M.\ \forall x.\ (\sum i{\in}UNIV.\ |$*blinfun-apply* (*f i x1*) *x* − *blinfun-apply* (*f i x2*) *x*$|$ / *norm x*) ≤ *M*
apply(*rule exI*[**where** *x*= *?bigBound*])
by(*auto simp add*: *bounds sum-mono*)
qed
have *bdd-above*:(*bdd-above* ((λ*y*. ($\sum i{\in}UNIV.$ *norm*(*f i x1 y* − *f i x2 y*))/*norm*(*y*)) ‘ *UNIV*))
using *bdd-above* **unfolding** *bdd-above-def* **apply** *auto*
proof −
fix *M* :: *real*
assume *a1*: ∀ *x*. ($\sum i{\in}UNIV.$ |*blinfun-apply* (*f i x1*) *x* − *blinfun-apply* (*f i x2*) *x*| / *norm x*) ≤ *M*
{ fix *bb* :: *real* ⇒ *'b*
have ⋀*b*. ($\sum a{\in}UNIV.$ |*blinfun-apply* (*f a x1*) *b* − *blinfun-apply* (*f a x2*) *b*|) / *norm b* ≤ *M*
using *a1* **by** (*simp add*: *sum-divide-distrib*)
then have ∃ *r*. ($\sum a{\in}UNIV.$ |*blinfun-apply* (*f a x1*) (*bb r*) − *blinfun-apply* (*f a x2*) (*bb r*)|) / *norm* (*bb r*) ≤ *r*
by *blast* **}**
then show ∃ *r*. ∀ *b*. ($\sum a{\in}UNIV.$ |*blinfun-apply* (*f a x1*) *b* − *blinfun-apply* (*f a x2*) *b*|) / *norm b* ≤ *r*
by *meson*
qed
have *dist* (*?f x2*) (*?f x1*) = *norm*((*?f x2*) − (*?f x1*))
by (*simp add*: *dist-blinfun-def*)
moreover have ... = (*SUP y*∈*UNIV*. *norm*(*?f x1 y* − *?f x2 y*)/*norm*(*y*))
by (*metis* (*no-types*, *lifting*) *SUP-cong blinfun.diff-left norm-blinfun.rep-eq norm-minus-commute onorm-def*)
moreover have ... = (*SUP y*∈*UNIV*. (*L2-set* (λ*i*. *norm*(*f i x1 y* − *f i x2 y*)) *UNIV*)/*norm*(*y*))
using *euclid* **by** *auto*
moreover have ... ≤ (*SUP y*∈*UNIV*. ($\sum i{\in}UNIV.$ *norm*(*f i x1 y* − *f i x2 y*))/*norm*(*y*))
using *L2' SUP-cong SUP-leq bdd-above* **by** *auto*
moreover have ... = (*SUP y*∈*UNIV*. ($\sum i{\in}UNIV.$ *norm*(*f i x1 y* − *f i x2 y*)/*norm*(*y*)))
by (*simp add*: *sum-divide-distrib*)
moreover have ... ≤ ($\sum i{\in}UNIV.$ (*SUP y*∈*UNIV*. *norm*(*f i x1 y* − *f i x2 y*)/*norm*(*y*)))
by(*rule SUP-sum-comm*[*OF finite nonemptyB*, *of x1 x2*])
moreover have ... = ($\sum i{\in}UNIV.$ *norm*(*f i x1* − *f i x2*))
using *each-norm* **by** *simp*
moreover have ... = ($\sum i{\in}UNIV.$ *dist*(*f i x1*) (*f i x2*))
by (*simp add*: *dist-blinfun-def*)
moreover have ... < ($\sum i{\in}(UNIV{::}'a\ set).$ ε / *?n*)
using *conts'''*[*OF ne dists*] **using** *SUM-leq*[*OF nonempty*, *of* (λ*i*. *dist* (*f i x1*) (*f i x2*)) (λ*i*. ε / *?n*)]

by (*simp add*: *dist-commute*)
moreover have ... = ε
by(*auto*)
ultimately show *dist* (*?f x2*) (*?f x1*) < ε
by *linarith*
qed
then show $\exists s{>}0.\ \forall x2.\ x2 \neq x1 \wedge$ *dist x2 x1* < $s \longrightarrow$ *dist* (*blinfun-vec* ($\lambda i.$ *f i x2*)) (*blinfun-vec* ($\lambda i.$ *f i x1*)) < ε
using δ **by** *blast*
qed

lemma *has-derivative-vec*[*derivative-intros*]:
assumes $\bigwedge i.$ (($\lambda x.$ *f i x*) *has-derivative* ($\lambda h.$ *f' i h*)) *F*
shows (($\lambda x.$ χ *i. f i x*) *has-derivative* ($\lambda h.$ χ *i. f' i h*)) *F*
proof −
have ∗: (χ *i. f i x*) = ($\sum i{\in}UNIV.$ *axis i* (*f i x*)) (χ *i. f' i x*) = ($\sum i{\in}UNIV.$ *axis i* (*f' i x*)) **for** *x*
by (*simp-all add*: *axis-def sum.If-cases vec-eq-iff*)
show *?thesis*
unfolding ∗
by (*intro has-derivative-sum bounded-linear.has-derivative*[*OF bounded-linear-axis*] *assms*)
qed

lemma *has-derivative-proj*:
fixes *j*::('*a*::*finite*)
fixes *f*::'*a* ⇒ *real* ⇒ *real*
assumes *assm*:(($\lambda x.$ χ *i. f i x*) *has-derivative* ($\lambda h.$ χ *i. f' i h*)) *F*
shows (($\lambda x.$ *f j x*) *has-derivative* ($\lambda h.$ *f' j h*)) *F*
proof −
have *bounded-proj*:*bounded-linear* (λ *x*::(*real*^'*a*). *x* $ *j*)
by (*simp add*: *bounded-linear-vec-nth*)
show *?thesis*
using *bounded-linear.has-derivative*[*OF bounded-proj*, *of* ($\lambda x.$ χ *i. f i x*) ($\lambda h.$ χ *i. f' i h*), *OF assm*]
by *auto*
qed

lemma *has-derivative-proj'*:
fixes *i*::'*a*::*finite*
shows $\forall x.$ ((λ *x. x* $ *i*) *has-derivative* (λx::(*real*^'*a*). *x* $ *i*)) (*at x*)
proof −
have *bounded-proj*:*bounded-linear* (λ *x*::(*real*^'*a*). *x* $ *i*)
by (*simp add*: *bounded-linear-vec-nth*)
show *?thesis*
using *bounded-proj* **unfolding** *has-derivative-def* **by** *auto*
qed

lemma *constant-when-zero*:

fixes *v*::*real* ⇒ (*real*, *'i*::*finite*) *vec*
assumes *x0*: (*v t0*) $ *i* = *x0*
assumes *sol*: (*v solves-ode f*) *T S*
assumes *f0*: ⋀*s x*. *s* ∈ *T* ⟹ *f s x* $ *i* = *0*
assumes *t0*:*t0* ∈ *T*
assumes *t*:*t* ∈ *T*
assumes *convex*:*convex T*
shows *v t* $ *i* = *x0*
proof −
from *solves-odeD*[*OF sol*]
have *deriv*: (*v has-vderiv-on* (λ*t*. *f t* (*v t*))) *T* **by** *simp*
then have ((λ*t*. *v t* $ *i*) *has-vderiv-on* (λ*t*. *0*)) *T*
using *f0*
by (*auto simp*: *has-vderiv-on-def has-vector-derivative-def cart-eq-inner-axis intro*!: *derivative-eq-intros*)
from *has-vderiv-on-zero-constant*[*OF convex this*]
obtain *c* **where** *c*:⋀*x*. *x* ∈ *T* ⟹ *v x* $ *i* = *c* **by** *blast*
with *x0* **have** *c* = *x0 v t* $ *i* = *c*
using *t t0 c x0* **by** *blast*+
then show *?thesis* **by** *simp*
qed

lemma
solves-ode-subset:
assumes *x*: (*x solves-ode f*) *T X*
assumes *s*: *S* ⊆ *T*
shows (*x solves-ode f*) *S X*
apply(*rule solves-odeI*)
using *has-vderiv-on-subset s solves-ode-vderivD x* **apply** *force*
using *assms* **by** (*auto intro*!: *solves-odeI dest*!: *solves-ode-domainD*)

lemma
solves-ode-supset-range:
assumes *x*: (*x solves-ode f*) *T X*
assumes *y*: *X* ⊆ *Y*
shows (*x solves-ode f*) *T Y*
apply(*rule solves-odeI*)
using *has-vderiv-on-subset y solves-ode-vderivD x* **apply** *force*
using *assms* **by** (*auto intro*!: *solves-odeI dest*!: *solves-ode-domainD*)

lemma
usolves-ode-subset:
assumes *x*: (*x usolves-ode f from t0*) *T X*
assumes *s*: *S* ⊆ *T*
assumes *t0*: *t0* ∈ *S*
assumes *S*: *is-interval S*
shows (*x usolves-ode f from t0*) *S X*
proof (*rule usolves-odeI*)
note *usolves-odeD*[*OF x*]

show (*x solves-ode f*) *S X* **by** (*rule solves-ode-subset*; *fact*)
show *t0* ∈ *S is-interval S* **by**(*fact+*)
fix *z t*
assume *s*: {*t0* −− *t*} ⊆ *S* **and** *z*: (*z solves-ode f*) {*t0* −− *t*} *X* **and** *z0*: *z t0* = *x t0*
then have *t0* ∈ {*t0* −− *t*} *is-interval* {*t0* −− *t*}
by *auto*
moreover note *s*
moreover have (*z solves-ode f*) {*t0*−−*t*} *X*
using *solves-odeD*[*OF z*] ‹*S* ⊆ *T*›
by (*intro solves-ode-subset-range*[*OF z*]) *force*
moreover note *z0*
moreover have *t* ∈ {*t0* −− *t*} **by** *simp*
ultimately show *z t* = *x t*
by (*meson* ‹⋀*z ta T'*. ⟦*t0* ∈ *T'*; *is-interval T'*; *T'* ⊆ *T*; (*z solves-ode f*) *T' X*; *z t0* = *x t0*; *ta* ∈ *T'*⟧ ⟹ *z ta* = *x ta*› *assms*(*2*) *dual-order.trans*)
qed

— Example of using lemmas above to show a lemma that could be useful for dL: The constant ODE
— 0 does not change the state.
lemma *example*:
fixes *x t*::*real* **and** *i*::('*sz*::*finite*)
assumes *t* > *0*
shows *x* = (*ll-on-open.flow UNIV* (λ*t*. λ*x*. χ (*i*::('*sz*::*finite*)). *0*) *UNIV 0* (χ *i*. *x*) *t*) $ *i*
proof −
let *?T* = *UNIV*
let *?f* = (λ*t*. λ*x*. χ *i*::('*sz*::*finite*). *0*)
let *?X* = *UNIV*
let *?t0.0* = *0*
let *?x0.0* = χ *i*::('*sz*::*finite*). *x*
interpret *ll*: *ll-on-open UNIV* (λ*t x*. χ *i*::('*sz*::*finite*). *0*) *UNIV*
using *gt-ex*
by *unfold-locales*
(*auto simp*: *interval-def continuous-on-def local-lipschitz-def intro*!: *lipschitz-intros*)
have *foo1*:*?t0.0* ∈ *?T* **by** *auto*
have *foo2*:*?x0.0* ∈ *?X* **by** *auto*
let *?v* = *ll.flow ?t0.0 ?x0.0*
from *ll.flow-solves-ode*[*OF foo1 foo2*]
have *solves*:(*ll.flow ?t0.0 ?x0.0 solves-ode ?f*) (*ll.existence-ivl ?t0.0 ?x0.0*) *?X* **by** (*auto*)
then have *solves*:(*?v solves-ode ?f*) (*ll.existence-ivl ?t0.0 ?x0.0*) *?X* **by** *auto*
have *thex0*: (*?v ?t0.0*) $ (*i*::('*sz*::*finite*)) = *x* **by** *auto*
have *sol-help*: (*?v solves-ode ?f*) (*ll.existence-ivl ?t0.0 ?x0.0*) *?X* **using** *solves* **by** *auto*
have *ivl*:*ll.existence-ivl ?t0.0 ?x0.0* = *UNIV*
by (*rule ll.existence-ivl-eq-domain*)
(*auto intro*!: *exI*[**where** *x*=*0*] *simp*: *vec-eq-iff*)

```
  have sol: (?v solves-ode ?f) UNIV ?X using solves ivl by auto
  have thef0: ⋀t x. ?f t x $ i = 0 by auto
  from constant-when-zero [OF thex0 sol thef0]
  have ?v t $ i = x
    by auto
  thus ?thesis by auto
 qed

lemma MVT-ivl:
  fixes f::'a::ordered-euclidean-space⇒'b::ordered-euclidean-space
  assumes fderiv: ⋀x. x ∈ D ⟹ (f has-derivative J x) (at x within D)
  assumes J-ivl: ⋀x. x ∈ D ⟹ J x u ≥ J0
  assumes line-in: ⋀x. x ∈ {0..1} ⟹ a + x *R u ∈ D
  shows f (a + u) − f a ≥ J0
proof −
  from MVT-corrected[OF fderiv line-in] obtain t where
    t: t∈Basis → {0<..<1} and
    mvt: f (a + u) − f a = (∑ i∈Basis. (J (a + t i *R u) u • i) *R i)
    by auto
  note mvt
  also have ... ≥ J0
  proof −
    have J: ⋀i. i ∈ Basis ⟹ J0 ≤ J (a + t i *R u) u
      using J-ivl t line-in by (auto simp: Pi-iff)
    show ?thesis
      using J
      unfolding atLeastAtMost-iff eucl-le[where 'a='b]
      by auto
  qed
  finally show ?thesis .
qed

lemma MVT-ivl':
  fixes f::'a::ordered-euclidean-space⇒'b::ordered-euclidean-space
  assumes fderiv: (⋀x. x ∈ D ⟹ (f has-derivative J x) (at x within D))
  assumes J-ivl: ⋀x. x ∈ D ⟹ J x (a − b) ≥ J0
  assumes line-in: ⋀x. x ∈ {0..1} ⟹ b + x *R (a − b) ∈ D
  shows f a ≥ f b + J0
proof −
  have f (b + (a − b)) − f b ≥ J0
    apply (rule MVT-ivl[OF fderiv ])
      apply assumption
     apply (rule J-ivl) apply assumption
    using line-in
    apply (auto simp: diff-le-eq le-diff-eq ac-simps)
    done
  thus ?thesis
    by (auto simp: diff-le-eq le-diff-eq ac-simps)
qed
```

end
theory *Syntax*
imports
Complex-Main
Ids
begin

3 Syntax

We define the syntax of dL terms, formulas and hybrid programs. As in CADE'15, the syntax allows arbitrarily nested differentials. However, the semantics of such terms is very surprising (e.g. (x')' is zero in every state), so we define predicates dfree and dsafe to describe terms with no differentials and no nested differentials, respectively.

In keeping with the CADE'15 presentation we currently make the simplifying assumption that all terms are smooth, and thus division and arbitrary exponentiation are absent from the syntax. Several other standard logical constructs are implemented as derived forms to reduce the soundness burden.

The types of formulas and programs are parameterized by three finite types ('a, 'b, 'c) used as identifiers for function constants, context constants, and everything else, respectively. These type variables are distinct because some substitution operations affect one type variable while leaving the others unchanged. Because these types will be finite in practice, it is more useful to think of them as natural numbers that happen to be represented as types (due to HOL's lack of dependent types). The types of terms and ODE systems follow the same approach, but have only two type variables because they cannot contain contexts.

datatype $('a, 'c)$ *trm* =
— Real-valued variables given meaning by the state and modified by programs.
Var $'c$
— N.B. This is technically more expressive than true dL since most reals
— can't be written down.
| *Const real*
— A function (applied to its arguments) consists of an identifier for the function
— and a function $'c \Rightarrow ('a, 'c)$ *trm* (where $'c$ is a finite type) which specifies one
— argument of the function for each element of type $'c$. To simulate a function with
— less than $'c$ arguments, set the remaining arguments to a constant, such as *Const 0*
| *Function* $'a$ $'c \Rightarrow ('a, 'c)$ *trm* (‹$f›)
| *Plus* $('a, 'c)$ *trm* $('a, 'c)$ *trm*
| *Times* $('a, 'c)$ *trm* $('a, 'c)$ *trm*
— A (real-valued) variable standing for a differential, such as x', given meaning by the state

— and modified by programs.
| *DiffVar* 'c (‹$''›)
— The differential of an arbitrary term $(\vartheta)'$
| *Differential* ('a, 'c) *trm*

datatype('a, 'c) *ODE* =
— Variable standing for an ODE system, given meaning by the interpretation
OVar 'c
— Singleton ODE defining $x' = \vartheta$, where ϑ may or may not contain x
— (but must not contain differentials)
| *OSing* 'c ('a, 'c) *trm*
— The product *OProd ODE1 ODE2* composes two ODE systems in parallel, e.g.
— *OProd* $(x' = y)$ $(y' = -x)$ is the system $\{x' = y, y' = -x\}$
| *OProd* ('a, 'c) *ODE* ('a, 'c) *ODE*

datatype ('a, 'b, 'c) *hp* =
— Variables standing for programs, given meaning by the interpretation.
Pvar 'c (‹$α›)
— Assignment to a real-valued variable $x := \vartheta$
| *Assign* 'c ('a, 'c) *trm* (**infixr** ‹:=› *10*)
— Assignment to a differential variable
| *DiffAssign* 'c ('a, 'c) *trm*
— Program $?\varphi$ succeeds iff φ holds in current state.
| *Test* ('a, 'b, 'c) *formula* (‹?›)
— An ODE program is an ODE system with some evolution domain.
| *EvolveODE* ('a, 'c) *ODE* ('a, 'b, 'c) *formula*
— Non-deterministic choice between two programs a and b
| *Choice* ('a, 'b, 'c) *hp* ('a, 'b, 'c) *hp* (**infixl** ‹∪∪› *10*)
— Sequential composition of two programs a and b
| *Sequence* ('a, 'b, 'c) *hp* ('a, 'b, 'c) *hp* (**infixr** ‹;;› *8*)
— Nondeterministic repetition of a program a, zero or more times.
| *Loop* ('a, 'b, 'c) *hp* (‹-**›)

and ('a, 'b, 'c) *formula* =
Geq ('a, 'c) *trm* ('a, 'c) *trm*
| *Prop* 'c 'c ⇒ ('a, 'c) *trm* (‹$φ›)
| *Not* ('a, 'b, 'c) *formula* (‹!›)
| *And* ('a, 'b, 'c) *formula* ('a, 'b, 'c) *formula* (**infixl** ‹&&› *8*)
| *Exists* 'c ('a, 'b, 'c) *formula*
— $\langle\alpha\rangle\varphi$ iff exists run of α where φ is true in end state
| *Diamond* ('a, 'b, 'c) *hp* ('a, 'b, 'c) *formula* (‹(⟨ - ⟩ -)› *10*)
— Contexts C are symbols standing for functions from (the semantics of) formulas to
— (the semantics of) formulas, thus $C(\varphi)$ is another formula. While not necessary
— in terms of expressiveness, contexts allow for more efficient reasoning principles.
| *InContext* 'b ('a, 'b, 'c) *formula*

— Derived forms
definition *Or* :: ('a, 'b, 'c) *formula* ⇒ ('a, 'b, 'c) *formula* ⇒ ('a, 'b, 'c) *formula*

(**infixl** ‹||› *7*)
where *Or P Q* = *Not* (*And* (*Not P*) (*Not Q*))

definition *Implies* :: ('*a*, '*b*, '*c*) *formula* ⇒ ('*a*, '*b*, '*c*) *formula* ⇒ ('*a*, '*b*, '*c*) *formula*
(**infixr** ‹→› *10*)
where *Implies P Q* = *Or Q* (*Not P*)

definition *Equiv* :: ('*a*, '*b*, '*c*) *formula* ⇒ ('*a*, '*b*, '*c*) *formula* ⇒ ('*a*, '*b*, '*c*) *formula*
(**infixl** ‹↔› *10*)
where *Equiv P Q* = *Or* (*And P Q*) (*And* (*Not P*) (*Not Q*))

definition *Forall* :: '*c* ⇒ ('*a*, '*b*, '*c*) *formula* ⇒ ('*a*, '*b*, '*c*) *formula*
where *Forall x P* = *Not* (*Exists x* (*Not P*))

definition *Equals* :: ('*a*, '*c*) *trm* ⇒ ('*a*, '*c*) *trm* ⇒ ('*a*, '*b*, '*c*) *formula*
where *Equals* ϑ ϑ′ = ((*Geq* ϑ ϑ′) && (*Geq* ϑ′ ϑ))

definition *Greater* :: ('*a*, '*c*) *trm* ⇒ ('*a*, '*c*) *trm* ⇒ ('*a*, '*b*, '*c*) *formula*
where *Greater* ϑ ϑ′ = ((*Geq* ϑ ϑ′) && (*Not* (*Geq* ϑ′ ϑ)))

definition *Box* :: ('*a*, '*b*, '*c*) *hp* ⇒ ('*a*, '*b*, '*c*) *formula* ⇒ ('*a*, '*b*, '*c*) *formula*
(‹([[-]]-)› *10*)
where *Box* α *P* = *Not* (*Diamond* α (*Not P*))

definition *TT* ::('*a*,'*b*,'*c*) *formula*
where *TT* = *Geq* (*Const 0*) (*Const 0*)

definition *FF* ::('*a*,'*b*,'*c*) *formula*
where *FF* = *Geq* (*Const 0*) (*Const 1*)

type-synonym ('*a*,'*b*,'*c*) *sequent* = ('*a*,'*b*,'*c*) *formula list* ∗ ('*a*,'*b*,'*c*) *formula list*
— Rule: assumptions, then conclusion
type-synonym ('*a*,'*b*,'*c*) *rule* = ('*a*,'*b*,'*c*) *sequent list* ∗ ('*a*,'*b*,'*c*) *sequent*

— silliness to enable proving disequality lemmas
primrec *sizeF*::('*sf*,'*sc*, '*sz*) *formula* ⇒ *nat*
and *sizeP*::('*sf*,'*sc*, '*sz*) *hp* ⇒ *nat*
where
sizeP (*Pvar a*) = *1*
| *sizeP* (*Assign x* ϑ) = *1*
| *sizeP* (*DiffAssign x* ϑ) = *1*
| *sizeP* (*Test* φ) = *Suc* (*sizeF* φ)
| *sizeP* (*EvolveODE ODE* φ) = *Suc* (*sizeF* φ)
| *sizeP* (*Choice* α β) = *Suc* (*sizeP* α + *sizeP* β)
| *sizeP* (*Sequence* α β) = *Suc* (*sizeP* α + *sizeP* β)
| *sizeP* (*Loop* α) = *Suc* (*sizeP* α)
| *sizeF* (*Geq p q*) = *1*
| *sizeF* (*Prop p args*) = *1*

| *sizeF* (*Not p*) = *Suc* (*sizeF p*)
| *sizeF* (*And p q*) = *sizeF p* + *sizeF q*
| *sizeF* (*Exists x p*) = *Suc* (*sizeF p*)
| *sizeF* (*Diamond p q*) = *Suc* (*sizeP p* + *sizeF q*)
| *sizeF* (*InContext C φ*) = *Suc* (*sizeF φ*)

lemma *sizeF-diseq*:*sizeF p* ≠ *sizeF q* ⟹ *p* ≠ *q* **by** *auto*

named-theorems *expr-diseq Structural disequality rules for expressions*
lemma [*expr-diseq*]:*p* ≠ *And p q* **by**(*induction p*, *auto*)
lemma [*expr-diseq*]:*q* ≠ *And p q* **by**(*induction q*, *auto*)
lemma [*expr-diseq*]:*p* ≠ *Not p* **by**(*induction p*, *auto*)
lemma [*expr-diseq*]:*p* ≠ *Or p q* **by**(*rule sizeF-diseq*, *auto simp add*: *Or-def*)
lemma [*expr-diseq*]:*q* ≠ *Or p q* **by**(*rule sizeF-diseq*, *auto simp add*: *Or-def*)
lemma [*expr-diseq*]:*p* ≠ *Implies p q* **by**(*rule sizeF-diseq*, *auto simp add*: *Implies-def Or-def*)
lemma [*expr-diseq*]:*q* ≠ *Implies p q* **by**(*rule sizeF-diseq*, *auto simp add*: *Implies-def Or-def*)
lemma [*expr-diseq*]:*p* ≠ *Equiv p q* **by**(*rule sizeF-diseq*, *auto simp add*: *Equiv-def Or-def*)
lemma [*expr-diseq*]:*q* ≠ *Equiv p q* **by**(*rule sizeF-diseq*, *auto simp add*: *Equiv-def Or-def*)
lemma [*expr-diseq*]:*p* ≠ *Exists x p* **by**(*induction p*, *auto*)
lemma [*expr-diseq*]:*p* ≠ *Diamond a p* **by**(*induction p*, *auto*)
lemma [*expr-diseq*]:*p* ≠ *InContext C p* **by**(*induction p*, *auto*)

— A predicational is like a context with no argument, i.e. a variable standing for a
— state-dependent formula, given meaning by the interpretation. This differs from a predicate
— because predicates depend only on their arguments (which might then indirectly depend on the state).
— We encode a predicational as a context applied to a formula whose truth value is constant with
— respect to the state (specifically, always true)
fun *Predicational* :: *'b* ⇒ (*'a*, *'b*, *'c*) *formula* (‹*Pc*›)
where *Predicational P* = *InContext P* (*Geq* (*Const 0*) (*Const 0*))

— Abbreviations for common syntactic constructs in order to make axiom definitions, etc. more
— readable.
context *ids* **begin**
— "Empty" function argument tuple, encoded as tuple where all arguments assume a constant value.
definition *empty*:: *'b* ⇒ (*'a*, *'b*) *trm*
where *empty* ≡ λ*i*.(*Const 0*)

— Function argument tuple with (effectively) one argument, where all others have a constant value.
fun *singleton* :: (*'a*, *'sz*) *trm* ⇒ (*'sz* ⇒ (*'a*, *'sz*) *trm*)

where *singleton t i* = (*if i* = *vid1 then t else* (*Const 0*))

lemma *expand-singleton*:*singleton t* = (λ*i*. (*if i* = *vid1 then t else* (*Const 0*)))
by *auto*

— Function applied to one argument
definition *f1*::*'sf* ⇒ *'sz* ⇒ (*'sf*,*'sz*) *trm*
where *f1 f x* = *Function f* (*singleton* (*Var x*))

— Function applied to zero arguments (simulates a constant symbol given meaning by the interpretation)
definition *f0*::*'sf* ⇒ (*'sf*,*'sz*) *trm*
where *f0 f* = *Function f empty*

— Predicate applied to one argument
definition *p1*::*'sz* ⇒ *'sz* ⇒ (*'sf*, *'sc*, *'sz*) *formula*
where *p1 p x* = *Prop p* (*singleton* (*Var x*))

— Predicational
definition *P*::*'sc* ⇒ (*'sf*, *'sc*, *'sz*) *formula*
where *P p* = *Predicational p*
end

3.1 Well-Formedness predicates

inductive *dfree* :: (*'a*, *'c*) *trm* ⇒ *bool*
where
dfree-Var: *dfree* (*Var i*)
| *dfree-Const*: *dfree* (*Const r*)
| *dfree-Fun*: (⋀*i*. *dfree* (*args i*)) ⟹ *dfree* (*Function i args*)
| *dfree-Plus*: *dfree* ϑ_1 ⟹ *dfree* ϑ_2 ⟹ *dfree* (*Plus* ϑ_1 ϑ_2)
| *dfree-Times*: *dfree* ϑ_1 ⟹ *dfree* ϑ_2 ⟹ *dfree* (*Times* ϑ_1 ϑ_2)

inductive *dsafe* :: (*'a*, *'c*) *trm* ⇒ *bool*
where
dsafe-Var: *dsafe* (*Var i*)
| *dsafe-Const*: *dsafe* (*Const r*)
| *dsafe-Fun*: (⋀*i*. *dsafe* (*args i*)) ⟹ *dsafe* (*Function i args*)
| *dsafe-Plus*: *dsafe* ϑ_1 ⟹ *dsafe* ϑ_2 ⟹ *dsafe* (*Plus* ϑ_1 ϑ_2)
| *dsafe-Times*: *dsafe* ϑ_1 ⟹ *dsafe* ϑ_2 ⟹ *dsafe* (*Times* ϑ_1 ϑ_2)
| *dsafe-Diff*: *dfree* ϑ ⟹ *dsafe* (*Differential* ϑ)
| *dsafe-DiffVar*: *dsafe* (*$' i*)

— Explictly-written variables that are bound by the ODE. Needed to compute whether
— ODE's are valid (e.g. whether they bind the same variable twice)
fun *ODE-dom*::(*'a*, *'c*) *ODE* ⇒ *'c set*
where
ODE-dom (*OVar c*) = {}

| *ODE-dom* (*OSing x* ϑ) = {*x*}
| *ODE-dom* (*OProd ODE1 ODE2*) = *ODE-dom ODE1* ∪ *ODE-dom ODE2*

inductive *osafe*:: ('*a*, '*c*) *ODE* ⇒ *bool*
where
osafe-Var:*osafe* (*OVar c*)
| *osafe-Sing*:*dfree* ϑ ⟹ *osafe* (*OSing x* ϑ)
| *osafe-Prod*:*osafe ODE1* ⟹ *osafe ODE2* ⟹ *ODE-dom ODE1* ∩ *ODE-dom ODE2* = {} ⟹ *osafe* (*OProd ODE1 ODE2*)

— Programs/formulas without any differential terms. This definition not currently used but may
— be useful in the future.
inductive *hpfree*:: ('*a*, '*b*, '*c*) *hp* ⇒ *bool*
and *ffree*:: ('*a*, '*b*, '*c*) *formula* ⇒ *bool*
where
hpfree (*Pvar x*)
| *dfree e* ⟹ *hpfree* (*Assign x e*)
— Differential programs allowed but not differential terms
| *dfree e* ⟹ *hpfree* (*DiffAssign x e*)
| *ffree P* ⟹ *hpfree* (*Test P*)
— Differential programs allowed but not differential terms
| *osafe ODE* ⟹ *ffree P* ⟹ *hpfree* (*EvolveODE ODE P*)
| *hpfree a* ⟹ *hpfree b* ⟹ *hpfree* (*Choice a b*)
| *hpfree a* ⟹ *hpfree b* ⟹ *hpfree* (*Sequence a b*)
| *hpfree a* ⟹ *hpfree* (*Loop a*)
| *ffree f* ⟹ *ffree* (*InContext C f*)
| (⋀*arg*. *arg* ∈ *range args* ⟹ *dfree arg*) ⟹ *ffree* (*Prop p args*)
| *ffree p* ⟹ *ffree* (*Not p*)
| *ffree p* ⟹ *ffree q* ⟹ *ffree* (*And p q*)
| *ffree p* ⟹ *ffree* (*Exists x p*)
| *hpfree a* ⟹ *ffree p* ⟹ *ffree* (*Diamond a p*)
| *ffree* (*Predicational P*)
| *dfree t1* ⟹ *dfree t2* ⟹ *ffree* (*Geq t1 t2*)

inductive *hpsafe*:: ('*a*, '*b*, '*c*) *hp* ⇒ *bool*
and *fsafe*:: ('*a*, '*b*, '*c*) *formula* ⇒ *bool*
where
hpsafe-Pvar:*hpsafe* (*Pvar x*)
| *hpsafe-Assign*:*dsafe e* ⟹ *hpsafe* (*Assign x e*)
| *hpsafe-DiffAssign*:*dsafe e* ⟹ *hpsafe* (*DiffAssign x e*)
| *hpsafe-Test*:*fsafe P* ⟹ *hpsafe* (*Test P*)
| *hpsafe-Evolve*:*osafe ODE* ⟹ *fsafe P* ⟹ *hpsafe* (*EvolveODE ODE P*)
| *hpsafe-Choice*:*hpsafe a* ⟹ *hpsafe b* ⟹ *hpsafe* (*Choice a b*)
| *hpsafe-Sequence*:*hpsafe a* ⟹ *hpsafe b* ⟹ *hpsafe* (*Sequence a b*)
| *hpsafe-Loop*:*hpsafe a* ⟹ *hpsafe* (*Loop a*)

| *fsafe-Geq*:*dsafe t1* ⟹ *dsafe t2* ⟹ *fsafe* (*Geq t1 t2*)
| *fsafe-Prop*:(⋀*i*. *dsafe* (*args i*)) ⟹ *fsafe* (*Prop p args*)

| *fsafe-Not*:*fsafe p* $\Longrightarrow$ *fsafe* (*Not p*)
| *fsafe-And*:*fsafe p* $\Longrightarrow$ *fsafe q* $\Longrightarrow$ *fsafe* (*And p q*)
| *fsafe-Exists*:*fsafe p* $\Longrightarrow$ *fsafe* (*Exists x p*)
| *fsafe-Diamond*:*hpsafe a* $\Longrightarrow$ *fsafe p* $\Longrightarrow$ *fsafe* (*Diamond a p*)
| *fsafe-InContext*:*fsafe f* $\Longrightarrow$ *fsafe* (*InContext C f*)

— Auto-generated simplifier rules for safety predicates
inductive-simps
dfree-Plus-simps[*simp*]: *dfree* (*Plus a b*)
and *dfree-Times-simps*[*simp*]: *dfree* (*Times a b*)
and *dfree-Var-simps*[*simp*]: *dfree* (*Var x*)
and *dfree-DiffVar-simps*[*simp*]: *dfree* (*DiffVar x*)
and *dfree-Differential-simps*[*simp*]: *dfree* (*Differential x*)
and *dfree-Fun-simps*[*simp*]: *dfree* (*Function i args*)
and *dfree-Const-simps*[*simp*]: *dfree* (*Const r*)

inductive-simps
dsafe-Plus-simps[*simp*]: *dsafe* (*Plus a b*)
and *dsafe-Times-simps*[*simp*]: *dsafe* (*Times a b*)
and *dsafe-Var-simps*[*simp*]: *dsafe* (*Var x*)
and *dsafe-DiffVar-simps*[*simp*]: *dsafe* (*DiffVar x*)
and *dsafe-Fun-simps*[*simp*]: *dsafe* (*Function i args*)
and *dsafe-Diff-simps*[*simp*]: *dsafe* (*Differential a*)
and *dsafe-Const-simps*[*simp*]: *dsafe* (*Const r*)

inductive-simps
osafe-OVar-simps[*simp*]:*osafe* (*OVar c*)
and *osafe-OSing-simps*[*simp*]:*osafe* (*OSing x* ϑ)
and *osafe-OProd-simps*[*simp*]:*osafe* (*OProd ODE1 ODE2*)

inductive-simps
hpsafe-Pvar-simps[*simp*]: *hpsafe* (*Pvar a*)
and *hpsafe-Sequence-simps*[*simp*]: *hpsafe* (*a* ;; *b*)
and *hpsafe-Loop-simps*[*simp*]: *hpsafe* (*a*$**$)
and *hpsafe-ODE-simps*[*simp*]: *hpsafe* (*EvolveODE ODE p*)
and *hpsafe-Choice-simps*[*simp*]: *hpsafe* (*a* $\cup\cup$ *b*)
and *hpsafe-Assign-simps*[*simp*]: *hpsafe* (*Assign x e*)
and *hpsafe-DiffAssign-simps*[*simp*]: *hpsafe* (*DiffAssign x e*)
and *hpsafe-Test-simps*[*simp*]: *hpsafe* (*? p*)

and *fsafe-Geq-simps*[*simp*]: *fsafe* (*Geq t1 t2*)
and *fsafe-Prop-simps*[*simp*]: *fsafe* (*Prop p args*)
and *fsafe-Not-simps*[*simp*]: *fsafe* (*Not p*)
and *fsafe-And-simps*[*simp*]: *fsafe* (*And p q*)
and *fsafe-Exists-simps*[*simp*]: *fsafe* (*Exists x p*)
and *fsafe-Diamond-simps*[*simp*]: *fsafe* (*Diamond a p*)
and *fsafe-Context-simps*[*simp*]: *fsafe* (*InContext C p*)

definition *Ssafe*::(*'sf*,*'sc*,*'sz*) *sequent* $\Rightarrow$ *bool*

where *Ssafe S* $\longleftrightarrow$(($\forall$ *i*. *i* $\geq$ *0* $\longrightarrow$ *i* < *length* (*fst S*) $\longrightarrow$ *fsafe* (*nth* (*fst S*) *i*))
$\wedge$($\forall$ *i*. *i* $\geq$ *0* $\longrightarrow$ *i* < *length* (*snd S*) $\longrightarrow$ *fsafe* (*nth* (*snd S*) *i*)))

definition *Rsafe*::(*'sf*,*'sc*,*'sz*) *rule* $\Rightarrow$ *bool*
where *Rsafe R* $\longleftrightarrow$ (($\forall$ *i*. *i* $\geq$ *0* $\longrightarrow$ *i* < *length* (*fst R*) $\longrightarrow$ *Ssafe* (*nth* (*fst R*) *i*))
$\wedge$ *Ssafe* (*snd R*))

— Basic reasoning principles about syntactic constructs, including inductive principles
lemma *dfree-is-dsafe*: *dfree* ϑ $\Longrightarrow$ *dsafe* ϑ
by (*induction rule*: *dfree.induct*) (*auto intro*: *dsafe.intros*)

lemma *hp-induct* [*case-names Var Assign DiffAssign Test Evolve Choice Compose Star*]:
($\bigwedge$*x*. *P* (\$$\alpha$ *x*)) $\Longrightarrow$
($\bigwedge$*x1 x2*. *P* (*x1* := *x2*)) $\Longrightarrow$
($\bigwedge$*x1 x2*. *P* (*DiffAssign x1 x2*)) $\Longrightarrow$
($\bigwedge$*x*. *P* (*? x*)) $\Longrightarrow$
($\bigwedge$*x1 x2*. *P* (*EvolveODE x1 x2*)) $\Longrightarrow$
($\bigwedge$*x1 x2*. *P x1* $\Longrightarrow$ *P x2* $\Longrightarrow$ *P* (*x1* $\cup\cup$ *x2*)) $\Longrightarrow$
($\bigwedge$*x1 x2*. *P x1* $\Longrightarrow$ *P x2* $\Longrightarrow$ *P* (*x1* ;; *x2*)) $\Longrightarrow$
($\bigwedge$*x*. *P x* $\Longrightarrow$ *P x***) $\Longrightarrow$
P hp
by(*induction rule*: *hp.induct*) (*auto*)

lemma *fml-induct*:
($\bigwedge$*t1 t2*. *P* (*Geq t1 t2*))
$\Longrightarrow$ ($\bigwedge$*p args*. *P* (*Prop p args*))
$\Longrightarrow$ ($\bigwedge$*p*. *P p* $\Longrightarrow$ *P* (*Not p*))
$\Longrightarrow$ ($\bigwedge$*p q*. *P p* $\Longrightarrow$ *P q* $\Longrightarrow$ *P* (*And p q*))
$\Longrightarrow$ ($\bigwedge$*x p*. *P p* $\Longrightarrow$ *P* (*Exists x p*))
$\Longrightarrow$ ($\bigwedge$*a p*. *P p* $\Longrightarrow$ *P* (*Diamond a p*))
$\Longrightarrow$ ($\bigwedge$*C p*. *P p* $\Longrightarrow$ *P* (*InContext C p*))
$\Longrightarrow$ *P* φ
by (*induction rule*: *formula.induct*) (*auto*)

context *ids* **begin**
lemma *proj-sing1*:(*singleton* ϑ *vid1*) = ϑ
by (*auto*)

lemma *proj-sing2*:*vid1* $\neq$ *y* $\Longrightarrow$ (*singleton* ϑ *y*) = (*Const 0*)
by (*auto*)
end

end
theory *Denotational-Semantics*
imports
Ordinary-Differential-Equations.ODE-Analysis
Lib

Ids
Syntax
begin

3.2 Denotational Semantics

The canonical dynamic semantics of dL are given as a denotational semantics. The important definitions for the denotational semantics are states ν, interpretations I and the semantic functions $[[\psi]]I$, $[[\theta]]I\nu$, $[[\alpha]]I$, which are represented by the Isabelle functions `fml_sem`, `dterm_sem` and `prog_sem`, respectively.

3.3 States

We formalize a state S as a pair $(S_V, S'_V) : R^n \times R^n$, where S_V assigns values to the program variables and S_V' assigns values to their differentials. Function constants are also formalized as having a fixed arity m `(Rvec_dim)` which may differ from n. If a function does not need to have m arguments, any remaining arguments can be uniformly set to 0, which simulates the affect of having functions of less arguments.

Most semantic proofs need to reason about states agreeing on variables. We say Vagree A B V if states A and B have the same values on all variables in V, similarly with VSagree A B V for simple states A and B and Iagree I J V for interpretations I and J.

type-synonym *'a Rvec* = *real^('a::finite)*
— A state specifies one vector of values for unprimed variables x and a second vector for x'
type-synonym *'a state* = *'a Rvec* × *'a Rvec*
— *'a simple-state* is half a state - either the xs or the x's
type-synonym *'a simple-state* = *'a Rvec*

definition *Vagree* :: *'c::finite state* ⇒ *'c state* ⇒ *('c + 'c) set* ⇒ *bool*
where *Vagree ν ν' V* ≡
  (∀ *i*. *Inl i* ∈ *V* ⟶ *fst ν $ i* = *fst ν' $ i*)
∧ (∀ *i*. *Inr i* ∈ *V* ⟶ *snd ν $ i* = *snd ν' $ i*)

definition *VSagree* :: *'c::finite simple-state* ⇒ *'c simple-state* ⇒ *'c set* ⇒ *bool*
where *VSagree ν ν' V* ⟷ (∀ *i* ∈ *V*. (*ν $ i*) = (*ν' $ i*))

— Agreement lemmas
lemma *agree-nil*:*Vagree ν ω* {}
  by (*auto simp add*: *Vagree-def*)

lemma *agree-supset*:*A* ⊇ *B* ⟹ *Vagree ν ν' A* ⟹ *Vagree ν ν' B*
  by (*auto simp add*: *Vagree-def*)

lemma *VSagree-nil*:*VSagree* ν ω {}
by (*auto simp add*: *VSagree-def*)

lemma *VSagree-supset*:$A \supseteq B \Longrightarrow$ *VSagree* ν ν' $A \Longrightarrow$ *VSagree* ν ν' B
by (*auto simp add*: *VSagree-def*)

lemma *VSagree-UNIV-eq*:*VSagree* A B *UNIV* $\Longrightarrow$ $A = B$
unfolding *VSagree-def* **by** (*auto simp add*: *vec-eq-iff*)

lemma *agree-comm*:$\bigwedge A\ B\ V$. *Vagree* A B $V \Longrightarrow$ *Vagree* B A V **unfolding** *Vagree-def* **by** *auto*

lemma *agree-sub*:$\bigwedge \nu\ \omega\ A\ B$. $A \subseteq B \Longrightarrow$ *Vagree* ν ω $B \Longrightarrow$ *Vagree* ν ω A
unfolding *Vagree-def* **by** *auto*

lemma *agree-UNIV-eq*:$\bigwedge \nu\ \omega$. *Vagree* ν ω *UNIV* $\Longrightarrow \nu = \omega$
unfolding *Vagree-def* **by** (*auto simp add*: *vec-eq-iff*)

lemma *agree-UNIV-fst*:$\bigwedge \nu\ \omega$. *Vagree* ν ω (*Inl* ‘ *UNIV*) $\Longrightarrow$ (*fst* ν) = (*fst* ω)
unfolding *Vagree-def* **by** (*auto simp add*: *vec-eq-iff*)

lemma *agree-UNIV-snd*:$\bigwedge \nu\ \omega$. *Vagree* ν ω (*Inr* ‘ *UNIV*) $\Longrightarrow$ (*snd* ν) = (*snd* ω)
unfolding *Vagree-def* **by** (*auto simp add*: *vec-eq-iff*)

lemma *Vagree-univ*:$\bigwedge a\ b\ c\ d$. *Vagree* (a,b) (c,d) *UNIV* $\Longrightarrow a = c \wedge b = d$
by (*auto simp add*: *Vagree-def vec-eq-iff*)

lemma *agree-union*:$\bigwedge \nu\ \omega\ A\ B$. *Vagree* ν ω $A \Longrightarrow$ *Vagree* ν ω $B \Longrightarrow$ *Vagree* ν ω $(A \cup B)$
unfolding *Vagree-def* **by** (*auto simp add*: *vec-eq-iff*)

lemma *agree-trans*:*Vagree* ν μ $A \Longrightarrow$ *Vagree* μ ω $B \Longrightarrow$ *Vagree* ν ω $(A \cap B)$
by (*auto simp add*: *Vagree-def*)

lemma *agree-refl*:*Vagree* ν ν A
by (*auto simp add*: *Vagree-def*)

lemma *VSagree-sub*:$\bigwedge \nu\ \omega\ A\ B$. $A \subseteq B \Longrightarrow$ *VSagree* ν ω $B \Longrightarrow$ *VSagree* ν ω A
unfolding *VSagree-def* **by** *auto*

lemma *VSagree-refl*:*VSagree* ν ν A
by (*auto simp add*: *VSagree-def*)

3.4 Interpretations

For convenience we pretend interpretations contain an extra field called FunctionFrechet specifying the Frechet derivative `(FunctionFrechet f \<nu>)` : $R^m \to R$ for every function in every state. The proposition `(is_interp I)` says that such a derivative actually exists and is continuous (i.e. all functions

are C1-continuous) without saying what the exact derivative is.

The type parameters 'a, 'b, 'c are finite types whose cardinalities indicate the maximum number of functions, contexts, and <everything else defined by the interpretation>, respectively.

record $('a, 'b, 'c)$ *interp* =
  Functions :: $'a \Rightarrow 'c$ *Rvec* $\Rightarrow$ *real*
  Predicates :: $'c \Rightarrow 'c$ *Rvec* $\Rightarrow$ *bool*
  Contexts :: $'b \Rightarrow 'c$ *state set* $\Rightarrow 'c$ *state set*
  Programs :: $'c \Rightarrow ('c$ *state* $*$ $'c$ *state*$)$ *set*
  ODEs :: $'c \Rightarrow 'c$ *simple-state* $\Rightarrow 'c$ *simple-state*
  ODEBV :: $'c \Rightarrow 'c$ *set*

fun *FunctionFrechet* :: $('a$::*finite*, $'b$::*finite*, $'c$::*finite*$)$ *interp* $\Rightarrow 'a \Rightarrow 'c$ *Rvec* $\Rightarrow 'c$ *Rvec* $\Rightarrow$ *real*
  where *FunctionFrechet I i* = (*THE f'*. $\forall$ *x*. (*Functions I i has-derivative f' x*) (*at x*))

— For an interpretation to be valid, all functions must be differentiable everywhere.
definition *is-interp* :: $('a$::*finite*, $'b$::*finite*, $'c$::*finite*$)$ *interp* $\Rightarrow$ *bool*
  where *is-interp I* $\equiv$
  $\forall x.\ \forall i.$ ((*FDERIV* (*Functions I i*) *x* :> (*FunctionFrechet I i x*)) $\wedge$ *continuous-on UNIV* (λx. *Blinfun* (*FunctionFrechet I i x*)))

lemma *is-interpD*:*is-interp I* $\Longrightarrow \forall x.\ \forall i.$ (*FDERIV* (*Functions I i*) *x* :> (*FunctionFrechet I i x*))
  unfolding *is-interp-def* **by** *auto*

— Agreement between interpretations.
definition *Iagree* :: $('a$::*finite*, $'b$::*finite*, $'c$::*finite*$)$ *interp* $\Rightarrow$ $('a$::*finite*, $'b$::*finite*, $'c$::*finite*$)$ *interp* $\Rightarrow$ $('a + 'b + 'c)$ *set* $\Rightarrow$ *bool*
where *Iagree I J V* $\equiv$
  $(\forall i \in V.$
    $(\forall x.\ i = Inl\ x \longrightarrow$ *Functions I x* = *Functions J x*$) \wedge$
    $(\forall x.\ i = Inr\ (Inl\ x) \longrightarrow$ *Contexts I x* = *Contexts J x*$) \wedge$
    $(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow$ *Predicates I x* = *Predicates J x*$) \wedge$
    $(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow$ *Programs I x* = *Programs J x*$) \wedge$
    $(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow$ *ODEs I x* = *ODEs J x*$) \wedge$
    $(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow$ *ODEBV I x* = *ODEBV J x*$))$

lemma *Iagree-Func*:*Iagree I J V* $\Longrightarrow$ *Inl f* $\in$ *V* $\Longrightarrow$ *Functions I f* = *Functions J f*
  unfolding *Iagree-def* **by** *auto*

lemma *Iagree-Contexts*:*Iagree I J V* $\Longrightarrow$ *Inr* (*Inl C*) $\in$ *V* $\Longrightarrow$ *Contexts I C* = *Contexts J C*
  unfolding *Iagree-def* **by** *auto*

lemma *Iagree-Pred*:*Iagree I J V* $\Longrightarrow$ *Inr* (*Inr p*) $\in$ *V* $\Longrightarrow$ *Predicates I p* = *Pred-*

icates J p
unfolding *Iagree-def* **by** *auto*

lemma *Iagree-Prog*:*Iagree I J V* $\Longrightarrow$ *Inr* (*Inr a*) $\in$ *V* $\Longrightarrow$ *Programs I a* = *Programs J a*
unfolding *Iagree-def* **by** *auto*

lemma *Iagree-ODE*:*Iagree I J V* $\Longrightarrow$ *Inr* (*Inr a*) $\in$ *V* $\Longrightarrow$ *ODEs I a* = *ODEs J a*
unfolding *Iagree-def* **by** *auto*

lemma *Iagree-comm*:$\bigwedge$*A B V*. *Iagree A B V* $\Longrightarrow$ *Iagree B A V*
unfolding *Iagree-def* **by** *auto*

lemma *Iagree-sub*:$\bigwedge$*I J A B* . *A* $\subseteq$ *B* $\Longrightarrow$ *Iagree I J B* $\Longrightarrow$ *Iagree I J A*
unfolding *Iagree-def* **by** *auto*

lemma *Iagree-refl*:*Iagree I I A*
by (*auto simp add*: *Iagree-def*)

— Semantics for differential-free terms. Because there are no differentials, depends only on the x variables
— and not the x' variables.
primrec *sterm-sem* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* $\Rightarrow$ ('*a*, '*c*) *trm* $\Rightarrow$ '*c simple-state* $\Rightarrow$ *real*
where
sterm-sem I (*Var x*) *v* = *v* $ *x*
| *sterm-sem I* (*Function f args*) *v* = *Functions I f* (χ *i*. *sterm-sem I* (*args i*) *v*)
| *sterm-sem I* (*Plus t1 t2*) *v* = *sterm-sem I t1 v* + *sterm-sem I t2 v*
| *sterm-sem I* (*Times t1 t2*) *v* = *sterm-sem I t1 v* $*$ *sterm-sem I t2 v*
| *sterm-sem I* (*Const r*) *v* = *r*
| *sterm-sem I* ($' *c*) *v* = *undefined*
| *sterm-sem I* (*Differential d*) *v* = *undefined*

— *frechet I* ϑ ν syntactically computes the frechet derivative of the term ϑ in the interpretation
— *I* at state ν (containing only the unprimed variables). The frechet derivative is a
— linear map from the differential state ν to reals.
primrec *frechet* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* $\Rightarrow$ ('*a*, '*c*) *trm* $\Rightarrow$ '*c simple-state* $\Rightarrow$ '*c simple-state* $\Rightarrow$ *real*
where
frechet I (*Var x*) *v* = (λ*v'*. *v'* $\cdot$ *axis x 1*)
| *frechet I* (*Function f args*) *v* =
(*λv'*. *FunctionFrechet I f* (χ *i*. *sterm-sem I* (*args i*) *v*) (χ *i*. *frechet I* (*args i*) *v v'*))
| *frechet I* (*Plus t1 t2*) *v* = (λ*v'*. *frechet I t1 v v'* + *frechet I t2 v v'*)
| *frechet I* (*Times t1 t2*) *v* =
(*λv'*. *sterm-sem I t1 v* $*$ *frechet I t2 v v'* + *frechet I t1 v v'* $*$ *sterm-sem I t2 v*)
| *frechet I* (*Const r*) *v* = (λ*v'*. *0*)

| *frechet I* ($' *c*) *v* = *undefined*
| *frechet I* (*Differential d*) *v* = *undefined*

definition *directional-derivative* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* ⇒ ('*a*, '*c*) *trm* ⇒ '*c state* ⇒ *real*
where *directional-derivative I t* = (λ*v*. *frechet I t* (*fst v*) (*snd v*))

— Sem for terms that are allowed to contain differentials.
— Note there is some duplication with *sterm-sem*.
primrec *dterm-sem* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* ⇒ ('*a*, '*c*) *trm* ⇒ '*c state* ⇒ *real*
where
dterm-sem I (*Var x*) = (λ*v*. *fst v* $ *x*)
| *dterm-sem I* (*DiffVar x*) = (λ*v*. *snd v* $ *x*)
| *dterm-sem I* (*Function f args*) = (λ*v*. *Functions I f* (χ *i*. *dterm-sem I* (*args i*) *v*))
| *dterm-sem I* (*Plus t1 t2*) = (λ*v*. (*dterm-sem I t1 v*) + (*dterm-sem I t2 v*))
| *dterm-sem I* (*Times t1 t2*) = (λ*v*. (*dterm-sem I t1 v*) ∗ (*dterm-sem I t2 v*))
| *dterm-sem I* (*Differential t*) = (λ*v*. *directional-derivative I t v*)
| *dterm-sem I* (*Const c*) = (λ*v*. *c*)

The semantics of an ODE is the vector field at a given point. ODE's are all time-independent so no time variable is necessary. Terms on the RHS of an ODE must be differential-free, so depends only on the xs.

The safety predicate `osafe` ensures the domains of ODE1 and ODE2 are disjoint, so vector addition is equivalent to saying "take things defined from ODE1 from ODE1, take things defined by ODE2 from ODE2"

fun *ODE-sem*:: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* ⇒ ('*a*, '*c*) *ODE* ⇒ '*c Rvec* ⇒ '*c Rvec*
where
ODE-sem-OVar:*ODE-sem I* (*OVar x*) = *ODEs I x*
| *ODE-sem-OSing*:*ODE-sem I* (*OSing x* ϑ) = (λν. (χ *i*. *if i* = *x then sterm-sem I* ϑ ν *else 0*))
— Note: Could define using *SOME* operator in a way that more closely matches above description,
— but that gets complicated in the *OVar* case because not all variables are bound by the *OVar*
| *ODE-sem-OProd*:*ODE-sem I* (*OProd ODE1 ODE2*) = (λν. *ODE-sem I ODE1* ν + *ODE-sem I ODE2* ν)

— The bound variables of an ODE
fun *ODE-vars* :: ('*a*,'*b*,'*c*) *interp* ⇒ ('*a*, '*c*) *ODE* ⇒ '*c set*
where
ODE-vars I (*OVar c*) = *ODEBV I c*
| *ODE-vars I* (*OSing x* ϑ) = {*x*}
| *ODE-vars I* (*OProd ODE1 ODE2*) = *ODE-vars I ODE1* ∪ *ODE-vars I ODE2*

fun *semBV* ::('*a*, '*b*,'*c*) *interp* ⇒ ('*a*, '*c*) *ODE* ⇒ ('*c* + '*c*) *set*

where *semBV I ODE* = *Inl* ‘ (*ODE-vars I ODE*) ∪ *Inr* ‘ (*ODE-vars I ODE*)

lemma *ODE-vars-lr*:
fixes *x*::*'sz* **and** *ODE*::(*'sf*,*'sz*) *ODE* **and** *I*::(*'sf*,*'sc*,*'sz*) *interp*
shows *Inl x* ∈ *semBV I ODE* ⟷ *Inr x* ∈ *semBV I ODE*
by (*induction ODE*, *auto*)

fun *mk-xode*::(*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* ⇒ (*'a*::*finite*, *'c*::*finite*) *ODE* ⇒ *'c*::*finite simple-state* ⇒ *'c*::*finite state*
where *mk-xode I ODE sol* = (*sol*, *ODE-sem I ODE sol*)

Given an initial state ν and solution to an ODE at some point, construct the resulting state ω. This is defined using the SOME operator because the concrete definition is unwieldy.

definition *mk-v*::(*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* ⇒ (*'a*::*finite*, *'c*::*finite*) *ODE* ⇒ *'c*::*finite state* ⇒ *'c*::*finite simple-state* ⇒ *'c*::*finite state*
where *mk-v I ODE ν sol* = (*THE ω*.
Vagree ω ν (− *semBV I ODE*)
∧ *Vagree ω* (*mk-xode I ODE sol*) (*semBV I ODE*))

— *repv ν x r* replaces the value of (unprimed) variable *x* in the state ν with r
fun *repv* :: *'c*::*finite state* ⇒ *'c* ⇒ *real* ⇒ *'c state*
where *repv v x r* = ((*χ y*. *if x* = *y then r else vec-nth* (*fst v*) *y*), *snd v*)

— *repd ν x' r* replaces the value of (primed) variable *x'* in the state ν with *r*
fun *repd* :: *'c*::*finite state* ⇒ *'c* ⇒ *real* ⇒ *'c state*
where *repd v x r* = (*fst v*, (*χ y*. *if x* = *y then r else vec-nth* (*snd v*) *y*))

— Semantics for formulas, differential formulas, programs.
fun *fml-sem* :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* ⇒ (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *formula* ⇒ *'c*::*finite state set* **and**
prog-sem :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* ⇒ (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *hp* ⇒ (*'c*::*finite state* ∗ *'c*::*finite state*) *set*
where
fml-sem I (*Geq t1 t2*) = {*v*. *dterm-sem I t1 v* ≥ *dterm-sem I t2 v*}
| *fml-sem I* (*Prop P terms*) = {*ν*. *Predicates I P* (*χ i*. *dterm-sem I* (*terms i*) *ν*)}
| *fml-sem I* (*Not φ*) = {*v*. *v* ∉ *fml-sem I φ*}
| *fml-sem I* (*And φ ψ*) = *fml-sem I φ* ∩ *fml-sem I ψ*
| *fml-sem I* (*Exists x φ*) = {*v* | *v r*. (*repv v x r*) ∈ *fml-sem I φ*}
| *fml-sem I* (*Diamond α φ*) = {*ν* | *ν ω*. (*ν*, *ω*) ∈ *prog-sem I α* ∧ *ω* ∈ *fml-sem I φ*}
| *fml-sem I* (*InContext c φ*) = *Contexts I c* (*fml-sem I φ*)

| *prog-sem I* (*Pvar p*) = *Programs I p*
| *prog-sem I* (*Assign x t*) = {(*ν*, *ω*). *ω* = *repv ν x* (*dterm-sem I t ν*)}
| *prog-sem I* (*DiffAssign x t*) = {(*ν*, *ω*). *ω* = *repd ν x* (*dterm-sem I t ν*)}
| *prog-sem I* (*Test φ*) = {(*ν*, *ν*) | *ν*. *ν* ∈ *fml-sem I φ*}
| *prog-sem I* (*Choice α β*) = *prog-sem I α* ∪ *prog-sem I β*
| *prog-sem I* (*Sequence α β*) = *prog-sem I α O prog-sem I β*

| *prog-sem I* (*Loop* α) = (*prog-sem I* α)*
| *prog-sem I* (*EvolveODE ODE* φ) =
  ({(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
      t $\geq$ *0* $\wedge$
      (*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0..t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
      sol 0 = *fst* ν})

context *ids* **begin**
definition *valid* :: (*'sf*, *'sc*, *'sz*) *formula* $\Rightarrow$ *bool*
where *valid* $\varphi \equiv$ ($\forall$ *I*. $\forall$ ν. *is-interp I* $\longrightarrow$ $\nu \in$ *fml-sem I* φ)
end

Because mk_v is defined with the SOME operator, need to construct a state that satisfies `Vagree`$\omega\nu$(−`ODE_vars ODE`)$\wedge$`Vagree`ω(`mk_xode I ODE sol`) (`ODE_vars ODE`)) to do anything useful

fun *concrete-v*::(*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* $\Rightarrow$ (*'a*::*finite*, *'c*::*finite*) *ODE* $\Rightarrow$ *'c*::*finite state* $\Rightarrow$ *'c*::*finite simple-state* $\Rightarrow$ *'c*::*finite state*
where *concrete-v I ODE* ν *sol* =
((χ *i*. (*if Inl i* $\in$ *semBV I ODE then sol else* (*fst* ν)) $ *i*),
 (χ *i*. (*if Inr i* $\in$ *semBV I ODE then ODE-sem I ODE sol else* (*snd* ν)) $ *i*))

lemma *mk-v-exists*:$\exists\omega$. *Vagree* ω ν (− *semBV I ODE*)
$\wedge$ *Vagree* ω (*mk-xode I ODE sol*) (*semBV I ODE*)
  by(*rule exI*[**where** *x*=(*concrete-v I ODE* ν *sol*)], *auto simp add*: *Vagree-def*)

lemma *mk-v-agree*:*Vagree* (*mk-v I ODE* ν *sol*) ν (− *semBV I ODE*)
$\wedge$ *Vagree* (*mk-v I ODE* ν *sol*) (*mk-xode I ODE sol*) (*semBV I ODE*)
  unfolding *mk-v-def*
  apply(*rule theI*[**where** *a*= ((χ *i*. (*if Inl i* $\in$ *semBV I ODE then sol else* (*fst* ν)) $ *i*),
  (χ *i*. (*if Inr i* $\in$ *semBV I ODE then ODE-sem I ODE sol else* (*snd* ν)) $ *i*))])
   using *exE*[*OF mk-v-exists*, *of* ν *I ODE sol*]
   by (*auto simp add*: *Vagree-def vec-eq-iff*)

lemma *mk-v-concrete*:*mk-v I ODE* ν *sol* = ((χ *i*. (*if Inl i* $\in$ *semBV I ODE then sol else* (*fst* ν)) $ *i*),
  (χ *i*. (*if Inr i* $\in$ *semBV I ODE then ODE-sem I ODE sol else* (*snd* ν)) $ *i*))
  apply(*rule agree-UNIV-eq*)
  using *mk-v-agree*[*of I ODE* ν *sol*]
  unfolding *Vagree-def* **by** *auto*

3.5 Trivial Simplification Lemmas

We often want to pretend the definitions in the semantics are written slightly differently than they are. Since the simplifier has some trouble guessing that these are the right simplifications to do, we write them all out explicitly as lemmas, even though they prove trivially.

lemma *svar-case*:
sterm-sem I (*Var x*) = (λ*v*. *v* $ *x*)
by *auto*

lemma *sconst-case*:
sterm-sem I (*Const r*) = (λ*v*. *r*)
by *auto*

lemma *sfunction-case*:
sterm-sem I (*Function f args*) = (λ*v*. *Functions I f* (χ *i*. *sterm-sem I* (*args i*) *v*))
by *auto*

lemma *splus-case*:
sterm-sem I (*Plus t1 t2*) = (λ*v*. (*sterm-sem I t1 v*) + (*sterm-sem I t2 v*))
by *auto*

lemma *stimes-case*:
sterm-sem I (*Times t1 t2*) = (λ*v*. (*sterm-sem I t1 v*) ∗ (*sterm-sem I t2 v*))
by *auto*

lemma *or-sem* [*simp*]:
fml-sem I (*Or* φ ψ) = *fml-sem I* φ ∪ *fml-sem I* ψ
by (*auto simp add*: *Or-def*)

lemma *iff-sem* [*simp*]: (ν ∈ *fml-sem I* (*A* ↔ *B*))
⟷ ((ν ∈ *fml-sem I A*) ⟷ (ν ∈ *fml-sem I B*))
by (*auto simp add*: *Equiv-def*)

lemma *box-sem* [*simp*]:*fml-sem I* (*Box* α φ) = {ν. ∀ ω. (ν, ω) ∈ *prog-sem I* α ⟶ ω ∈ *fml-sem I* φ}
unfolding *Box-def fml-sem.simps*
using *Collect-cong* **by** (*auto*)

lemma *forall-sem* [*simp*]:*fml-sem I* (*Forall x* φ) = {*v*. ∀ *r*. (*repv v x r*) ∈ *fml-sem I* φ}
unfolding *Forall-def fml-sem.simps*
using *Collect-cong* **by** (*auto*)

lemma *greater-sem*[*simp*]:*fml-sem I* (*Greater* ϑ ϑ′) = {*v*. *dterm-sem I* ϑ *v* > *dterm-sem I* ϑ′ *v*}
unfolding *Greater-def* **by** *auto*

lemma *loop-sem*:*prog-sem I* (*Loop* α) = (*prog-sem I* α)∗
by (*auto*)

lemma *impl-sem* [*simp*]: (ν ∈ *fml-sem I* (*A* → *B*))
= ((ν ∈ *fml-sem I A*) ⟶ (ν ∈ *fml-sem I B*))
by (*auto simp add*: *Implies-def*)

lemma *equals-sem* [*simp*]: ($\nu \in$ *fml-sem I* (*Equals* ϑ ϑ'))
= (*dterm-sem I* ϑ ν = *dterm-sem I* ϑ' ν)
by (*auto simp add*: *Equals-def*)

lemma *diamond-sem* [*simp*]: *fml-sem I* (*Diamond* α φ)
= {ν. $\exists$ ω. (ν, ω) $\in$ *prog-sem I* α $\wedge$ $\omega \in$ *fml-sem I* φ}
by *auto*

lemma *tt-sem* [*simp*]:*fml-sem I TT* = *UNIV* **unfolding** *TT-def* **by** *auto*
lemma *ff-sem* [*simp*]:*fml-sem I FF* = {} **unfolding** *FF-def* **by** *auto*

lemma *iff-to-impl*: (($\nu \in$ *fml-sem I A*) $\longleftrightarrow$ ($\nu \in$ *fml-sem I B*))
$\longleftrightarrow$ ((($\nu \in$ *fml-sem I A*) $\longrightarrow$ ($\nu \in$ *fml-sem I B*))
$\wedge$ (($\nu \in$ *fml-sem I B*) $\longrightarrow$ ($\nu \in$ *fml-sem I A*)))
by (*auto*)

fun *seq2fml* :: ('*a*,'*b*,'*c*) *sequent* $\Rightarrow$ ('*a*,'*b*,'*c*) *formula*
where
seq2fml (*ante*,*succ*) = *Implies* (*foldr And ante TT*) (*foldr Or succ FF*)

context *ids* **begin**
fun *seq-sem* ::('*sf*, '*sc*, '*sz*) *interp* $\Rightarrow$ ('*sf*, '*sc*, '*sz*) *sequent* $\Rightarrow$ '*sz state set*
where *seq-sem I S* = *fml-sem I* (*seq2fml S*)

lemma *and-foldl-sem*:$\nu \in$ *fml-sem I* (*foldr And* Γ *TT*) $\Longrightarrow$ ($\bigwedge\varphi$. *List.member* Γ φ $\Longrightarrow$ $\nu \in$ *fml-sem I* φ)
by(*induction* Γ, *auto simp add*: *member-rec*)

lemma *and-foldl-sem-conv*:($\bigwedge\varphi$. *List.member* Γ φ $\Longrightarrow$ $\nu \in$ *fml-sem I* φ) $\Longrightarrow$ $\nu \in$ *fml-sem I* (*foldr And* Γ *TT*)
by(*induction* Γ, *auto simp add*: *member-rec*)

lemma *or-foldl-sem*:*List.member* Γ φ $\Longrightarrow$ $\nu \in$ *fml-sem I* φ $\Longrightarrow$ $\nu \in$ *fml-sem I* (*foldr Or* Γ *FF*)
by(*induction* Γ, *auto simp add*: *member-rec*)

lemma *or-foldl-sem-conv*:$\nu \in$ *fml-sem I* (*foldr Or* Γ *FF*) $\Longrightarrow$ $\exists$ φ. $\nu \in$ *fml-sem I* φ $\wedge$ *List.member* Γ φ
by(*induction* Γ, *auto simp add*: *member-rec*)

lemma *seq-semI'*:($\nu \in$ *fml-sem I* (*foldr And* Γ *TT*) $\Longrightarrow$ $\nu \in$ *fml-sem I* (*foldr Or* Δ *FF*)) $\Longrightarrow$ $\nu \in$ *seq-sem I* (Γ,Δ)
by *auto*

lemma *seq-semD'*:$\bigwedge$*P*. $\nu \in$ *seq-sem I* (Γ,Δ) $\Longrightarrow$ (($\nu \in$ *fml-sem I* (*foldr And* Γ *TT*) $\Longrightarrow$ $\nu \in$ *fml-sem I* (*foldr Or* Δ *FF*)) $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by *simp*

definition *sublist*::'*a list* $\Rightarrow$ '*a list* $\Rightarrow$ *bool*

where *sublist A B* $\equiv (\forall x.$ *List.member A x* $\longrightarrow$ *List.member B x*)

lemma *sublistI*:$(\bigwedge x.$ *List.member A x* $\Longrightarrow$ *List.member B x*$) \Longrightarrow$ *sublist A B*
unfolding *sublist-def* **by** *auto*

lemma Γ*-sub-sem*:*sublist* $\Gamma 1\ \Gamma 2 \Longrightarrow \nu \in$ *fml-sem I* (*foldr And* $\Gamma 2$ *TT*) $\Longrightarrow \nu \in$ *fml-sem I* (*foldr And* $\Gamma 1$ *TT*)
unfolding *sublist-def*
by (*metis and-foldl-sem and-foldl-sem-conv*)

lemma *seq-semI*:*List.member* $\Delta\ \psi \Longrightarrow$((*List.member* $\Gamma\ \varphi \Longrightarrow \nu \in$ *fml-sem I* $\varphi) \Longrightarrow \nu \in$ *fml-sem I* $\psi) \Longrightarrow \nu \in$ *seq-sem I* (Γ,Δ)
apply(*rule seq-semI'*)
using *and-foldl-sem*[*of* ν *I* Γ] *or-foldl-sem* **by** *blast*

lemma *seq-semD*:$\nu \in$ *seq-sem I* (Γ,Δ) $\Longrightarrow$ (*List.member* $\Gamma\ \varphi \Longrightarrow \nu \in$ *fml-sem I* $\varphi) \Longrightarrow \exists \varphi.$ (*List.member* $\Delta\ \varphi$) $\wedge \nu \in$ *fml-sem I* φ
apply(*rule seq-semD'*)
using *and-foldl-sem-conv or-foldl-sem-conv*
by *blast+*

lemma *seq-MP*:$\nu \in$ *seq-sem I* (Γ,Δ) $\Longrightarrow \nu \in$ *fml-sem I* (*foldr And* Γ *TT*) $\Longrightarrow \nu \in$ *fml-sem I* (*foldr Or* Δ *FF*)
by(*induction* Δ, *auto*)

definition *seq-valid*
where *seq-valid S* $\equiv \forall$ *I. is-interp I* $\longrightarrow$ *seq-sem I S* = *UNIV*

Soundness for derived rules is local soundness, i.e. if the premisses are all true in the same interpretation, then the conclusion is also true in that same interpretation.

definition *sound* :: ('*sf*, '*sc*, '*sz*) *rule* $\Rightarrow$ *bool*
where *sound R* $\longleftrightarrow$ ($\forall$ *I. is-interp I* $\longrightarrow$ ($\forall$ *i. i* $\geq$ *0* $\longrightarrow$ *i* $<$ *length* (*fst R*) $\longrightarrow$ *seq-sem I* (*nth* (*fst R*) *i*) = *UNIV*) $\longrightarrow$ *seq-sem I* (*snd R*) = *UNIV*)

lemma *soundI*:$(\bigwedge I.$ *is-interp I* $\Longrightarrow$ $(\bigwedge i.$ *i* $\geq$ *0* $\Longrightarrow$ *i* $<$ *length SG* $\Longrightarrow$ *seq-sem I* (*nth SG i*) = *UNIV*$) \Longrightarrow$ *seq-sem I G* = *UNIV*$) \Longrightarrow$ *sound* (*SG*,*G*)
unfolding *sound-def* **by** *auto*

lemma *soundI'*:$(\bigwedge I\ \nu.$ *is-interp I* $\Longrightarrow$ $(\bigwedge i\ .$ *i* $\geq$ *0* $\Longrightarrow$ *i* $<$ *length SG* $\Longrightarrow \nu \in$ *seq-sem I* (*nth SG i*)$) \Longrightarrow \nu \in$ *seq-sem I G*$) \Longrightarrow$ *sound* (*SG*,*G*)
unfolding *sound-def* **by** *auto*

lemma *soundI-mem*:$(\bigwedge I.$ *is-interp I* $\Longrightarrow$ $(\bigwedge \varphi.$ *List.member SG* $\varphi \Longrightarrow$ *seq-sem I* φ = *UNIV*$) \Longrightarrow$ *seq-sem I C* = *UNIV*$) \Longrightarrow$ *sound* (*SG*,*C*)
apply (*auto simp add*: *sound-def*)
by (*metis in-set-conv-nth in-set-member iso-tuple-UNIV-I seq2fml.simps*)

lemma *soundI-memv*:$(\bigwedge I.$ *is-interp I* $\Longrightarrow$ $(\bigwedge \varphi\ \nu.$ *List.member SG* $\varphi \Longrightarrow \nu \in$

seq-sem I φ) ⟹ (⋀ν. ν ∈ *seq-sem I C*)) ⟹ *sound* (*SG*,*C*)
apply(*rule soundI-mem*)
using *impl-sem* **by** *blast*

lemma *soundI-memv'*:(⋀*I*. *is-interp I* ⟹ (⋀φ ν. *List.member SG φ* ⟹ ν ∈ *seq-sem I φ*) ⟹ (⋀ν. ν ∈ *seq-sem I C*)) ⟹ *R* = (*SG*,*C*) ⟹ *sound R*
using *soundI-mem*
using *impl-sem* **by** *blast*

lemma *soundD-mem*:*sound* (*SG*,*C*) ⟹ (⋀*I*. *is-interp I* ⟹ (⋀φ. *List.member SG φ* ⟹ *seq-sem I φ* = *UNIV*) ⟹ *seq-sem I C* = *UNIV*)
apply (*auto simp add*: *sound-def*)
using *in-set-conv-nth in-set-member iso-tuple-UNIV-I seq2fml.simps*
by (*metis seq2fml.elims*)

lemma *soundD-memv*:*sound* (*SG*,*C*) ⟹ (⋀*I*. *is-interp I* ⟹ (⋀φ ν. *List.member SG φ* ⟹ ν ∈ *seq-sem I φ*) ⟹ (⋀ν. ν ∈ *seq-sem I C*))
using *soundD-mem*
by (*metis UNIV-I UNIV-eq-I*)

end
end
theory *Axioms*
imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
begin context *ids* **begin**

4 Axioms

The uniform substitution calculus is based on a finite list of concrete axioms, which are defined and proved valid (as in sound) in this section. When axioms apply to arbitrary programs or formulas, they mention concrete program or formula variables, which are then instantiated by uniform substitution, as opposed metavariables.

This section contains axioms and rules for propositional connectives and programs other than ODE's. Differential axioms are handled separately because the proofs are significantly more involved.

named-theorems *axiom-defs Axiom definitions*

definition *assign-axiom* :: ('*sf*, '*sc*, '*sz*) *formula*
where [*axiom-defs*]:*assign-axiom* ≡
([[*vid1* := (§*f fid1 empty*)]] (*Prop vid1* (*singleton* (*Var vid1*))))
↔ *Prop vid1* (*singleton* (§*f fid1 empty*))

definition *diff-assign-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*diff-assign-axiom* ≡
([[*DiffAssign vid1* ($f *fid1 empty*)]] (*Prop vid1* (*singleton* (*DiffVar vid1*))))
↔ *Prop vid1* (*singleton* ($f *fid1 empty*))

definition *loop-iterate-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*loop-iterate-axiom* ≡ ([[$α *vid1*∗∗]]*Predicational pid1*)
↔ ((*Predicational pid1*) && ([[$α *vid1*]][[$α *vid1*∗∗]]*Predicational pid1*))

definition *test-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*test-axiom* ≡
([[?($φ *vid2 empty*)]]$φ *vid1 empty*) ↔ (($φ *vid2 empty*) → ($φ *vid1 empty*))

definition *box-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*box-axiom* ≡ (⟨$α *vid1*⟩*Predicational pid1*) ↔ !([[$α *vid1*]]!(*Predicational pid1*))

definition *choice-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*choice-axiom* ≡ ([[$α *vid1* ∪∪ $α *vid2*]]*Predicational pid1*)
↔ (([[$α *vid1*]]*Predicational pid1*) && ([[$α *vid2*]]*Predicational pid1*))

definition *compose-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*compose-axiom* ≡ ([[$α *vid1* ;; $α *vid2*]]*Predicational pid1*) ↔

([[$α *vid1*]][[$α *vid2*]]*Predicational pid1*)

definition *Kaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*Kaxiom* ≡ ([[$α *vid1*]]((*Predicational pid1*) → (*Predicational pid2*)))
→ ([[$α *vid1*]]*Predicational pid1*) → ([[$α *vid1*]]*Predicational pid2*)

definition *Iaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*Iaxiom* ≡
([[($α *vid1*)∗∗]](*Predicational pid1* → ([[$α *vid1*]]*Predicational pid1*)))
→((*Predicational pid1* → ([[($α *vid1*)∗∗]]*Predicational pid1*)))

definition *Vaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*Vaxiom* ≡ ($φ *vid1 empty*) → ([[$α *vid1*]]($φ *vid1 empty*))

4.1 Validity proofs for axioms

Because an axiom in a uniform substitution calculus is an individual formula, proving the validity of that formula suffices to prove soundness

theorem *test-valid*: *valid test-axiom*
by (*auto simp add*: *valid-def test-axiom-def*)

lemma *assign-lem1*:
dterm-sem I (if i = vid1 then Var vid1 else (Const 0))
(vec-lambda (λy. if vid1 = y then Functions I fid1
(vec-lambda (λi. dterm-sem I (empty i) ν)) else vec-nth (fst ν) y), snd ν)
=
dterm-sem I (if i = vid1 then $f fid1 empty else (Const 0)) ν
by (*cases i = vid1*) (*auto simp*: *proj-sing1*)

lemma *diff-assign-lem1*:
dterm-sem I (if i = vid1 then DiffVar vid1 else (Const 0))
(fst ν, vec-lambda (λy. if vid1 = y then Functions I fid1 (vec-lambda
(λi. dterm-sem I (empty i) ν)) else vec-nth (snd ν) y))
=
dterm-sem I (if i = vid1 then $f fid1 empty else (Const 0)) ν

by (*cases i = vid1*) (*auto simp*: *proj-sing1*)

theorem *assign-valid*: *valid assign-axiom*
unfolding *valid-def assign-axiom-def*
by (*simp add*: *assign-lem1*)

theorem *diff-assign-valid*: *valid diff-assign-axiom*
unfolding *valid-def diff-assign-axiom-def*
by (*simp add*: *diff-assign-lem1*)

lemma *mem-to-nonempty*: $\omega \in S \Longrightarrow (S \neq \{\})$
by (*auto*)

lemma *loop-forward*: *ν ∈ fml-sem I ([[$α id1∗∗]]Predicational pid1)*
⟶ ν ∈ fml-sem I (Predicational pid1&&[[$α id1]][[$α id1∗∗]]Predicational pid1)
by (*cases ν*) (*auto intro*: *converse-rtrancl-into-rtrancl simp add*: *box-sem*)

lemma *loop-backward*:
ν ∈ fml-sem I (Predicational pid1 && [[$α id1]][[$α id1∗∗]]Predicational pid1)
⟶ ν ∈ fml-sem I ([[$α id1∗∗]]Predicational pid1)
by (*auto elim*: *converse-rtranclE simp add*: *box-sem*)

theorem *loop-valid*: *valid loop-iterate-axiom*
apply(*simp only*: *valid-def loop-iterate-axiom-def*)
apply(*simp only*: *iff-sem*)
apply(*simp only*: *HOL.iff-conv-conj-imp*)
apply(*rule allI | rule impI*)+
apply(*rule conjI*)
apply(*rule loop-forward*)
apply(*rule loop-backward*)
done

theorem *box-valid*: *valid box-axiom*

unfolding *valid-def box-axiom-def* **by**(*auto*)

theorem *choice-valid*: *valid choice-axiom*
unfolding *valid-def choice-axiom-def* **by** (*auto*)

theorem *compose-valid*: *valid compose-axiom*
unfolding *valid-def compose-axiom-def* **by** (*auto*)

theorem *K-valid*: *valid Kaxiom*
unfolding *valid-def Kaxiom-def* **by** (*auto*)

lemma *I-axiom-lemma*:
fixes *I*::('*sf*,'*sc*,'*sz*) *interp* **and** ν
assumes *is-interp I*
assumes *IS*:$\nu \in$ *fml-sem I* ([[$\$\alpha$ *vid1*∗∗]](*Predicational pid1* →
[[$\$\alpha$ *vid1*]]*Predicational pid1*))
assumes *BC*:$\nu \in$ *fml-sem I* (*Predicational pid1*)
shows $\nu \in$ *fml-sem I* ([[$\$\alpha$ *vid1*∗∗]](*Predicational pid1*))
proof −
have *IS'*:$\bigwedge \nu 2.\ (\nu, \nu 2) \in (\textit{prog-sem I}\ (\$\alpha\ \textit{vid1}))^* \Longrightarrow \nu 2 \in \textit{fml-sem I}\ (\textit{Predicational pid1} \rightarrow [[\$\alpha\ \textit{vid1}]](\textit{Predicational pid1}))$
using *IS* **by** (*auto simp add*: *box-sem*)
have *res*:$\bigwedge \nu 3.\ ((\nu, \nu 3) \in (\textit{prog-sem I}\ (\$\alpha\ \textit{vid1}))^*) \Longrightarrow \nu 3 \in \textit{fml-sem I}\ (\textit{Predicational pid1})$
proof −
fix $\nu 3$
show $((\nu, \nu 3) \in (\textit{prog-sem I}\ (\$\alpha\ \textit{vid1}))^*) \Longrightarrow \nu 3 \in \textit{fml-sem I}\ (\textit{Predicational pid1})$
apply(*induction rule*:*rtrancl-induct*)
apply(*rule BC*)
proof −
fix *y z*
assume *vy*:$(\nu,\ y) \in (\textit{prog-sem I}\ (\$\alpha\ \textit{vid1}))^*$
assume *yz*:$(y,\ z) \in \textit{prog-sem I}\ (\$\alpha\ \textit{vid1})$
assume *yPP*:$y \in \textit{fml-sem I}\ (\textit{Predicational pid1})$
have *imp3*:$y \in \textit{fml-sem I}\ (\textit{Predicational pid1} \rightarrow [[\$\alpha\ \textit{vid1}\ ]](\textit{Predicational pid1}))$
using *IS' vy* **by** (*simp*)
have *imp4*:$y \in \textit{fml-sem I}\ (\textit{Predicational pid1}) \Longrightarrow y \in \textit{fml-sem I}\ ([[\$\alpha\ \textit{vid1}]](\textit{Predicational pid1}))$
using *imp3 impl-sem* **by** (*auto*)
have *yaPP*:$y \in$ *fml-sem I* ([[$\$\alpha$ *vid1*]]*Predicational pid1*) **using** *imp4 yPP* **by** *auto*
have *zPP*:$z \in$ *fml-sem I* (*Predicational pid1*) **using** *yaPP box-sem yz mem-Collect-eq* **by** *blast*
show
$(\nu,\ y) \in (\textit{prog-sem I}\ (\$\alpha\ \textit{vid1}))^* \Longrightarrow$
$(y,\ z) \in \textit{prog-sem I}\ (\$\alpha\ \textit{vid1}) \Longrightarrow$
$y \in \textit{fml-sem I}\ (\textit{Predicational pid1}) \Longrightarrow$

```
      z ∈ fml-sem I (Predicational pid1) using zPP by simp
    qed
  qed
  show ν ∈ fml-sem I ([[$α vid1∗∗]]Predicational pid1)
    using res by (simp add: mem-Collect-eq box-sem loop-sem)
qed

theorem I-valid: valid Iaxiom
  apply(unfold Iaxiom-def valid-def)
  apply(rule impI | rule allI)+
  apply(simp only: impl-sem)
  using I-axiom-lemma by blast

theorem V-valid: valid Vaxiom
  apply(simp only: valid-def Vaxiom-def impl-sem box-sem)
  apply(rule allI | rule impI)+
  apply(auto simp add: empty-def)
done

definition G-holds :: ('sf, 'sc, 'sz) formula ⇒ ('sf, 'sc, 'sz) hp ⇒ bool
where G-holds φ α ≡ valid φ ⟶ valid ([[α]]φ)

definition Skolem-holds :: ('sf, 'sc, 'sz) formula ⇒ 'sz ⇒ bool
where Skolem-holds φ var ≡ valid φ ⟶ valid (Forall var φ)

definition MP-holds :: ('sf, 'sc, 'sz) formula ⇒ ('sf, 'sc, 'sz) formula ⇒ bool
where MP-holds φ ψ ≡ valid (φ → ψ) ⟶ valid φ ⟶ valid ψ

definition CT-holds :: 'sf ⇒ ('sf, 'sz) trm ⇒ ('sf, 'sz) trm ⇒ bool
where CT-holds g ϑ ϑ' ≡ valid (Equals ϑ ϑ')
  ⟶ valid (Equals (Function g (singleton ϑ)) (Function g (singleton ϑ')))

definition CQ-holds :: 'sz ⇒ ('sf, 'sz) trm ⇒ ('sf, 'sz) trm ⇒ bool
where CQ-holds p ϑ ϑ' ≡ valid (Equals ϑ ϑ')
  ⟶ valid ((Prop p (singleton ϑ)) ↔ (Prop p (singleton ϑ')))

definition CE-holds :: 'sc ⇒ ('sf, 'sc, 'sz) formula ⇒ ('sf, 'sc, 'sz) formula ⇒
bool
where CE-holds var φ ψ ≡ valid (φ ↔ ψ)
  ⟶ valid (InContext var φ ↔ InContext var ψ)
```

4.2 Soundness proofs for rules

```
theorem G-sound: G-holds φ α
  by (simp add: G-holds-def valid-def box-sem)

theorem Skolem-sound: Skolem-holds φ var
  by (simp add: Skolem-holds-def valid-def)
```

theorem *MP-sound*: *MP-holds* φ ψ
by (*auto simp add*: *MP-holds-def valid-def*)

lemma *CT-lemma*:$\bigwedge I$::(*'sf*::*finite*, *'sc*::*finite*, *'sz*::{*finite*,*linorder*}) *interp*. $\bigwedge$ *a*::(*real*, *'sz*) *vec*. $\bigwedge$ *b*::(*real*, *'sz*) *vec*. $\forall$ *I*::(*'sf*,*'sc*,*'sz*) *interp*. *is-interp I* $\longrightarrow$ ($\forall$ *a b*. *dterm-sem I* ϑ (*a*, *b*) = *dterm-sem I* ϑ' (*a*, *b*)) $\Longrightarrow$
is-interp I $\Longrightarrow$
Functions I var (*vec-lambda* (λ*i*. *dterm-sem I* (*if i* = *vid1 then* ϑ *else* (*Const 0*)) (*a*, *b*))) =
Functions I var (*vec-lambda* (λ*i*. *dterm-sem I* (*if i* = *vid1 then* ϑ' *else* (*Const 0*)) (*a*, *b*)))
proof −
fix *I* :: (*'sf*::*finite*, *'sc*::*finite*, *'sz*::{*finite*,*linorder*}) *interp* **and** *a* :: (*real*, *'sz*) *vec* **and** *b* :: (*real*, *'sz*) *vec*
assume *a1*: *is-interp I*
assume $\forall$ *I*::(*'sf*,*'sc*,*'sz*) *interp*. *is-interp I* $\longrightarrow$ ($\forall$ *a b*. *dterm-sem I* ϑ (*a*, *b*) = *dterm-sem I* ϑ' (*a*, *b*))
then have $\forall$ *i*. *dterm-sem I* (*if i* = *vid1 then* ϑ' *else* (*Const 0*)) (*a*, *b*) = *dterm-sem I* (*if i* = *vid1 then* ϑ *else* (*Const 0*)) (*a*, *b*)
using *a1* **by** *presburger*
then show *Functions I var* (*vec-lambda* (λ*i*. *dterm-sem I* (*if i* = *vid1 then* ϑ *else* (*Const 0*)) (*a*, *b*)))
= *Functions I var* (*vec-lambda* (λ*i*. *dterm-sem I* (*if i* = *vid1 then* ϑ' *else* (*Const 0*)) (*a*, *b*)))
by *presburger*
qed

theorem *CT-sound*: *CT-holds var* ϑ ϑ'
apply(*simp only*: *CT-holds-def valid-def equals-sem vec-extensionality vec-eq-iff*)
apply(*simp*)
apply(*rule allI* | *rule impI*)+
apply(*simp add*: *CT-lemma*)
done

theorem *CQ-sound*: *CQ-holds var* ϑ ϑ'
proof (*auto simp only*: *CQ-holds-def valid-def equals-sem vec-extensionality vec-eq-iff singleton.simps mem-Collect-eq*)
fix *I* :: (*'sf*,*'sc*,*'sz*) *interp* **and** *a b*
assume *sem*:$\forall$ *I*::(*'sf*,*'sc*,*'sz*) *interp*. $\forall$ ν. *is-interp I* $\longrightarrow$ *dterm-sem I* ϑ ν = *dterm-sem I* ϑ' ν
assume *good*:*is-interp I*
have *sem-eq*:*dterm-sem I* ϑ (*a*,*b*) = *dterm-sem I* ϑ' (*a*,*b*)
using *sem good* **by** *auto*
have *feq*:(χ *i*. *dterm-sem I* (*if i* = *vid1 then* ϑ *else Const 0*) (*a*, *b*)) = (χ *i*. *dterm-sem I* (*if i* = *vid1 then* ϑ' *else Const 0*) (*a*, *b*))
apply(*rule vec-extensionality*)
using *sem-eq* **by** *auto*
then show (*a*, *b*) ∈ *fml-sem I* (\$$\varphi$ *var* (*singleton* ϑ) ↔ \$$\varphi$ *var* (*singleton* ϑ'))
by *auto*

qed

theorem *CE-sound*: *CE-holds var* φ ψ
apply(*simp only*: *CE-holds-def valid-def iff-sem*)
apply(*rule allI* | *rule impI*)+
apply(*simp*)
apply(*metis subsetI subset-antisym surj-pair*)
done
end end
theory *Frechet-Correctness*
imports
Ordinary-Differential-Equations.ODE-Analysis
Lib
Syntax
Denotational-Semantics
Ids
begin
context *ids* **begin**

5 Characterization of Term Derivatives

This section builds up to a proof that in well-formed interpretations, all terms have derivatives, and those derivatives agree with the expected rules of derivatives. In particular, we show the [frechet] function given in the denotational semantics is the true Frechet derivative of a term. From this theorem we can recover all the standard derivative identities as corollaries.

lemma *inner-prod-eq*:
fixes *i*::*'a*::*finite*
shows (λ(*v*::*'a Rvec*). *v* $\cdot$ *axis i 1*) = (λ(*v*::*'a Rvec*). *v* $ *i*)
unfolding *cart-eq-inner-axis axis-def* **by** (*simp add*: *eq-commute*)

theorem *svar-deriv*:
fixes *x*:: *'sv*::*finite* **and** ν:: *'sv Rvec* **and** *F*::*real filter*
shows ((λ*v*. *v* $ *x*) *has-derivative* (λ*v'*. *v'* $\cdot$ (χ *i*. *if i* = *x then 1 else 0*))) (*at* ν)
proof −
let *?f* = (λ*v*. *v*)
let *?f'* = (λ*v'*. *v'*)
let *?g* = (λ*v*. *axis x 1*)
let *?g'* = (λ*v*. *0*)
have *id-deriv*: (*?f has-derivative ?f'*) (*at* ν)
by (*rule has-derivative-ident*)
have *const-deriv*: (*?g has-derivative ?g'*) (*at* ν)
by (*rule has-derivative-const*)
have *inner-deriv*:((λ*x*. *inner* (*?f x*) (*?g x*)) *has-derivative*
(λ*h*. *inner* (*?f* ν) (*?g' h*) + *inner* (*?f' h*) (*?g* ν))) (*at* ν)
by (*intro has-derivative-inner* [*OF id-deriv const-deriv*])
from *inner-prod-eq*
have *left-eq*: (λ*x*. *inner* (*?f x*) (*?g x*)) = (λ*v*. *vec-nth v x*)

```
    by (auto)
  from inner-deriv and inner-prod-eq
  have better-deriv:((λv. vec-nth v x) has-derivative
                     (λh. inner (?f ν) (?g' h) + inner (?f' h) (?g ν))) (at ν)
    by (metis (no-types, lifting) UNIV-I has-derivative-transform)
  have vec-eq:(χ i. if i = x then 1 else 0) = (χ i. if x = i then 1 else 0)
    by(rule vec-extensionality, auto)
  have deriv-eq:(λh. ν · 0 + h · axis x 1) = (λv'. v' · (χ i. if i = x then 1 else 0))
    by(rule ext, auto simp add: axis-def vec-eq)
  show ?thesis
    apply(rule has-derivative-eq-rhs[where f'= (λh. ν · 0 + h · axis x 1)])
    using better-deriv deriv-eq  by auto
qed

lemma function-case-inner:
  assumes good-interp:
    (∀ x i. (Functions I i has-derivative FunctionFrechet I i x) (at x))
  assumes IH:((λv. χ i. sterm-sem I (args i) v)
           has-derivative (λ v. (χ i. frechet I (args i) ν v))) (at ν)
  shows  ((λv. Functions I f (χ i. sterm-sem I (args i) v))
          has-derivative (λv. frechet I ($f f args) ν v)) (at ν)
proof −
  let ?h = (λv. Functions I f (χ i. sterm-sem I (args i) v))
  let ?h' = frechet I ($f f args) ν
  let ?g = (λv. χ i. sterm-sem I (args i) v)
  let ?g' = (λv. χ i. frechet I (args i) ν v)
  let ?f = (λy. Functions I f y)
  let ?f' = FunctionFrechet I f (?g ν)
  have hEqFG:  ?h  = ?f  o ?g by (auto)
  have hEqFG': ?h' = ?f' o ?g'
  proof −
    have frechet-def:frechet I (Function f args) ν
       = (λv'. FunctionFrechet I f (?g ν) (χ i. frechet I (args i) ν v'))
      by (auto)
    have composition:
      (λv'. FunctionFrechet I f (?g ν) (χ i. frechet I (args i) ν v'))
       = (FunctionFrechet I f (?g ν)) o (λ v'. χ i. frechet I (args i) ν v')
      by (auto)
    from frechet-def and composition show ?thesis by (auto)
  qed
  have fDeriv: (?f has-derivative ?f') (at (?g ν))
    using good-interp is-interp-def by blast
  from IH have gDeriv: (?g has-derivative ?g') (at ν) by (auto)
  from fDeriv and gDeriv
  have composeDeriv: ((?f o ?g) has-derivative (?f' o ?g')) (at ν)
    using diff-chain-at good-interp by blast
  from hEqFG hEqFG' composeDeriv show ?thesis by (auto)
qed
```

lemma *func-lemma2*:(∀ *x i*. (*Functions I i has-derivative* (*THE f'*. ∀ *x*. (*Functions I i has-derivative f' x*) (*at x*)) *x*) (*at x*) ∧
continuous-on UNIV (λ*x*. *Blinfun* ((*THE f'*. ∀ *x*. (*Functions I i has-derivative f' x*) (*at x*)) *x*))) ⟹
(⋀ϑ. ϑ ∈ *range args* ⟹ (*sterm-sem I* ϑ *has-derivative frechet I* ϑ ν) (*at* ν)) ⟹
((λ*v*. *Functions I f* (*vec-lambda*(λ*i*. *sterm-sem I* (*args i*) *v*))) *has-derivative* (λ*v'*. (*THE f'*. ∀ *x*. (*Functions I f has-derivative f' x*) (*at x*)) (χ *i*. *sterm-sem I* (*args i*) ν) (χ *i*. *frechet I* (*args i*) ν *v'*))) (*at* ν)
proof −
assume *a1*: ∀ *x i*. (*Functions I i has-derivative* (*THE f'*. ∀ *x*. (*Functions I i has-derivative f' x*) (*at x*)) *x*) (*at x*) ∧
continuous-on UNIV (λ*x*. *Blinfun* ((*THE f'*. ∀ *x*. (*Functions I i has-derivative f' x*) (*at x*)) *x*))
then have *a1'*:∀ *x i*. (*Functions I i has-derivative* (*THE f'*. ∀ *x*. (*Functions I i has-derivative f' x*) (*at x*)) *x*) (*at x*) **by** *auto*
assume *a2*: ⋀ϑ. ϑ ∈ *range args* ⟹ (*sterm-sem I* ϑ *has-derivative frechet I* ϑ ν) (*at* ν)
have ∀ *f fa v*. (∃ *fb*. ¬ (*f* (*fb*::'*a*) *has-derivative fa fb* (*v*::(*real*, '*a*) *vec*)) (*at v*)) ∨ ((λ*v*. (χ *fa*. (*f fa v*::*real*))) *has-derivative* (λ*va*. (χ *f*. *fa f v va*))) (*at v*)
using *has-derivative-vec* **by** *force*
then have ((λ*v*. χ *f*. *sterm-sem I* (*args f*) *v*) *has-derivative* (λ*v*. χ *f*. *frechet I* (*args f*) ν *v*)) (*at* ν)
by (*auto simp add*: *a2 has-derivative-vec*)
then show ((λ*v*. *Functions I f* (*vec-lambda*(λ*f*. *sterm-sem I* (*args f*) *v*))) *has-derivative* (λ*v'*. (*THE f'*. ∀ *x*. (*Functions I f has-derivative f' x*) (*at x*)) (χ *i*. *sterm-sem I* (*args i*) ν) (χ *i*. *frechet I* (*args i*) ν *v'*))) (*at* ν)
using *a1' function-case-inner* **by** *auto*
qed

lemma *func-lemma*:
is-interp I ⟹
(⋀ϑ :: ('*a*::*finite*, '*c*::*finite*) *trm*. ϑ ∈ *range args* ⟹ (*sterm-sem I* ϑ *has-derivative frechet I* ϑ ν) (*at* ν)) ⟹
(*sterm-sem I* (*$f f args*) *has-derivative frechet I* (*$f f args*) ν) (*at* ν)
apply(*auto simp add*: *sfunction-case is-interp-def function-case-inner*)
apply(*erule func-lemma2*)
apply(*auto*)
done

The syntactic definition of term derivatives agrees with the semantic definition. Since the syntactic definition of derivative is total, this gives us that derivatives are "decidable" for terms (modulo computations on reals) and that they obey all the expected identities, which gives us the axioms we want for differential terms essentially for free.

lemma *frechet-correctness*:
fixes *I* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* **and** ν
assumes *good-interp*: *is-interp I*
shows *dfree* ϑ ⟹ *FDERIV* (*sterm-sem I* ϑ) ν :> (*frechet I* ϑ ν)

proof (*induct rule*: *dfree.induct*)
case (*dfree-Var i*) **then show** *?case*
by (*auto simp add*: *svar-case svar-deriv axis-def*)
next
case (*dfree-Fun args i*) **with** *good-interp* **show** *?case*
by (*intro func-lemma*) *auto*
qed *auto*

If terms are semantically equivalent in all states, so are their derivatives

lemma *sterm-determines-frechet*:
fixes *I* ::('*a1*::*finite*, '*b1*::*finite*, '*c*::*finite*) *interp*
and *J* ::('*a2*::*finite*, '*b2*::*finite*, '*c*::*finite*) *interp*
and *ϑ1* :: ('*a1*::*finite*, '*c*::*finite*) *trm*
and *ϑ2* :: ('*a2*::*finite*, '*c*::*finite*) *trm*
and *ν*
assumes *good-interp1*:*is-interp I*
assumes *good-interp2*:*is-interp J*
assumes *free1*:*dfree ϑ1*
assumes *free2*:*dfree ϑ2*
assumes *sem*:*sterm-sem I ϑ1* = *sterm-sem J ϑ2*
shows *frechet I ϑ1* (*fst ν*) (*snd ν*) = *frechet J ϑ2* (*fst ν*) (*snd ν*)
proof −
have *d1*:(*sterm-sem I ϑ1 has-derivative* (*frechet I ϑ1* (*fst ν*))) (*at* (*fst ν*))
using *frechet-correctness*[*OF good-interp1 free1*] **by** *auto*
have *d2*:(*sterm-sem J ϑ2 has-derivative* (*frechet J ϑ2* (*fst ν*))) (*at* (*fst ν*))
using *frechet-correctness*[*OF good-interp2 free2*] **by** *auto*
then have *d1'*:(*sterm-sem I ϑ1 has-derivative* (*frechet J ϑ2* (*fst ν*))) (*at* (*fst ν*))
using *sem* **by** *auto*
thus *?thesis* **using** *has-derivative-unique d1 d1'* **by** *metis*
qed

lemma *the-deriv*:
assumes *deriv*:(*f has-derivative F*) (*at x*)
shows (*THE G*. (*f has-derivative G*) (*at x*)) = *F*
apply(*rule the-equality*)
subgoal by (*rule deriv*)
subgoal for *G* **by** (*auto simp add*: *deriv has-derivative-unique*)
done

lemma *the-all-deriv*:
assumes *deriv*:∀ *x*. (*f has-derivative F x*) (*at x*)
shows (*THE G*. ∀ *x*. (*f has-derivative G x*) (*at x*)) = *F*
apply(*rule the-equality*)
subgoal by (*rule deriv*)
subgoal for *G*
apply(*rule ext*)
subgoal for *x*
apply(*erule allE*[**where** *x*=*x*])
by (*auto simp add*: *deriv has-derivative-unique*)

done
done

typedef ('a, 'c) *strm* = {ϑ:: ('a,'c) *trm*. *dfree* ϑ}
morphisms *raw-term simple-term*
by(*rule exI*[**where** *x*= *Const 0*], *auto simp add*: *dfree-Const*)

typedef ('a, 'b, 'c) *good-interp* = {*I*::('a::*finite*,'b::*finite*,'c::*finite*) *interp*. *is-interp I*}
morphisms *raw-interp good-interp*
apply(*rule exI*[**where** *x*=(| *Functions* = (λ*f x*. *0*), *Predicates* = (λ*p x*. *True*), *Contexts* = (λ*C S*. *S*), *Programs* = (λ*a*. {}), *ODEs* = (λ*c v*. (χ *i*. *0*)), *ODEBV* = λ*c*. {}|)])
apply(*auto simp add*: *is-interp-def*)
proof −
fix *x* ::*real*
have *eq*:(*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) = (λ- -. *0*)
by(*rule the-all-deriv*, *auto*)
have *eq'*:(*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) *x* = (λ-. *0*)
by (*simp add*: *eq*)
have *deriv*:((λ*x*.*0*) *has-derivative* (λ*x*. *0*)) (*at x*)
by *auto*
then show ⋀*x*. ((λ*x*. *0*) *has-derivative* (*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) *x*) (*at x*)
by (*auto simp add*: *eq eq' deriv*)
next
have *eq*:(*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) = (λ- -. *0*)
by(*rule the-all-deriv*, *auto*)
have *eq'*:∀ *x*. (*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) *x* = (λ-. *0*)
by (*simp add*: *eq*)
have *deriv*:⋀*x*. ((λ*x*.*0*) *has-derivative* (λ*x*. *0*)) (*at x*)
by *auto*
have *blin*:⋀*x*. *bounded-linear* ((*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) *x*)
by (*simp add*: *eq'*)
show *continuous-on UNIV* (λ*x*. *Blinfun* ((*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) *x*))
apply(*clarsimp simp add*: *continuous-on-topological*[*of UNIV* (λ*x*. *Blinfun* ((*THE f'*. ∀ *x*. ((λ*x*. *0*) *has-derivative f'* *x*) (*at x*)) *x*))])
apply(*rule exI*[**where** *x* = *UNIV*])
by(*auto simp add*: *eq' blin*)
qed

lemma *frechet-linear*:
assumes *good-interp*:*is-interp I*
fixes *v* ϑ
shows *dfree* ϑ ⟹ *bounded-linear* (*frechet I* ϑ *v*)
proof(*induction rule*: *dfree.induct*)
case (*dfree-Var i*)

then show *?case* **by**(*auto*)
next
case (*dfree-Const r*)
then show *?case* **by** *auto*
next
case (*dfree-Fun args i*)
have *blin1*:⋀*x*. *bounded-linear*(*FunctionFrechet I i x*)
using *good-interp* **unfolding** *is-interp-def* **using** *has-derivative-bounded-linear*
by *blast*
have *blin2*:*bounded-linear* (λ *a*. (χ *i*. *frechet I* (*args i*) *v a*))
using *dfree-Fun.IH* **by**(*rule bounded-linear-vec*)
then show *?case*
using *bounded-linear-compose*[*of FunctionFrechet I i* (χ *i*. *sterm-sem I* (*args i*)
v) (λ*a*. (χ *i*. *frechet I* (*args i*) *v a*)), *OF blin1 blin2*]
by *auto*
next
case (*dfree-Plus* ϑ_1 ϑ_2)
then show *?case*
apply *auto*
using *bounded-linear-add* **by** (*blast*)
next
case (*dfree-Times* ϑ_1 ϑ_2)
then show *?case*
by (*auto simp add*: *bounded-linear-add bounded-linear-const-mult bounded-linear-mult-const*)
qed

setup-lifting *type-definition-good-interp*

setup-lifting *type-definition-strm*

lift-definition *blin-frechet*::('*sf*, '*sc*, '*sz*) *good-interp* ⇒ ('*sf*,'*sz*) *strm* ⇒ (*real*, '*sz*) *vec* ⇒ (*real*, '*sz*) *vec* ⇒$_L$ *real* **is** *frechet*
using *frechet-linear* **by** *auto*

lemmas [*simp*] = *blin-frechet.rep-eq*

lemma *frechet-blin*:*is-interp I* ⟹ *dfree* ϑ ⟹ (λ*v*. *Blinfun* (λ*v'*. *frechet I* ϑ *v v'*)) = *blin-frechet* (*good-interp I*) (*simple-term* ϑ)
apply(*rule ext*)
apply(*rule blinfun-eqI*)
by (*simp add*: *bounded-linear-Blinfun-apply frechet-linear good-interp-inverse simple-term-inverse*)

lemma *sterm-continuous*:
assumes *good-interp*:*is-interp I*
shows *dfree* ϑ ⟹ *continuous-on UNIV* (*sterm-sem I* ϑ)
proof(*induction rule*: *dfree.induct*)
case (*dfree-Fun args i*)
assume *IH*:⋀*i*. *continuous-on UNIV* (*sterm-sem I* (*args i*))

have *con1*:*continuous-on UNIV* (*Functions I i*)
using *good-interp* **unfolding** *is-interp-def*
using *continuous-on-eq-continuous-within has-derivative-continuous* **by** *blast*
have *con2*:*continuous-on UNIV* (*λ x.* (*χ i. sterm-sem I* (*args i*) *x*))
apply(*rule continuous-on-vec-lambda*)
using *IH* **by** *auto*
have *con*:*continuous-on UNIV* ((*Functions I i*) ∘ (*λx.* (*χ i. sterm-sem I* (*args i*) *x*)))
apply(*rule continuous-on-compose*)
using *con1 con2* **apply** *auto*
using *continuous-on-subset* **by** *blast*
show *?case*
using *con comp-def* **by**(*simp*)
qed (*auto intro*: *continuous-intros*)

lemma *sterm-continuous′*:
assumes *good-interp*:*is-interp I*
shows *dfree ϑ* ⟹ *continuous-on S* (*sterm-sem I ϑ*)
using *sterm-continuous continuous-on-subset good-interp* **by** *blast*

lemma *frechet-continuous*:
fixes *I* :: (*'sf*, *'sc*, *'sz*) *interp*
assumes *good-interp*:*is-interp I*
shows *dfree ϑ* ⟹ *continuous-on UNIV* (*blin-frechet* (*good-interp I*) (*simple-term ϑ*))
proof (*induction rule*: *dfree.induct*)
case (*dfree-Var i*)
have *free*:*dfree* (*Var i*) **by** (*rule dfree-Var*)
have *bounded-linear*:*bounded-linear* (*λ ν′. ν′ ∙ axis i 1*)
by (*auto simp add*: *bounded-linear-vec-nth*)
have *cont*:*continuous-on UNIV* (*λν. Blinfun*(*λ ν′. ν′ ∙ axis i 1*))
using *continuous-on-const* **by** *blast*
have *eq*:⋀*ν ν′*. (*λν. Blinfun*(*λ ν′. ν′ ∙ axis i 1*)) *ν ν′* = (*blin-frechet* (*good-interp I*) (*simple-term* (*Var i*))) *ν ν′*
unfolding *axis-def* **apply**(*auto*)
by (*metis* (*no-types*) *axis-def blinfun-inner-left.abs-eq blinfun-inner-left.rep-eq dfree-Var-simps frechet.simps*(*1*) *mem-Collect-eq simple-term-inverse*)
have *eq′*:(*λν. Blinfun*(*λ ν′. ν′ ∙ axis i 1*)) = (*blin-frechet* (*good-interp I*) (*simple-term* (*Var i*)))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
using *eq* **by** *auto*
then show *?case* **by** (*metis cont*)
next
case (*dfree-Const r*)
have *free*:*dfree* (*Const r*) **by** (*rule dfree-Const*)
have *bounded-linear*:*bounded-linear* (*λ ν′. 0*) **by** (*simp*)
have *cont*:*continuous-on UNIV* (*λν. Blinfun*(*λ ν′. 0*))
using *continuous-on-const* **by** *blast*

have *eq'*:($\lambda\nu$. *Blinfun*(λ ν'. *0*)) = (*blin-frechet* (*good-interp I*) (*simple-term* (*Const r*)))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
apply *auto*
using *zero-blinfun.abs-eq zero-blinfun.rep-eq free*
by (*metis frechet.simps*(*5*) *mem-Collect-eq simple-term-inverse*)
then show *?case* **by** (*metis cont*)
next
case (*dfree-Fun args f*)
assume *IH*:$\bigwedge$*i*. *continuous-on UNIV* (*blin-frechet* (*good-interp I*) (*simple-term* (*args i*)))
assume *frees*:($\bigwedge$*i*. *dfree* (*args i*))
then have *free*:*dfree* (*$f f args*) **by** (*auto*)
have *great-interp*:$\bigwedge$*f*. *continuous-on UNIV* (λ*x*. *Blinfun* (*FunctionFrechet I f x*)) **using** *good-interp* **unfolding** *is-interp-def* **by** *auto*
have *cont1*:$\bigwedge$*v*. *continuous-on UNIV* (λ*v'*. (χ *i*. *frechet I* (*args i*) *v v'*))
apply(*rule continuous-on-vec-lambda*)
using *IH* **by** (*simp add*: *frechet-linear frees good-interp linear-continuous-on*)
have *eq*:(λ*v*. *Blinfun*(λ*v'*. *FunctionFrechet I f* (χ *i*. *sterm-sem I* (*args i*) *v*) (χ *i*. *frechet I* (*args i*) *v v'*)))
= (*blin-frechet* (*good-interp I*) (*simple-term* (*Function f args*)))
using *frechet-blin*[*OF good-interp free*] **by** *auto*
have *bounded-linears*:$\bigwedge$*x*. *bounded-linear* (*FunctionFrechet I f x*) **using** *good-interp* **unfolding** *is-interp-def* **by** *blast*
let *?blin-ff* =(λ*x*. *Blinfun* (*FunctionFrechet I f x*))
have *some-eq*:(λ*x*. *Blinfun* (*FunctionFrechet I f* (χ *i*. *sterm-sem I* (*args i*) *x*))) =
(*(?blin-ff)* $\circ$ (λ*x*. (χ *i*. *sterm-sem I* (*args i*) *x*)))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
unfolding *comp-def* **by** *blast*
have *sub-cont*:*continuous-on UNIV* ((*?blin-ff*) $\circ$ (λ*x*. (χ *i*. *sterm-sem I* (*args i*) *x*)))
apply(*rule continuous-intros*)+
apply (*simp add*: *frees good-interp sterm-continuous'*)
using *continuous-on-subset great-interp* **by** *blast*
have *blin-frech-vec*:$\bigwedge$*x*. *bounded-linear* (λ*v'*. χ *i*. *frechet I* (*args i*) *x v'*)
by (*simp add*: *bounded-linear-vec frechet-linear frees good-interp*)
have *frech-vec-eq*:(λ*x*. *Blinfun* (λ*v'*. χ *i*. *frechet I* (*args i*) *x v'*)) = (λ*x*. *blinfun-vec* (λ *i*. *blin-frechet* (*good-interp I*) (*simple-term* (*args i*)) *x*))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
apply(*rule vec-extensionality*)
subgoal for *x i j*
using *blin-frech-vec*[*of x*]
apply *auto*
by (*metis* (*no-types*, *lifting*) *blin-frechet.rep-eq bounded-linear-Blinfun-apply frechet-blin frechet-linear frees good-interp vec-lambda-beta*)

done
have *cont-frech-vec*:*continuous-on UNIV* (λ*x*. *blinfun-vec* (λ *i*. *blin-frechet* (*good-interp I*) (*simple-term* (*args i*)) *x*))
apply(*rule continuous-blinfun-vec'*)
using *IH* **by** *blast*
have *cont-intro*:⋀ *s f g*. *continuous-on s f* ⟹ *continuous-on s g* ⟹ *continuous-on s* (λ*x*. *f x* o_L *g x*)
by (*auto intro*: *continuous-intros*)
have *cont*:*continuous-on UNIV* (λ*v*. *blinfun-compose* (*Blinfun* (*FunctionFrechet I f* (χ *i*. *sterm-sem I* (*args i*) *v*))) (*Blinfun*(λ*v'*. (χ *i*. *frechet I* (*args i*) *v v'*))))
apply(*rule cont-intro*)
subgoal using *sub-cont* **by** *simp*
using *frech-vec-eq cont-frech-vec* **by** *presburger*
have *best-eq*:(*blin-frechet* (*good-interp I*) (*simple-term* ($*f f args*))) = (λ*v*. *blinfun-compose* (*Blinfun* (*FunctionFrechet I f* (χ *i*. *sterm-sem I* (*args i*) *v*))) (*Blinfun*(λ*v'*. (χ *i*. *frechet I* (*args i*) *v v'*))))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
proof −
fix *v* :: (*real*, *'sz*) *vec* **and** *i* :: (*real*, *'sz*) *vec*
have *frechet I* ($*f f args*) *v i* = *blinfun-apply* (*blin-frechet* (*good-interp I*) (*simple-term* ($*f f args*)) *v*) *i*
by (*metis* (*no-types*) *bounded-linear-Blinfun-apply dfree-Fun-simps frechet-blin frechet-linear frees good-interp*)
then have *FunctionFrechet I f* (χ *s*. *sterm-sem I* (*args s*) *v*) (*blinfun-apply* (*Blinfun* (λ*va*. χ *s*. *frechet I* (*args s*) *v va*)) *i*) = *blinfun-apply* (*blin-frechet* (*good-interp I*) (*simple-term* ($*f f args*)) *v*) *i*
by (*simp add*: *blin-frech-vec bounded-linear-Blinfun-apply*)
then show *blinfun-apply* (*blin-frechet* (*good-interp I*) (*simple-term* ($*f f args*)) *v*) *i* = *blinfun-apply* (*Blinfun* (*FunctionFrechet I f* (χ *s*. *sterm-sem I* (*args s*) *v*)) o_L *Blinfun* (λ*va*. χ *s*. *frechet I* (*args s*) *v va*)) *i*
by (*metis* (*no-types*) *blinfun-apply-blinfun-compose bounded-linear-Blinfun-apply bounded-linears*)
qed
then show *?case* **using** *cont best-eq* **by** *auto*
next
case (*dfree-Plus* ϑ_1 ϑ_2)
assume *IH1*:*continuous-on UNIV* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_1))
assume *IH2*:*continuous-on UNIV* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_2))
assume *free1*:*dfree* ϑ_1
assume *free2*:*dfree* ϑ_2
have *free*:*dfree* (*Plus* ϑ_1 ϑ_2) **using** *free1 free2* **by** *auto*
have *bounded-linear*:⋀*v*. *bounded-linear* (λ*v'*. *frechet I* ϑ_1 *v v'* + *frechet I* ϑ_2 *v v'*)
subgoal for *v*
using *frechet-linear*[*OF good-interp free*] **by** *auto*
done
have *eq2*:(λ*v*. *blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v* + *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*) = *blin-frechet* (*good-interp I*) (*simple-term*

(*Plus* ϑ_1 ϑ_2))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
by (*simp add*: *blinfun.add-left free1 free2 simple-term-inverse*)
have *cont*:*continuous-on UNIV* (λv. *blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v* + *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*)
using *continuous-on-add dfree-Plus.IH*(*1*) *dfree-Plus.IH*(*2*) **by** *blast*
then show *?case* **using** *cont eq2* **by** *metis*
next
case (*dfree-Times* ϑ_1 ϑ_2)
assume *IH1*:*continuous-on UNIV* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_1))
assume *IH2*:*continuous-on UNIV* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_2))
assume *free1*:*dfree* ϑ_1
assume *free2*:*dfree* ϑ_2
have *free*:*dfree* (*Times* ϑ_1 ϑ_2) **using** *free1 free2* **by** *auto*
have *bounded-linear*:$\bigwedge v$. *bounded-linear* ($\lambda v'$. *sterm-sem I* ϑ_1 *v* ∗ *frechet I* ϑ_2 *v* *v'* + *frechet I* ϑ_1 *v v'* ∗ *sterm-sem I* ϑ_2 *v*)
apply(*rule bounded-linear-add*)
apply(*rule bounded-linear-const-mult*)
using *frechet-linear*[*OF good-interp free2*] **apply** *auto*
apply(*rule bounded-linear-mult-const*)
using *frechet-linear*[*OF good-interp free1*] **by** *auto*
then have *blin'*:$\bigwedge v$. ($\lambda v'$. *sterm-sem I* ϑ_1 *v* ∗ *frechet I* ϑ_2 *v v'* + *frechet I* ϑ_1 *v v'* ∗ *sterm-sem I* ϑ_2 *v*) ∈ {*f*. *bounded-linear f*} **by** *auto*
have *blinfun-eq*:$\bigwedge v$. *Blinfun* ($\lambda v'$. *sterm-sem I* ϑ_1 *v* ∗ *frechet I* ϑ_2 *v v'* + *frechet I* ϑ_1 *v v'* ∗ *sterm-sem I* ϑ_2 *v*)
= *scaleR* (*sterm-sem I* ϑ_1 *v*) (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*) + *scaleR* (*sterm-sem I* ϑ_2 *v*) (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*)
apply(*rule blinfun-eqI*)
subgoal for *v i*
using *Blinfun-inverse*[*OF blin'*, *of v*] **apply** *auto*
using *blinfun.add-left*[*of sterm-sem I* ϑ_1 *v* ∗$_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v sterm-sem I* ϑ_2 *v* ∗$_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*]
blinfun.scaleR-left[*of sterm-sem I* ϑ_1 *v blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*]
blinfun.scaleR-left[*of sterm-sem I* ϑ_2 *v blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*]
bounded-linear-Blinfun-apply
frechet-blin[*OF good-interp free1*]
frechet-blin[*OF good-interp free2*]
frechet-linear[*OF good-interp free1*]
frechet-linear[*OF good-interp free2*]
mult.commute
real-scaleR-def
proof −
have *f1*: $\bigwedge v$. *blinfun-apply* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*) = *frechet I* ϑ_1 *v*
by (*metis* (*no-types*) ‹(λv. *Blinfun* (*frechet I* ϑ_1 *v*)) = *blin-frechet* (*good-interp*

I) (*simple-term* ϑ_1)› ‹$\bigwedge$*v*. *bounded-linear* (*frechet I* ϑ_1 *v*)› *bounded-linear-Blinfun-apply*)
have $\bigwedge$*v*. *blinfun-apply* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*) = *frechet I* ϑ_2 *v*
by (*metis* (*no-types*) ‹(λ*v*. *Blinfun* (*frechet I* ϑ_2 *v*)) = *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2)› ‹$\bigwedge$*v*. *bounded-linear* (*frechet I* ϑ_2 *v*)› *bounded-linear-Blinfun-apply*)
then show *sterm-sem I* ϑ_1 *v* ∗ *frechet I* ϑ_2 *v i* + *frechet I* ϑ_1 *v i* ∗ *sterm-sem I* ϑ_2 *v* = *blinfun-apply* (*sterm-sem I* ϑ_1 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v* + *sterm-sem I* ϑ_2 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*) *i*
using *f1* **by** (*simp add*: ‹$\bigwedge$*b*. *blinfun-apply* (*sterm-sem I* ϑ_1 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v* + *sterm-sem I* ϑ_2 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*) *b* = *blinfun-apply* (*sterm-sem I* ϑ_1 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*) *b* + *blinfun-apply* (*sterm-sem I* ϑ_2 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*) *b*› ‹$\bigwedge$*b*. *blinfun-apply* (*sterm-sem I* ϑ_1 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*) *b* = *sterm-sem I* ϑ_1 *v* $*_R$ *blinfun-apply* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*) *b*› ‹$\bigwedge$*b*. *blinfun-apply* (*sterm-sem I* ϑ_2 *v* $*_R$ *blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*) *b* = *sterm-sem I* ϑ_2 *v* $*_R$ *blinfun-apply* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*) *b*›)
qed
done
have *cont'*:*continuous-on UNIV*
(λ*v*. *scaleR* (*sterm-sem I* ϑ_1 *v*) (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_2) *v*)
+ *scaleR* (*sterm-sem I* ϑ_2 *v*) (*blin-frechet* (*good-interp I*) (*simple-term* ϑ_1) *v*))
apply(*rule continuous-on-add*)
apply(*rule continuous-on-scaleR*)
apply(*rule sterm-continuous*[*OF good-interp free1*])
apply(*rule IH2*)
apply(*rule continuous-on-scaleR*)
apply(*rule sterm-continuous*[*OF good-interp free2*])
by(*rule IH1*)
have *cont*:*continuous-on UNIV* (λ*v*. *Blinfun* (λ*v'*. *sterm-sem I* ϑ_1 *v* ∗ *frechet I* ϑ_2 *v v'* + *frechet I* ϑ_1 *v v'* ∗ *sterm-sem I* ϑ_2 *v*))
using *cont' blinfun-eq* **by** *auto*
have *eq*:(λ*v*. *Blinfun* (λ*v'*. *sterm-sem I* ϑ_1 *v* ∗ *frechet I* ϑ_2 *v v'* + *frechet I* ϑ_1 *v v'* ∗ *sterm-sem I* ϑ_2 *v*)) = *blin-frechet* (*good-interp I*) (*simple-term* (*Times* ϑ_1 ϑ_2))
using *frechet-blin*[*OF good-interp free*]
by *auto*
then show *?case* **by** (*metis cont*)
qed
end end
theory *Static-Semantics*
imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics

Frechet-Correctness
begin

6 Static Semantics

This section introduces functions for computing properties of the static semantics, specifically the following dependencies:

- Signatures: Symbols (from the interpretation) which influence the result of a term, ode, formula, program
- Free variables: Variables (from the state) which influence the result of a term, ode, formula, program
- Bound variables: Variables (from the state) that *might* be influenced by a program
- Must-bound variables: Variables (from the state) that are *always* influenced by a program (i.e. will never depend on anything other than the free variables of that program)

We also prove basic lemmas about these definitions, but their overall correctness is proved elsewhere in the Bound Effect and Coincidence theorems.

6.1 Signature Definitions

primrec $SIGT :: ('a, 'c)\ trm \Rightarrow 'a\ set$
where
$SIGT\ (Var\ var) = \{\}$
$| SIGT\ (Const\ r) = \{\}$
$| SIGT\ (Function\ var\ f) = \{var\} \cup (\bigcup i.\ SIGT\ (f\ i))$
$| SIGT\ (Plus\ t1\ t2) = SIGT\ t1 \cup SIGT\ t2$
$| SIGT\ (Times\ t1\ t2) = SIGT\ t1 \cup SIGT\ t2$
$| SIGT\ (DiffVar\ x) = \{\}$
$| SIGT\ (Differential\ t) = SIGT\ t$

primrec $SIGO :: ('a, 'c)\ ODE \Rightarrow ('a + 'c)\ set$
where
$SIGO\ (OVar\ c) = \{Inr\ c\}$
$| SIGO\ (OSing\ x\ \vartheta) = \{Inl\ x | x.\ x \in SIGT\ \vartheta\}$
$| SIGO\ (OProd\ ODE1\ ODE2) = SIGO\ ODE1 \cup SIGO\ ODE2$

primrec $SIGP :: ('a, 'b, 'c)\ hp \Rightarrow ('a + 'b + 'c)\ set$
and $SIGF :: ('a, 'b, 'c)\ formula \Rightarrow ('a + 'b + 'c)\ set$
where
$SIGP\ (Pvar\ var) = \{Inr\ (Inr\ var)\}$
$| SIGP\ (Assign\ var\ t) = \{Inl\ x \mid x.\ x \in SIGT\ t\}$
$| SIGP\ (DiffAssign\ var\ t) = \{Inl\ x \mid x.\ x \in SIGT\ t\}$

| *SIGP* (*Test p*) = *SIGF p*
| *SIGP* (*EvolveODE ODE p*) = *SIGF p* ∪ {*Inl x* | *x*. *Inl x* ∈ *SIGO ODE*} ∪ {*Inr* (*Inr x*) | *x*. *Inr x* ∈ *SIGO ODE*}
| *SIGP* (*Choice a b*) = *SIGP a* ∪ *SIGP b*
| *SIGP* (*Sequence a b*) = *SIGP a* ∪ *SIGP b*
| *SIGP* (*Loop a*) = *SIGP a*
| *SIGF* (*Geq t1 t2*) = {*Inl x* | *x*. *x* ∈ *SIGT t1* ∪ *SIGT t2*}
| *SIGF* (*Prop var args*) = {*Inr* (*Inr var*)} ∪ {*Inl x* | *x*. *x* ∈ (⋃ *i*. *SIGT* (*args i*))}
| *SIGF* (*Not p*) = *SIGF p*
| *SIGF* (*And p1 p2*) = *SIGF p1* ∪ *SIGF p2*
| *SIGF* (*Exists var p*) = *SIGF p*
| *SIGF* (*Diamond a p*) = *SIGP a* ∪ *SIGF p*
| *SIGF* (*InContext var p*) = {*Inr* (*Inl var*)} ∪ *SIGF p*

fun *primify* :: (′*a* + ′*a*) ⇒ (′*a* + ′*a*) *set*
where
primify (*Inl x*) = {*Inl x*, *Inr x*}
| *primify* (*Inr x*) = {*Inl x*, *Inr x*}

6.2 Variable Binding Definitions

We represent the (free or bound or must-bound) variables of a term as an (id + id) set, where all the (Inl x) elements are unprimed variables x and all the (Inr x) elements are primed variables x'.

Free variables of a term

primrec *FVT* :: (′*a*, ′*c*) *trm* ⇒ (′*c* + ′*c*) *set*
where
FVT (*Var x*) = {*Inl x*}
| *FVT* (*Const x*) = {}
| *FVT* (*Function f args*) = (⋃ *i*. *FVT* (*args i*))
| *FVT* (*Plus f g*) = *FVT f* ∪ *FVT g*
| *FVT* (*Times f g*) = *FVT f* ∪ *FVT g*
| *FVT* (*Differential f*) = (⋃ *x* ∈ (*FVT f*). *primify x*)
| *FVT* (*DiffVar x*) = {*Inr x*}

fun *FVDiff* :: (′*a*, ′*c*) *trm* ⇒ (′*c* + ′*c*) *set*
where *FVDiff f* = (⋃ *x* ∈ (*FVT f*). *primify x*)

Free variables of an ODE includes both the bound variables and the terms

fun *FVO* :: (′*a*, ′*c*) *ODE* ⇒ ′*c set*
where
FVO (*OVar c*) = *UNIV*
| *FVO* (*OSing x ϑ*) = {*x*} ∪ {*x* . *Inl x* ∈ *FVT ϑ*}
| *FVO* (*OProd ODE1 ODE2*) = *FVO ODE1* ∪ *FVO ODE2*

Bound variables of ODEs, formulas, programs

fun *BVO* :: (′*a*, ′*c*) *ODE* ⇒ (′*c* + ′*c*) *set*

where
  $BVO\ (OVar\ c) = UNIV$
| $BVO\ (OSing\ x\ \vartheta) = \{Inl\ x, Inr\ x\}$
| $BVO\ (OProd\ ODE1\ ODE2) = BVO\ ODE1 \cup BVO\ ODE2$

fun $BVF :: ('a, 'b, 'c)\ formula \Rightarrow ('c + 'c)\ set$
and $BVP :: ('a, 'b, 'c)\ hp \Rightarrow ('c + 'c)\ set$
where
  $BVF\ (Geq\ f\ g) = \{\}$
| $BVF\ (Prop\ p\ dfun\text{-}args) = \{\}$
| $BVF\ (Not\ p) = BVF\ p$
| $BVF\ (And\ p\ q) = BVF\ p \cup BVF\ q$
| $BVF\ (Exists\ x\ p) = \{Inl\ x\} \cup BVF\ p$
| $BVF\ (Diamond\ \alpha\ p) = BVP\ \alpha \cup BVF\ p$
| $BVF\ (InContext\ C\ p) = UNIV$

| $BVP\ (Pvar\ a) = UNIV$
| $BVP\ (Assign\ x\ \vartheta) = \{Inl\ x\}$
| $BVP\ (DiffAssign\ x\ \vartheta) = \{Inr\ x\}$
| $BVP\ (Test\ \varphi) = \{\}$
| $BVP\ (EvolveODE\ ODE\ \varphi) = BVO\ ODE$
| $BVP\ (Choice\ \alpha\ \beta) = BVP\ \alpha \cup BVP\ \beta$
| $BVP\ (Sequence\ \alpha\ \beta) = BVP\ \alpha \cup BVP\ \beta$
| $BVP\ (Loop\ \alpha) = BVP\ \alpha$

Must-bound variables (of a program)

fun $MBV :: ('a, 'b, 'c)\ hp \Rightarrow ('c + 'c)\ set$
where
  $MBV\ (Pvar\ a) = \{\}$
| $MBV\ (Choice\ \alpha\ \beta) = MBV\ \alpha \cap MBV\ \beta$
| $MBV\ (Sequence\ \alpha\ \beta) = MBV\ \alpha \cup MBV\ \beta$
| $MBV\ (Loop\ \alpha) = \{\}$
| $MBV\ (EvolveODE\ ODE\ \text{-}) = (Inl\ `(ODE\text{-}dom\ ODE)) \cup (Inr\ `(ODE\text{-}dom\ ODE))$
| $MBV\ \alpha = BVP\ \alpha$

Free variables of a formula, free variables of a program

fun $FVF :: ('a, 'b, 'c)\ formula \Rightarrow ('c + 'c)\ set$
and $FVP :: ('a, 'b, 'c)\ hp \Rightarrow ('c + 'c)\ set$
where
  $FVF\ (Geq\ f\ g) = FVT\ f \cup FVT\ g$
| $FVF\ (Prop\ p\ args) = (\bigcup i.\ FVT\ (args\ i))$
| $FVF\ (Not\ p) = FVF\ p$
| $FVF\ (And\ p\ q) = FVF\ p \cup FVF\ q$
| $FVF\ (Exists\ x\ p) = FVF\ p - \{Inl\ x\}$
| $FVF\ (Diamond\ \alpha\ p) = \quad FVP\ \alpha \cup (FVF\ p - MBV\ \alpha)$
| $FVF\ (InContext\ C\ p) = UNIV$
| $FVP\ (Pvar\ a) = UNIV$
| $FVP\ (Assign\ x\ \vartheta) = FVT\ \vartheta$
| $FVP\ (DiffAssign\ x\ \vartheta) = FVT\ \vartheta$

| $FVP\ (Test\ \varphi) = FVF\ \varphi$
| $FVP\ (EvolveODE\ ODE\ \varphi) = BVO\ ODE \cup (Inl\ `\ FVO\ ODE) \cup FVF\ \varphi$
| $FVP\ (Choice\ \alpha\ \beta) = FVP\ \alpha \cup FVP\ \beta$
| $FVP\ (Sequence\ \alpha\ \beta) = FVP\ \alpha \cup (FVP\ \beta - MBV\ \alpha)$
| $FVP\ (Loop\ \alpha) = FVP\ \alpha$

6.3 Lemmas for reasoning about static semantics

lemma *primify-contains*:$x \in primify\ x$
by (*cases x*) *auto*

lemma *FVDiff-sub*:$FVT\ f \subseteq FVDiff\ f$
by (*auto simp add*: *primify-contains*)

lemma *fvdiff-plus1*:$FVDiff\ (Plus\ t1\ t2) = FVDiff\ t1 \cup FVDiff\ t2$
by (*auto*)

lemma *agree-func-fvt*:$Vagree\ \nu\ \nu'\ (FVT\ (Function\ f\ args)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVT\ (args\ i))$
by (*auto simp add*: *Set.Un-upper1 agree-supset Vagree-def*)

lemma *agree-plus1*:$Vagree\ \nu\ \nu'\ (FVDiff\ (Plus\ t1\ t2)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ t1)$
proof −
assume *agree*:$Vagree\ \nu\ \nu'\ (FVDiff\ (Plus\ t1\ t2))$
have *agree'*:$Vagree\ \nu\ \nu'\ ((\bigcup i{\in}FVT\ t1.\ primify\ i) \cup (\bigcup i{\in}FVT\ t2.\ primify\ i))$
using *fvdiff-plus1 FVDiff.simps agree* **by** (*auto*)
have *agreeL*:$Vagree\ \nu\ \nu'\ ((\bigcup i{\in}FVT\ t1.\ primify\ i))$
using *agree' agree-supset Set.Un-upper1* **by** (*blast*)
show $Vagree\ \nu\ \nu'\ (FVDiff\ t1)$ **using** *agreeL* **by** (*auto*)
qed

lemma *agree-plus2*:$Vagree\ \nu\ \nu'\ (FVDiff\ (Plus\ t1\ t2)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ t2)$
proof −
assume *agree*:$Vagree\ \nu\ \nu'\ (FVDiff\ (Plus\ t1\ t2))$
have *agree'*:$Vagree\ \nu\ \nu'\ ((\bigcup i{\in}FVT\ t1.\ primify\ i) \cup (\bigcup i{\in}FVT\ t2.\ primify\ i))$
using *fvdiff-plus1 FVDiff.simps agree* **by** (*auto*)
have *agreeR*:$Vagree\ \nu\ \nu'\ ((\bigcup i{\in}FVT\ t2.\ primify\ i))$
using *agree' agree-supset Set.Un-upper1* **by** (*blast*)
show $Vagree\ \nu\ \nu'\ (FVDiff\ t2)$ **using** *agreeR* **by** (*auto*)
qed

lemma *agree-times1*:$Vagree\ \nu\ \nu'\ (FVDiff\ (Times\ t1\ t2)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ t1)$
proof −
assume *agree*:$Vagree\ \nu\ \nu'\ (FVDiff\ (Times\ t1\ t2))$
have *agree'*:$Vagree\ \nu\ \nu'\ ((\bigcup i{\in}FVT\ t1.\ primify\ i) \cup (\bigcup i{\in}FVT\ t2.\ primify\ i))$
using *fvdiff-plus1 FVDiff.simps agree* **by** (*auto*)

have *agreeL*: *Vagree* ν ν' (($\bigcup i{\in}FVT\ t1.\ primify\ i$))
using *agree'* *agree-supset* *Set.Un-upper1* **by** (*blast*)
show *Vagree* ν ν' (*FVDiff t1*) **using** *agreeL* **by** (*auto*)
qed

lemma *agree-times2*: *Vagree* ν ν' (*FVDiff* (*Times t1 t2*)) $\Longrightarrow$ *Vagree* ν ν' (*FVDiff t2*)
proof −
assume *agree*: *Vagree* ν ν' (*FVDiff* (*Times t1 t2*))
have *agree'*: *Vagree* ν ν' (($\bigcup i{\in}FVT\ t1.\ primify\ i$) ∪ ($\bigcup i{\in}FVT\ t2.\ primify\ i$))
using *fvdiff-plus1* *FVDiff.simps* *agree* **by** (*auto*)
have *agreeR*: *Vagree* ν ν' (($\bigcup i{\in}FVT\ t2.\ primify\ i$))
using *agree'* *agree-supset* *Set.Un-upper1* **by** (*blast*)
show *Vagree* ν ν' (*FVDiff t2*) **using** *agreeR* **by** (*auto*)
qed

lemma *agree-func*: *Vagree* ν ν' (*FVDiff* (\$*f var args*)) $\Longrightarrow$ ($\bigwedge i.$ *Vagree* ν ν' (*FVDiff* (*args i*)))
proof −
assume *agree*: *Vagree* ν ν' (*FVDiff* (\$*f var args*))
have *agree'*: *Vagree* ν ν' (($\bigcup i.$ ($\bigcup j \in (FVT\ (args\ i)).\ primify\ j$)))
using *fvdiff-plus1* *FVDiff.simps* *agree* **by** (*auto*)
fix $i :: 'a$
have $\bigwedge S.\ \neg\ S \subseteq (\bigcup f.\ \bigcup\ (primify\ `\ FVT\ (args\ f))) \vee$ *Vagree* ν ν' S
using *agree'* *agree-supset* **by** *blast*
then have $\bigwedge f.\ f \notin UNIV \vee$ *Vagree* ν ν' ($\bigcup\ (primify\ `\ FVT\ (args\ f))$)
by *blast*
then show *Vagree* ν ν' (*FVDiff* (*args i*))
by *simp*
qed

end
theory *Coincidence*
imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
Frechet-Correctness
Static-Semantics
begin

7 Coincidence Theorems and Corollaries

This section proves coincidence: semantics of terms, odes, formulas and programs depend only on the free variables. This is one of the major lemmas for the correctness of uniform substitutions. Along the way, we also prove the

equivalence between two similar, but different semantics for ODE programs: It does not matter whether the semantics of ODE's insist on the existence of a solution that agrees with the start state on all variables vs. one that agrees only on the variables that are actually relevant to the ODE. This is proven here by simultaneous induction with the coincidence theorem for the following reason:

The reason for having two different semantics is that some proofs are easier with one semantics and other proofs are easier with the other definition. The coincidence proof is either with the more complicated definition, which should not be used as the main definition because it would make the specification for the dL semantics significantly larger, effectively increasing the size of the trusted core. However, that the proof of equivalence between the semantics using the coincidence lemma for formulas. In order to use the coincidence proof in the equivalence proof and the equivalence proof in the coincidence proof, they are proved by simultaneous induction.

context *ids* **begin**

7.1 Term Coincidence Theorems

lemma *coincidence-sterm*:*Vagree* ν ν' (*FVT* ϑ) $\Longrightarrow$ *sterm-sem I* ϑ (*fst* ν) = *sterm-sem I* ϑ (*fst* ν')
apply(*induct* ϑ)
apply(*auto simp add*: *Vagree-def*)
by (*meson rangeI*)

lemma *coincidence-sterm'*:*dfree* ϑ $\Longrightarrow$ *Vagree* ν ν' (*FVT* ϑ) $\Longrightarrow$ *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* ϑ} $\Longrightarrow$ *sterm-sem I* ϑ (*fst* ν) = *sterm-sem J* ϑ (*fst* ν')
proof (*induction rule*: *dfree.induct*)
case (*dfree-Fun args i*)
then show *?case*
proof (*auto*)
assume *free*:(⋀*i*. *dfree* (*args i*))
and *IH*:(⋀*i*. *Vagree* ν ν' (*FVT* (*args i*)) $\Longrightarrow$ *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* (*args i*)} $\Longrightarrow$ *sterm-sem I* (*args i*) (*fst* ν) = *sterm-sem J* (*args i*) (*fst* ν'))
and *VA*:*Vagree* ν ν' (⋃*i*. *FVT* (*args i*))
and *IA*:*Iagree I J* {*Inl x* |*x*. *x* = *i* ∨ (∃*xa*. *x* ∈ *SIGT* (*args xa*))}
from *IA* **have** *IAorig*:*Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* (*Function i args*)} **by** *auto*
from *Iagree-Func*[*OF IAorig*] **have** *eqF*:*Functions I i* = *Functions J i* **by** *auto*
have *Vsubs*:⋀*i*. *FVT* (*args i*) ⊆ (⋃*i*. *FVT* (*args i*)) **by** *auto*
from *VA*
have *VAs*:⋀*i*. *Vagree* ν ν' (*FVT* (*args i*))
using *agree-sub*[*OF Vsubs*] **by** *auto*
have *Isubs*:⋀*j*. {*Inl x* |*x*. *x* ∈ *SIGT* (*args j*)} ⊆ {*Inl x* |*x*. *x* ∈ *SIGT* (*Function i args*)}
by *auto*

from *IA*
have *IAs*:$\bigwedge i$. *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* (*args i*)}
using *Iagree-sub*[*OF Isubs*] **by** *auto*
show *Functions I i* (*χ i*. *sterm-sem I* (*args i*) (*fst ν*)) = *Functions J i* (*χ i*. *sterm-sem J* (*args i*) (*fst ν′*))
using *IH*[*OF VAs IAs*] *eqF* **by** *auto*
qed
next
case (*dfree-Plus ϑ*$_1$ *ϑ*$_2$)
then show *?case*
proof (*auto*)
assume *dfree ϑ*$_1$ *dfree ϑ*$_2$
and *IH1*:(*Vagree ν ν′* (*FVT ϑ*$_1$) ⟹ *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_1$} ⟹ *sterm-sem I ϑ*$_1$ (*fst ν*) = *sterm-sem J ϑ*$_1$ (*fst ν′*))
and *IH2*:(*Vagree ν ν′* (*FVT ϑ*$_2$) ⟹ *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_2$} ⟹ *sterm-sem I ϑ*$_2$ (*fst ν*) = *sterm-sem J ϑ*$_2$ (*fst ν′*))
and *VA*:*Vagree ν ν′* (*FVT ϑ*$_1$ ∪ *FVT ϑ*$_2$)
and *IA*:*Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_1$ ∨ *x* ∈ *SIGT ϑ*$_2$}
from *VA*
have *VAs*:*Vagree ν ν′* (*FVT ϑ*$_1$) *Vagree ν ν′* (*FVT ϑ*$_2$)
unfolding *Vagree-def* **by** *auto*
have *Isubs*:{*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_1$} ⊆ {*Inl x* |*x*. *x* ∈ *SIGT* (*Plus ϑ*$_1$ *ϑ*$_2$)}
{*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_2$} ⊆ {*Inl x* |*x*. *x* ∈ *SIGT* (*Plus ϑ*$_1$ *ϑ*$_2$)}
by *auto*
from *IA*
have *IAs*:*Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_1$}
Iagree I J {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_2$}
using *Iagree-sub*[*OF Isubs*(*1*)] *Iagree-sub*[*OF Isubs*(*2*)] **by** *auto*
show *sterm-sem I ϑ*$_1$ (*fst ν*) + *sterm-sem I ϑ*$_2$ (*fst ν*) = *sterm-sem J ϑ*$_1$ (*fst ν′*) + *sterm-sem J ϑ*$_2$ (*fst ν′*)
using *IH1*[*OF VAs*(*1*) *IAs*(*1*)] *IH2*[*OF VAs*(*2*) *IAs*(*2*)] **by** *auto*
qed
next
case (*dfree-Times ϑ*$_1$ *ϑ*$_2$)
then show *?case*
proof (*auto*)
assume *dfree ϑ*$_1$ *dfree ϑ*$_2$
and *IH1*:(*Vagree ν ν′* (*FVT ϑ*$_1$) ⟹ *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_1$} ⟹ *sterm-sem I ϑ*$_1$ (*fst ν*) = *sterm-sem J ϑ*$_1$ (*fst ν′*))
and *IH2*:(*Vagree ν ν′* (*FVT ϑ*$_2$) ⟹ *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_2$} ⟹ *sterm-sem I ϑ*$_2$ (*fst ν*) = *sterm-sem J ϑ*$_2$ (*fst ν′*))
and *VA*:*Vagree ν ν′* (*FVT ϑ*$_1$ ∪ *FVT ϑ*$_2$)
and *IA*:*Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_1$ ∨ *x* ∈ *SIGT ϑ*$_2$}
from *VA*
have *VAs*:*Vagree ν ν′* (*FVT ϑ*$_1$) *Vagree ν ν′* (*FVT ϑ*$_2$)
unfolding *Vagree-def* **by** *auto*
have *Isubs*:{*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_1$} ⊆ {*Inl x* |*x*. *x* ∈ *SIGT* (*Times ϑ*$_1$ *ϑ*$_2$)}
{*Inl x* |*x*. *x* ∈ *SIGT ϑ*$_2$} ⊆ {*Inl x* |*x*. *x* ∈ *SIGT* (*Times ϑ*$_1$ *ϑ*$_2$)}
by *auto*

from *IA*
have *IAs*:*Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* ϑ_1}
Iagree I J {*Inl x* |*x*. *x* ∈ *SIGT* ϑ_2}
using *Iagree-sub*[*OF Isubs*(*1*)] *Iagree-sub*[*OF Isubs*(*2*)] **by** *auto*
show *sterm-sem I* ϑ_1 (*fst* ν) ∗ *sterm-sem I* ϑ_2 (*fst* ν) = *sterm-sem J* ϑ_1 (*fst* ν') ∗ *sterm-sem J* ϑ_2 (*fst* ν')
using *IH1*[*OF VAs*(*1*) *IAs*(*1*)] *IH2*[*OF VAs*(*2*) *IAs*(*2*)] **by** *auto*
qed
qed (*unfold Vagree-def Iagree-def*, *auto*)

lemma *sum-unique-nonzero*:
fixes *i*::'*sv*::*finite* **and** *f*::'*sv* ⇒ *real*
assumes *restZero*:⋀*j*. *j*∈(*UNIV*::'*sv set*) ⟹ *j* ≠ *i* ⟹ *f j* = *0*
shows (∑*j*∈(*UNIV*::'*sv set*). *f j*) = *f i*
proof −
have (∑*j*∈(*UNIV*::'*sv set*). *f j*) = (∑*j*∈{*i*}. *f j*)
using *restZero* **by** (*intro sum.mono-neutral-cong-right*) *auto*
then show *?thesis*
by *simp*
qed

lemma *coincidence-frechet* :
fixes *I* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* **and** ν :: '*c state* **and** ν'::'*c state*
shows *dfree* ϑ ⟹ *Vagree* ν ν' (*FVDiff* ϑ) ⟹ *frechet I* ϑ (*fst* ν) (*snd* ν) = *frechet I* ϑ (*fst* ν') (*snd* ν')
proof (*induction rule*: *dfree.induct*)
case *dfree-Var* **then show** *?case*
by (*auto simp*: *inner-prod-eq Vagree-def*)
next
case *dfree-Const* **then show** *?case*
by *auto*
next
case (*dfree-Fun args var*)
assume *free*:(⋀*i*. *dfree* (*args i*))
assume *IH*:(⋀*i*. *Vagree* ν ν' (*FVDiff* (*args i*)) ⟹ *frechet I* (*args i*) (*fst* ν) (*snd* ν) = *frechet I* (*args i*) (*fst* ν') (*snd* ν'))
have *frees*:(⋀*i*. *dfree* (*args i*)) **using** *free* **by** (*auto simp add*: *rangeI*)
assume *agree*:*Vagree* ν ν' (*FVDiff* ($f *var args*))
have *agrees*:⋀*i*. *Vagree* ν ν' (*FVDiff* (*args i*)) **using** *agree agree-func* **by** *metis*
have *agrees'*:⋀*i*. *Vagree* ν ν' (*FVT* (*args i*))
subgoal for *i*
using *agrees*[*of i*] *FVDiff-sub*[*of args i*] **unfolding** *Vagree-def* **by** *blast*
done
have *sterms*:⋀*i*. *sterm-sem I* (*args i*) (*fst* ν) = *sterm-sem I* (*args i*) (*fst* ν')
by (*rule coincidence-sterm*[*of* ν ν', *OF agrees'*])
have *frechets*:⋀*i*. *frechet I* (*args i*) (*fst* ν) (*snd* ν) = *frechet I* (*args i*) (*fst* ν') (*snd* ν') **using** *IH agrees frees rangeI* **by** *blast*
show *?case*
using *agrees sterms frechets* **by** (*auto*)

next
case (*dfree-Plus t1 t2*)
assume *dfree1*:*dfree t1*
assume *IH1*:(*Vagree ν ν′* (*FVDiff t1*) ⟹ *frechet I t1* (*fst ν*) (*snd ν*) = *frechet I t1* (*fst ν′*) (*snd ν′*))
assume *dfree2*:*dfree t2*
assume *IH2*:(*Vagree ν ν′* (*FVDiff t2*) ⟹ *frechet I t2* (*fst ν*) (*snd ν*) = *frechet I t2* (*fst ν′*) (*snd ν′*))
assume *agree*:*Vagree ν ν′* (*FVDiff* (*Plus t1 t2*))
have *agree1*:*Vagree ν ν′* (*FVDiff t1*) **using** *agree agree-plus1* **by** (*blast*)
have *agree2*:*Vagree ν ν′* (*FVDiff t2*) **using** *agree agree-plus2* **by** (*blast*)
have *IH1′*:(*frechet I t1* (*fst ν*) (*snd ν*) = *frechet I t1* (*fst ν′*) (*snd ν′*))
using *IH1 agree1* **by** (*auto*)
have *IH2′*:(*frechet I t2* (*fst ν*) (*snd ν*) = *frechet I t2* (*fst ν′*) (*snd ν′*))
using *IH2 agree2* **by** (*auto*)
show *?case*
by (*metis FVT.simps*(*4*) *IH1′ IH2′ UnCI Vagree-def coincidence-sterm frechet.simps*(*3*) *mem-Collect-eq*)
next
case (*dfree-Times t1 t2*)
assume *dfree1*:*dfree t1*
assume *IH1*:(*Vagree ν ν′* (*FVDiff t1*) ⟹ *frechet I t1* (*fst ν*) (*snd ν*) = *frechet I t1* (*fst ν′*) (*snd ν′*))
assume *dfree2*:*dfree t2*
assume *IH2*:(*Vagree ν ν′* (*FVDiff t2*) ⟹ *frechet I t2* (*fst ν*) (*snd ν*) = *frechet I t2* (*fst ν′*) (*snd ν′*))
assume *agree*:*Vagree ν ν′* (*FVDiff* (*Times t1 t2*))
have *agree1*:*Vagree ν ν′* (*FVDiff t1*) **using** *agree agree-times1* **by** *blast*
have *agree2*:*Vagree ν ν′* (*FVDiff t2*) **using** *agree agree-times2* **by** *blast*
have *agree1′*:*Vagree ν ν′* (*FVT t1*)
using *agree1* **apply**(*auto simp add*: *Vagree-def*)
using *primify-contains* **by** *blast+*
have *agree2′*:*Vagree ν ν′* (*FVT t2*)
using *agree2* **apply**(*auto simp add*: *Vagree-def*)
using *primify-contains* **by** *blast+*
have *IH1′*:(*frechet I t1* (*fst ν*) (*snd ν*) = *frechet I t1* (*fst ν′*) (*snd ν′*))
using *IH1 agree1* **by** (*auto*)
have *IH2′*:(*frechet I t2* (*fst ν*) (*snd ν*) = *frechet I t2* (*fst ν′*) (*snd ν′*))
using *IH2 agree2* **by** (*auto*)
have *almost*:*Vagree ν ν′* (*FVT* (*Times t1 t2*)) ⟹ *frechet I* (*Times t1 t2*) (*fst ν*) (*snd ν*) = *frechet I* (*Times t1 t2*) (*fst ν′*) (*snd ν′*)
by (*auto simp add*: *UnCI Vagree-def agree IH1′ IH2′ coincidence-sterm*[*OF agree1′*, *of I*] *coincidence-sterm*[*OF agree2′*, *of I*])
show *?case*
using *agree FVDiff-sub almost*
by (*metis agree-supset*)
qed

lemma *coincidence-frechet′* :

fixes $I\ J :: ('a::finite, 'b::finite, 'c::finite)\ interp$ **and** $\nu :: 'c\ state$ **and** $\nu'::'c\ state$
shows $dfree\ \vartheta \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ \vartheta) \Longrightarrow Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in (SIGT\ \vartheta)\} \Longrightarrow frechet\ I\ \vartheta\ (fst\ \nu)\ (snd\ \nu) = frechet\ J\ \vartheta\ (fst\ \nu')\ (snd\ \nu')$
proof (*induction rule*: *dfree.induct*)
case *dfree-Var* **then show** *?case*
by (*auto simp*: *inner-prod-eq Vagree-def*)
next
case *dfree-Const* **then show** *?case*
by *auto*
next
case (*dfree-Fun args var*)
assume *free*:$(\bigwedge i.\ dfree\ (args\ i))$
assume *IH*:$(\bigwedge i.\ Vagree\ \nu\ \nu'\ (FVDiff\ (args\ i)) \Longrightarrow Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in SIGT\ (args\ i)\} \Longrightarrow frechet\ I\ (args\ i)\ (fst\ \nu)\ (snd\ \nu) = frechet\ J\ (args\ i)\ (fst\ \nu')\ (snd\ \nu'))$
have *frees*:$(\bigwedge i.\ dfree\ (args\ i))$ **using** *free* **by** (*auto simp add*: *rangeI*)
assume *agree*:$Vagree\ \nu\ \nu'\ (FVDiff\ (\$f\ var\ args))$
assume *IA*:$Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in SIGT\ (\$f\ var\ args)\}$
have *agrees*:$\bigwedge i.\ Vagree\ \nu\ \nu'\ (FVDiff\ (args\ i))$ **using** *agree agree-func* **by** *metis*
then have *agrees'*:$\bigwedge i.\ Vagree\ \nu\ \nu'\ (FVT\ (args\ i))$
using *agrees FVDiff-sub*
by (*metis agree-sub*)
from *Iagree-Func* [*OF IA*]**have** *fEq*:$Functions\ I\ var = Functions\ J\ var$ **by** *auto*
have *subs*:$\bigwedge i.\{Inl\ x \mid x.\ x \in SIGT\ (args\ i)\} \subseteq \{Inl\ x \mid x.\ x \in SIGT\ (\$f\ var\ args)\}$
by *auto*
from *IA* **have** *IAs*:$\bigwedge i.\ Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in SIGT\ (args\ i)\}$
using *Iagree-sub*[*OF subs*] **by** *auto*
have *sterms*:$\bigwedge i.\ sterm\text{-}sem\ I\ (args\ i)\ (fst\ \nu) = sterm\text{-}sem\ J\ (args\ i)\ (fst\ \nu')$
subgoal for *i*
using *frees agrees' coincidence-sterm'*[*of args i ν ν' I J*] *IAs*
by (*auto*)
done
have *frechets*:$\bigwedge i.\ frechet\ I\ (args\ i)\ (fst\ \nu)\ (snd\ \nu) = frechet\ J\ (args\ i)\ (fst\ \nu')\ (snd\ \nu')$
using *IH*[*OF agrees IAs*] *agrees frees rangeI* **by** *blast*
show *?case*
using *agrees agrees' sterms frechets fEq* **by** *auto*
next
case (*dfree-Plus t1 t2*)
assume *dfree1*:*dfree t1*
assume *dfree2*:*dfree t2*
assume *IH1*:$(Vagree\ \nu\ \nu'\ (FVDiff\ t1) \Longrightarrow Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in SIGT\ t1\} \Longrightarrow frechet\ I\ t1\ (fst\ \nu)\ (snd\ \nu) = frechet\ J\ t1\ (fst\ \nu')\ (snd\ \nu'))$
assume *IH2*:$(Vagree\ \nu\ \nu'\ (FVDiff\ t2) \Longrightarrow Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in SIGT\ t2\} \Longrightarrow frechet\ I\ t2\ (fst\ \nu)\ (snd\ \nu) = frechet\ J\ t2\ (fst\ \nu')\ (snd\ \nu'))$
assume *agree*:$Vagree\ \nu\ \nu'\ (FVDiff\ (Plus\ t1\ t2))$
assume *IA*:$Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in SIGT\ (Plus\ t1\ t2)\}$
have *subs*:$\{Inl\ x \mid x.\ x \in SIGT\ t1\} \subseteq \{Inl\ x \mid x.\ x \in SIGT\ (Plus\ t1\ t2)\}\ \{Inl\ x$

$|x.\ x \in SIGT\ t2\} \subseteq \{Inl\ x\ |x.\ x \in SIGT\ (Plus\ t1\ t2)\}$
by *auto*
from *IA*
have *IA1*:*Iagree I J* $\{Inl\ x\ |x.\ x \in SIGT\ t1\}$
and *IA2*:*Iagree I J* $\{Inl\ x\ |x.\ x \in SIGT\ t2\}$
using *Iagree-sub*[*OF subs*(*1*)] *Iagree-sub*[*OF subs*(*2*)] **by** *auto*
have *agree1*:*Vagree* $\nu\ \nu'$ (*FVDiff t1*) **using** *agree agree-plus1* **by** (*blast*)
have *agree2*:*Vagree* $\nu\ \nu'$ (*FVDiff t2*) **using** *agree agree-plus2* **by** (*blast*)
have *agree1'*:*Vagree* $\nu\ \nu'$ (*FVT t1*) **using** *agree1 primify-contains* **by** (*auto simp add*: *Vagree-def*, *metis*)
have *agree2'*:*Vagree* $\nu\ \nu'$ (*FVT t2*) **using** *agree2 primify-contains* **by** (*auto simp add*: *Vagree-def*, *metis*)
have *IH1'*:(*frechet I t1* (*fst* ν) (*snd* ν) = *frechet J t1* (*fst* ν') (*snd* ν'))
using *IH1 agree1 IA1* **by** (*auto*)
have *IH2'*:(*frechet I t2* (*fst* ν) (*snd* ν) = *frechet J t2* (*fst* ν') (*snd* ν'))
using *IH2 agree2 IA2* **by** (*auto*)
show *?case*
using *coincidence-sterm*[*OF agree1'*] *coincidence-sterm*[*OF agree1'*] *coincidence-sterm*[*OF agree2'*]
by (*auto simp add*: *IH1' IH2' UnCI Vagree-def*)

next
case (*dfree-Times t1 t2*)
assume *dfree1*:*dfree t1*
assume *dfree2*:*dfree t2*
assume *IH1*:(*Vagree* $\nu\ \nu'$ (*FVDiff t1*) $\Longrightarrow$ *Iagree I J* $\{Inl\ x\ |x.\ x \in SIGT\ t1\}$ $\Longrightarrow$ *frechet I t1* (*fst* ν) (*snd* ν) = *frechet J t1* (*fst* ν') (*snd* ν'))
assume *IH2*:(*Vagree* $\nu\ \nu'$ (*FVDiff t2*) $\Longrightarrow$ *Iagree I J* $\{Inl\ x\ |x.\ x \in SIGT\ t2\}$ $\Longrightarrow$ *frechet I t2* (*fst* ν) (*snd* ν) = *frechet J t2* (*fst* ν') (*snd* ν'))
assume *agree*:*Vagree* $\nu\ \nu'$ (*FVDiff* (*Times t1 t2*))
assume *IA*:*Iagree I J* $\{Inl\ x\ |x.\ x \in SIGT\ (Times\ t1\ t2)\}$
have *subs*:$\{Inl\ x\ |x.\ x \in SIGT\ t1\} \subseteq \{Inl\ x\ |x.\ x \in SIGT\ (Times\ t1\ t2)\}$ $\{Inl\ x\ |x.\ x \in SIGT\ t2\} \subseteq \{Inl\ x\ |x.\ x \in SIGT\ (Times\ t1\ t2)\}$
by *auto*
from *IA*
have *IA1*:*Iagree I J* $\{Inl\ x\ |x.\ x \in SIGT\ t1\}$
and *IA2*:*Iagree I J* $\{Inl\ x\ |x.\ x \in SIGT\ t2\}$
using *Iagree-sub*[*OF subs*(*1*)] *Iagree-sub*[*OF subs*(*2*)] **by** *auto*
have *agree1*:*Vagree* $\nu\ \nu'$ (*FVDiff t1*) **using** *agree agree-times1* **by** (*blast*)
then have *agree1'*:*Vagree* $\nu\ \nu'$ (*FVT t1*)
using *agree1 primify-contains* **by** (*auto simp add*: *Vagree-def*, *metis*)
have *agree2*:*Vagree* $\nu\ \nu'$ (*FVDiff t2*) **using** *agree agree-times2* **by** (*blast*)
then have *agree2'*:*Vagree* $\nu\ \nu'$ (*FVT t2*)
using *agree2 primify-contains* **by** (*auto simp add*: *Vagree-def*, *metis*)
have *IH1'*:(*frechet I t1* (*fst* ν) (*snd* ν) = *frechet J t1* (*fst* ν') (*snd* ν'))
using *IH1 agree1 IA1* **by** (*auto*)
have *IH2'*:(*frechet I t2* (*fst* ν) (*snd* ν) = *frechet J t2* (*fst* ν') (*snd* ν'))
using *IH2 agree2 IA2* **by** (*auto*)
note *co1* = *coincidence-sterm'*[*of t1* $\nu\ \nu'$ *I J*] **and** *co2* = *coincidence-sterm'*[*of*

t2 ν ν′ I J]
show *?case*
using *co1* [*OF dfree1 agree1′ IA1*] *co2* [*OF dfree2 agree2′ IA2*] *IH1′ IH2′* **by** *auto*
qed

lemma *coincidence-dterm*:
fixes *I* :: (*′a*::*finite*, *′b*::*finite*, *′c*::*finite*) *interp* **and** *ν* :: *′c state* **and** *ν′*::*′c state*
shows *dsafe ϑ* ⟹ *Vagree ν ν′* (*FVT ϑ*) ⟹ *dterm-sem I ϑ ν* = *dterm-sem I ϑ ν′*
proof (*induction rule*: *dsafe.induct*)
case (*dsafe-Fun args f*)
assume *safe*:(⋀*i*. *dsafe* (*args i*))
assume *IH*:⋀*i*. *Vagree ν ν′* (*FVT* (*args i*)) ⟹ *dterm-sem I* (*args i*) *ν* = *dterm-sem I* (*args i*) *ν′*
assume *agree*:*Vagree ν ν′* (*FVT* (*$f f args*))
then have ⋀*i*. *Vagree ν ν′* (*FVT* (*args i*))
using *agree-func-fvt* **by** (*metis*)
then show *?case*
using *safe coincidence-sterm IH rangeI* **by** (*auto*)
qed (*auto simp*: *Vagree-def directional-derivative-def coincidence-frechet*)

lemma *coincidence-dterm′*:
fixes *I J* :: (*′a*::*finite*, *′b*::*finite*, *′c*::*finite*) *interp* **and** *ν* :: *′c*::*finite state* **and** *ν′*::*′c*::*finite state*
shows *dsafe ϑ* ⟹ *Vagree ν ν′* (*FVT ϑ*) ⟹ *Iagree I J* {*Inl x* | *x*. *x* ∈ (*SIGT ϑ*)} ⟹ *dterm-sem I ϑ ν* = *dterm-sem J ϑ ν′*
proof (*induction rule*: *dsafe.induct*)
case (*dsafe-Fun args f*) **then**
have *safe*:(⋀*i*. *dsafe* (*args i*))
and *IH*:⋀*i*. *Vagree ν ν′* (*FVT* (*args i*)) ⟹ *Iagree I J* {*Inl x* | *x*. *x* ∈ (*SIGT* (*args i*))} ⟹ *dterm-sem I* (*args i*) *ν* = *dterm-sem J* (*args i*) *ν′*
and *agree*:*Vagree ν ν′* (*FVT* (*$f f args*))
and *IA*:*Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* (*$f f args*)}
by *auto*
have *subs*:⋀*i*. {*Inl x* |*x*. *x* ∈ *SIGT* (*args i*)} ⊆ {*Inl x* |*x*. *x* ∈ *SIGT* (*$f f args*)}
by *auto*
from *IA* **have** *IAs*:
⋀*i*. *Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* (*args i*)}
using *Iagree-sub* [*OF subs IA*] **by** *auto*
from *agree* **have** *agrees*:⋀*i*. *Vagree ν ν′* (*FVT* (*args i*))
using *agree-func-fvt* **by** (*metis*)
from *Iagree-Func* [*OF IA*] **have** *fEq*:*Functions I f* = *Functions J f* **by** *auto*
then show *?case*
using *safe coincidence-sterm IH*[*OF agrees IAs*] *rangeI agrees fEq*
by (*auto*)
next
case (*dsafe-Plus ϑ*$_1$ *ϑ*$_2$) **then**
have *safe*:*dsafe ϑ*$_1$ *dsafe ϑ*$_2$

and *IH1*:*Vagree* ν ν' (*FVT* ϑ_1) $\Longrightarrow$ *Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_1} $\Longrightarrow$ *dterm-sem I* ϑ_1 ν = *dterm-sem J* ϑ_1 ν'
and *IH2*:*Vagree* ν ν' (*FVT* ϑ_2) $\Longrightarrow$ *Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_2} $\Longrightarrow$ *dterm-sem I* ϑ_2 ν = *dterm-sem J* ϑ_2 ν'
and *VA*:*Vagree* ν ν' (*FVT* (*Plus* ϑ_1 ϑ_2))
and *IA*:*Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* (*Plus* ϑ_1 ϑ_2)}
by *auto*
from *VA* **have** *VA1*:*Vagree* ν ν' (*FVT* ϑ_1) **and** *VA2*:*Vagree* ν ν' (*FVT* ϑ_2)
unfolding *Vagree-def* **by** *auto*
have *subs*:{*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_1} $\subseteq$ {*Inl x* |*x*. *x* $\in$ *SIGT* (*Plus* ϑ_1 ϑ_2)}
{*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_2} $\subseteq$ {*Inl x* |*x*. *x* $\in$ *SIGT* (*Plus* ϑ_1 ϑ_2)}**by** *auto*
from *IA* **have** *IA1*:*Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_1} **and** *IA2*:*Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_2}
using *Iagree-sub subs* **by** *auto*
then show *?case*
using *IH1*[*OF VA1 IA1*] *IH2*[*OF VA2 IA2*] **by** *auto*
next
case (*dsafe-Times* ϑ_1 ϑ_2) **then**
have *safe*:*dsafe* ϑ_1 *dsafe* ϑ_2
and *IH1*:*Vagree* ν ν' (*FVT* ϑ_1) $\Longrightarrow$ *Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_1} $\Longrightarrow$ *dterm-sem I* ϑ_1 ν = *dterm-sem J* ϑ_1 ν'
and *IH2*:*Vagree* ν ν' (*FVT* ϑ_2) $\Longrightarrow$ *Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_2} $\Longrightarrow$ *dterm-sem I* ϑ_2 ν = *dterm-sem J* ϑ_2 ν'
and *VA*:*Vagree* ν ν' (*FVT* (*Times* ϑ_1 ϑ_2))
and *IA*:*Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* (*Times* ϑ_1 ϑ_2)}
by *auto*
from *VA* **have** *VA1*:*Vagree* ν ν' (*FVT* ϑ_1) **and** *VA2*:*Vagree* ν ν' (*FVT* ϑ_2)
unfolding *Vagree-def* **by** *auto*
have *subs*:{*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_1} $\subseteq$ {*Inl x* |*x*. *x* $\in$ *SIGT* (*Times* ϑ_1 ϑ_2)}
{*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_2} $\subseteq$ {*Inl x* |*x*. *x* $\in$ *SIGT* (*Times* ϑ_1 ϑ_2)}**by** *auto*
from *IA* **have** *IA1*:*Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_1} **and** *IA2*:*Iagree I J* {*Inl x* |*x*. *x* $\in$ *SIGT* ϑ_2}
using *Iagree-sub subs* **by** *auto*
then show *?case*
using *IH1*[*OF VA1 IA1*] *IH2*[*OF VA2 IA2*] **by** *auto*
qed (*auto simp*: *Vagree-def directional-derivative-def coincidence-frechet'*)

7.2 ODE Coincidence Theorems

lemma *coincidence-ode*:
fixes *I J* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* **and** ν :: '*c*::*finite state* **and** ν'::'*c*::*finite state*
shows *osafe ODE* $\Longrightarrow$
Vagree ν ν' (*Inl* ' *FVO ODE*) $\Longrightarrow$
Iagree I J ({*Inl x* | *x*. *Inl x* $\in$ *SIGO ODE*} $\cup$ {*Inr* (*Inr x*) | *x*. *Inr x* $\in$ *SIGO ODE*}) $\Longrightarrow$
ODE-sem I ODE (*fst* ν) = *ODE-sem J ODE* (*fst* ν')
proof (*induction rule*: *osafe.induct*)
case (*osafe-Var c*)

then show *?case*
proof (*auto*)
assume *VA*:*Vagree* ν ν' (*range Inl*)
have *eqV*:(*fst* ν) = (*fst* ν')
using *agree-UNIV-fst*[*OF VA*] **by** *auto*
assume *IA*:*Iagree I J* {*Inr* (*Inr c*)}
have *eqIJ*:*ODEs I c* = *ODEs J c*
using *Iagree-ODE*[*OF IA*] **by** *auto*
show *ODEs I c* (*fst* ν) = *ODEs J c* (*fst* ν')
by (*auto simp add*: *eqV eqIJ*)
qed
next
case (*osafe-Sing* ϑ *x*)
then show *?case*
proof (*auto*)
assume *free*:*dfree* ϑ
and *VA*:*Vagree* ν ν' (*insert* (*Inl x*) (*Inl* ' {*x*. *Inl x* ∈ *FVT* ϑ}))
and *IA*:*Iagree I J* {*Inl x* |*x*. *x* ∈ *SIGT* ϑ}
from *VA* **have** *VA'*:*Vagree* ν ν' {*Inl x* | *x*. *Inl x* ∈ *FVT* ϑ} **unfolding** *Vagree-def*
by *auto*
have *agree-Lem*:⋀ϑ. *dfree* ϑ ⟹ *Vagree* ν ν' {*Inl x* | *x*. *Inl x* ∈ *FVT* ϑ} ⟹
Vagree ν ν' (*FVT* ϑ)
subgoal for ϑ
apply(*induction rule*: *dfree.induct*)
by(*auto simp add*: *Vagree-def*)
done
have *trm*:*sterm-sem I* ϑ (*fst* ν) = *sterm-sem J* ϑ (*fst* ν')
using *coincidence-sterm'* *free VA' IA agree-Lem*[*of* ϑ, *OF free*] **by** *blast*
show (λi. *if i* = *x then sterm-sem I* ϑ (*fst* ν) *else 0*) =
(λi. *if i* = *x then sterm-sem J* ϑ (*fst* ν') *else 0*)
by (*auto simp add*: *vec-eq-iff trm*)
qed
next
case (*osafe-Prod ODE1 ODE2*)
then show *?case*
proof (*auto*)
assume *safe1*:*osafe ODE1*
and *safe2*:*osafe ODE2*
and *disjoint*:*ODE-dom ODE1* ∩ *ODE-dom ODE2* = {}
and *IH1*:*Vagree* ν ν' (*Inl* ' *FVO ODE1*) ⟹
Iagree I J ({*Inl x* |*x*. *Inl x* ∈ *SIGO ODE1*} ∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO*
ODE1}) ⟹ *ODE-sem I ODE1* (*fst* ν) = *ODE-sem J ODE1* (*fst* ν')
and *IH2*:*Vagree* ν ν' (*Inl* ' *FVO ODE2*) ⟹
Iagree I J ({*Inl x* |*x*. *Inl x* ∈ *SIGO ODE2*} ∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO*
ODE2}) ⟹ *ODE-sem I ODE2* (*fst* ν) = *ODE-sem J ODE2* (*fst* ν')
and *VA*:*Vagree* ν ν' (*Inl* ' (*FVO ODE1* ∪ *FVO ODE2*))
and *IA*:*Iagree I J* ({*Inl x* |*x*. *Inl x* ∈ *SIGO ODE1* ∨ *Inl x* ∈ *SIGO ODE2*}
∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE1* ∨ *Inr x* ∈ *SIGO ODE2*})
let *?IA* = ({*Inl x* |*x*. *Inl x* ∈ *SIGO ODE1* ∨ *Inl x* ∈ *SIGO ODE2*} ∪ {*Inr* (*Inr*

```
x) |x. Inr x ∈ SIGO ODE1 ∨ Inr x ∈ SIGO ODE2})
    have FVsubs:
      Inl ‘ FVO ODE2 ⊆ Inl ‘ (FVO ODE1 ∪ FVO ODE2)
      Inl ‘ FVO ODE1 ⊆ Inl ‘ (FVO ODE1 ∪ FVO ODE2)
      by auto
    from VA
    have VA1:Vagree ν ν' (Inl ‘ FVO ODE1)
     and VA2:Vagree ν ν' (Inl ‘ FVO ODE2)
      using agree-sub[OF FVsubs(1)] agree-sub[OF FVsubs(2)]
      by (auto)
    have SIGsubs:
      ({Inl x |x. Inl x ∈ SIGO ODE1} ∪ {Inr (Inr x) |x. Inr x ∈ SIGO ODE1})
⊆ ?IA
      ({Inl x |x. Inl x ∈ SIGO ODE2} ∪ {Inr (Inr x) |x. Inr x ∈ SIGO ODE2})
⊆ ?IA
      by auto
    from IA
    have IA1:Iagree I J ({Inl x |x. Inl x ∈ SIGO ODE1} ∪ {Inr (Inr x) |x. Inr x
∈ SIGO ODE1})
     and IA2:Iagree I J ({Inl x |x. Inl x ∈ SIGO ODE2} ∪ {Inr (Inr x) |x. Inr x
∈ SIGO ODE2})
      using Iagree-sub[OF SIGsubs(1)] Iagree-sub[OF SIGsubs(2)] by auto
    show ODE-sem I ODE1 (fst ν) + ODE-sem I ODE2 (fst ν) = ODE-sem J
ODE1 (fst ν') + ODE-sem J ODE2 (fst ν')
      using IH1[OF VA1 IA1] IH2[OF VA2 IA2] by auto
  qed
qed

lemma coincidence-ode':
  fixes I J :: ('a::finite, 'b::finite, 'c::finite) interp and ν :: 'c simple-state and
ν'::'c simple-state
  shows osafe ODE ⟹
        VSagree ν ν' (FVO ODE) ⟹
        Iagree I J ({Inl x | x. Inl x ∈ SIGO ODE} ∪ {Inr (Inr x) | x. Inr x ∈
SIGO ODE}) ⟹
        ODE-sem I ODE ν = ODE-sem J ODE ν'
  using coincidence-ode[of ODE (ν, χ i. 0) (ν', χ i. 0) I J]
  apply(auto)
  unfolding VSagree-def Vagree-def apply auto
  done

lemma alt-sem-lemma:⋀ I::('a::finite,'b::finite,'c::finite) interp. ⋀ ODE::('a::finite,'c::finite)
ODE. ⋀sol. ⋀t::real. ⋀ ab. osafe ODE ⟹
  ODE-sem I ODE (sol t) = ODE-sem I ODE (χ i. if i ∈ FVO ODE then sol t $
i else ab $ i)
proof −
  fix I::('a,'b,'c) interp
    and ODE::('a,'c) ODE
    and sol
```

and *t*::*real*
and *ab*
assume *safe*:*osafe ODE*
have *VA*:*VSagree* (*sol t*) (*χ i*. *if i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*) (*FVO ODE*)
unfolding *VSagree-def Vagree-def* **by** *auto*
have *IA*: *Iagree I I* ({*Inl x* | *x*. *Inl x* ∈ *SIGO ODE*} ∪ {*Inr* (*Inr x*) | *x*. *Inr x* ∈ *SIGO ODE*}) **unfolding** *Iagree-def* **by** *auto*
show *ODE-sem I ODE* (*sol t*) = *ODE-sem I ODE* (*χ i*. *if i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*)
using *coincidence-ode'*[*OF safe VA IA*] **by** *auto*
qed

lemma *bvo-to-fvo*:*Inl x* ∈ *BVO ODE* ⟹ *x* ∈ *FVO ODE*
proof (*induction ODE*)
qed *auto*

lemma *ode-to-fvo*:*x* ∈ *ODE-vars I ODE* ⟹ *x* ∈ *FVO ODE*
proof (*induction ODE*)
qed *auto*

definition *coincide-hp* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *hp* ⇒ ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* ⇒ ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* ⇒ *bool*
where *coincide-hp α I J* ⟷ (∀ *ν ν' μ V*. *Iagree I J* (*SIGP α*) ⟶ *Vagree ν ν' V* ⟶ *V* ⊇ (*FVP α*) ⟶ (*ν*, *μ*) ∈ *prog-sem I α* ⟶ (∃ *μ'*. (*ν'*, *μ'*) ∈ *prog-sem J α* ∧ *Vagree μ μ'* (*MBV α* ∪ *V*)))

definition *ode-sem-equiv* ::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *hp* ⇒ ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *interp* ⇒ *bool*
where *ode-sem-equiv α I* ⟷
(∀ *ODE*::('*a*::*finite*,'*c*::*finite*) *ODE*. ∀ *φ*::('*a*::*finite*,'*b*::*finite*,'*c*::*finite*)*formula*. *osafe ODE* ⟶ *fsafe φ* ⟶
(*α* = *EvolveODE ODE φ*) ⟶
{(*ν*, *mk-v I ODE ν* (*sol t*)) | *ν sol t*.
t ≥ *0* ∧
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE ν x* ∈ *fml-sem I φ*} ∧
VSagree (*sol 0*) (*fst ν*) {*x* | *x*. *Inl x* ∈ *FVP* (*EvolveODE ODE φ*)}} =
{(*ν*, *mk-v I ODE ν* (*sol t*)) | *ν sol t*.
t ≥ *0* ∧
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE ν x* ∈ *fml-sem I φ*} ∧
sol 0 = *fst ν*})

definition *coincide-hp'* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *hp* ⇒ *bool*
where *coincide-hp' α* ⟷ (∀ *I J*. *coincide-hp α I J* ∧ *ode-sem-equiv α I*)

definition *coincide-fml* :: ('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *formula* ⇒ *bool*
where *coincide-fml φ* ⟷ (∀ *ν ν' I J* . *Iagree I J* (*SIGF φ*) ⟶ *Vagree ν ν'*

$(FVF\ \varphi) \longrightarrow \nu \in \mathit{fml\text{-}sem}\ I\ \varphi \longleftrightarrow \nu' \in \mathit{fml\text{-}sem}\ J\ \varphi)$

lemma *coinc-fml* [*simp*]: *coincide-fml* φ $= (\forall\ \nu\ \nu'\ I\ J.\ \mathit{Iagree}\ I\ J\ (\mathit{SIGF}\ \varphi) \longrightarrow \mathit{Vagree}\ \nu\ \nu'\ (\mathit{FVF}\ \varphi) \longrightarrow \nu \in \mathit{fml\text{-}sem}\ I\ \varphi \longleftrightarrow \nu' \in \mathit{fml\text{-}sem}\ J\ \varphi)$
unfolding *coincide-fml-def* **by** *auto*

7.3 Coincidence Theorems for Programs and Formulas

lemma *coincidence-hp-fml*:
fixes α::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *hp*
fixes φ::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *formula*
shows (*hpsafe* $\alpha \longrightarrow$ *coincide-hp'* $\alpha) \wedge$ (*fsafe* $\varphi \longrightarrow$ *coincide-fml* φ)
proof (*induction rule*: *hpsafe-fsafe.induct*)
case (*hpsafe-Pvar x*)
thus *?case*
apply(*unfold coincide-hp'-def* | *rule allI* | *rule conjI*)+
prefer *2* **unfolding** *ode-sem-equiv-def* **subgoal by** *auto*
unfolding *coincide-hp-def* **apply**(*auto*)
subgoal for *I J a b aa ba ab bb V*
proof −
assume *IA*:*Iagree I J* {*Inr* (*Inr x*)}
have *Peq*:$\bigwedge y.\ y \in \mathit{Programs}\ I\ x \longleftrightarrow y \in \mathit{Programs}\ J\ x$ **using** *Iagree-Prog*[*OF IA*] **by** *auto*
assume *agree*:*Vagree* (*a*, *b*) (*aa*, *ba*) *V*
and *sub*:*UNIV* $\subseteq$ *V*
and *sem*:((*a*, *b*), *ab*, *bb*) $\in$ *Programs I x*
from *agree-UNIV-eq*[*OF agree-sub* [*OF sub agree*]]
have *eq*:(*a*,*b*) = (*aa*,*ba*) **by** *auto*
hence *sem'*:((*aa*,*ba*), (*ab*,*bb*)) $\in$ *Programs I x*
using *sem* **by** *auto*
have *triv-sub*:*V* $\subseteq$ *UNIV* **by** *auto*
have *VA*:*Vagree* (*ab*,*bb*) (*ab*,*bb*) *V* **using** *agree-sub*[*OF triv-sub agree-refl*[*of* (*ab*,*bb*)]] *eq*
by *auto*
show $\exists$ *a b*. ((*aa*, *ba*), *a*, *b*) $\in$ *Programs J x* $\wedge$ *Vagree* (*ab*, *bb*) (*a*, *b*) *V*
apply(*rule exI*[**where** *x*=*ab*])
apply(*rule exI*[**where** *x*=*bb*])
using *sem eq VA* **by** (*auto simp add*: *Peq*)
qed
done
next
case (*hpsafe-Assign e x*) **then**
show *?case*
proof (*auto simp only*: *coincide-hp'-def ode-sem-equiv-def coincide-hp-def*)
fix *I J* :: ('*a*::*finite*,'*b*::*finite*,'*c*::*finite*) *interp*
and *ν1 ν2 ν'1 ν'2 μ1 μ2 V*
assume *safe*:*dsafe e*
and *IA*:*Iagree I J* (*SIGP* (*x* := *e*))
and *VA*:*Vagree* (*ν1*, *ν2*) (*ν'1*, *ν'2*) *V*

and *sub*:*FVP* $(x := e) \subseteq V$
and *sem*:$((\nu 1, \nu 2), (\mu 1, \mu 2)) \in$ *prog-sem I* $(x := e)$
from *VA* **have** *VA'*:*Vagree* $(\nu 1, \nu 2)$ $(\nu' 1, \nu' 2)$ $(FVT\ e)$ **unfolding** *FVP.simps Vagree-def* **using** *sub* **by** *auto*
have *Ssub*:$\{Inl\ x \mid x.\ x \in SIGT\ e\} \subseteq (SIGP\ (x := e))$ **by** *auto*
from *IA* **have** *IA'*:*Iagree I J* $\{Inl\ x \mid x.\ x \in SIGT\ e\}$ **using** *Ssub* **unfolding** *SIGP.simps* **by** *auto*
have $((\nu 1, \nu 2), repv\ (\nu 1, \nu 2)\ x\ (dterm\text{-}sem\ I\ e\ (\nu 1, \nu 2))) \in prog\text{-}sem\ I\ (x := e)$ **by** *auto*
then have *sem'*:$((\nu' 1, \nu' 2), repv\ (\nu' 1, \nu' 2)\ x\ (dterm\text{-}sem\ J\ e\ (\nu' 1, \nu' 2))) \in$ *prog-sem J* $(x := e)$
using *coincidence-dterm' safe VA' IA'* **by** *auto*
from *sem* **have** *eq*:$(\mu 1, \mu 2) = (repv\ (\nu 1, \nu 2)\ x\ (dterm\text{-}sem\ I\ e\ (\nu 1, \nu 2)))$ **by** *auto*
have *VA''*:*Vagree* $(\mu 1, \mu 2)$ $(repv\ (\nu' 1, \nu' 2)\ x\ (dterm\text{-}sem\ J\ e\ (\nu' 1, \nu' 2)))$ $(MBV\ (x := e) \cup V)$
using *coincidence-dterm'*[*of e* $(\nu 1,\nu 2)$ $(\nu' 1,\nu' 2)$ *I J*] *safe VA' IA' eq agree-refl VA* **unfolding** *MBV.simps Vagree-def*
by *auto*
show $\exists \mu'.\ ((\nu' 1, \nu' 2), \mu') \in prog\text{-}sem\ J\ (x := e) \wedge Vagree\ (\mu 1, \mu 2)\ \mu'\ (MBV\ (x := e) \cup V)$
using *VA'' sem'* **by** *blast*
qed
next
case (*hpsafe-DiffAssign e x*) **then show** *?case*
proof (*auto simp only*: *coincide-hp'-def ode-sem-equiv-def coincide-hp-def*)
fix *I J*::$('a,'b,'c)$ *interp*
and $\nu\ \nu'\ \mu\ V$
assume *safe*:*dsafe e*
and *IA*:*Iagree I J* (*SIGP* (*DiffAssign x e*))
and *VA*:*Vagree* $\nu\ \nu'\ V$
and *sub*:*FVP* (*DiffAssign x e*) $\subseteq V$
and *sem*:$(\nu, \mu) \in$ *prog-sem I* (*DiffAssign x e*)
from *VA* **have** *VA'*:*Vagree* $\nu\ \nu'$ $(FVT\ e)$ **unfolding** *FVP.simps Vagree-def* **using** *sub* **by** *auto*
have *Ssub*:$\{Inl\ x \mid x.\ x \in SIGT\ e\} \subseteq (SIGP\ (DiffAssign\ x\ e))$ **by** *auto*
from *IA* **have** *IA'*:*Iagree I J* $\{Inl\ x \mid x.\ x \in SIGT\ e\}$ **using** *Ssub* **unfolding** *SIGP.simps* **by** *auto*
have $(\nu, repv\ \nu\ x\ (dterm\text{-}sem\ I\ e\ \nu)) \in prog\text{-}sem\ I\ (x := e)$ **by** *auto*
then have *sem'*:$(\nu', repd\ \nu'\ x\ (dterm\text{-}sem\ J\ e\ \nu')) \in$ *prog-sem J* (*DiffAssign x e*)
using *coincidence-dterm' safe VA' IA'* **by** *auto*
from *sem* **have** *eq*:$\mu = (repd\ \nu\ x\ (dterm\text{-}sem\ I\ e\ \nu))$ **by** *auto*
have *VA'*:*Vagree* μ $(repd\ \nu'\ x\ (dterm\text{-}sem\ J\ e\ \nu'))$ $(MBV\ (DiffAssign\ x\ e) \cup V)$
using *coincidence-dterm'*[*OF safe VA', of I J, OF IA'*] *eq agree-refl VA* **unfolding** *MBV.simps Vagree-def*
by *auto*
show $\exists \mu'.\ (\nu', \mu') \in prog\text{-}sem\ J\ (DiffAssign\ x\ e) \wedge Vagree\ \mu\ \mu'\ (MBV$

```
(DiffAssign x e) ∪ V)
      using VA' sem' by blast
  qed

next
  case (hpsafe-Test P) then
  show ?case
  proof (auto simp add:coincide-hp'-def ode-sem-equiv-def coincide-hp-def)
    fix I J::('a,'b,'c) interp and ν ν' ω ω' ::'c simple-state
      and V
    assume safe:fsafe P
    assume ∀ a b aa ba I J. (Iagree I J (SIGF P) ⟶ Vagree (a, b) (aa, ba) (FVF
P) ⟶ ((a, b) ∈ fml-sem I P) = ((aa, ba) ∈ fml-sem J P))
     hence IH:Iagree I J (SIGF P) ⟹ Vagree (ν, ν') (ω, ω') (FVF P) ⟹ ((ν,
ν') ∈ fml-sem I P) = ((ω, ω') ∈ fml-sem J P)
      by auto
    assume IA:Iagree I J (SIGF P)
    assume VA:Vagree (ν, ν') (ω, ω') V
    assume sub:FVF P ⊆ V
      hence VA':Vagree (ν, ν') (ω, ω') (FVF P) using agree-supset VA by auto
    assume sem:(ν, ν') ∈ fml-sem I P
    show (ω, ω') ∈ fml-sem J P using IH[OF IA VA'] sem by auto
    qed
next
  case (hpsafe-Evolve ODE P) then show ?case
    proof (unfold coincide-hp'-def)
      assume osafe:osafe ODE
      assume fsafe:fsafe P
      assume IH:coincide-fml P
       from IH have IHF:⋀ν ν' I J. Iagree I J (SIGF P) ⟹ Vagree ν ν' (FVF
P) ⟹ (ν ∈ fml-sem I P) = (ν' ∈ fml-sem J P)
        unfolding coincide-fml-def by auto
      have equiv:⋀I. ode-sem-equiv (EvolveODE ODE P) I
        subgoal for I
          apply(unfold ode-sem-equiv-def)
          apply(rule allI)+
          subgoal for ODE φ
            apply(rule impI)+
            apply(auto)
            subgoal for aa ba ab bb sol t
              apply(rule exI[where x=(λt. χ i. if i ∈ FVO ODE then sol t $ i else
ab $ i)])
             apply(rule conjI)
            subgoal using mk-v-agree[of I ODE (ab,bb) sol t] mk-v-agree[of I ODE
(ab,bb) (χ i. if  i ∈ FVO ODE then sol t $ i else ab $ i)]
               unfolding Vagree-def VSagree-def by (auto simp add: vec-eq-iff)
             apply(rule exI[where x=t])
             apply(rule conjI)
             subgoal
```

apply(*rule agree-UNIV-eq*)
using *mk-v-agree*[*of I ODE* (*ab*,*bb*) *sol t*]
mk-v-agree[*of I ODE* (*ab*,*bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*)]
mk-v-agree[*of I ODE* (χ *i*. *if* *i* ∈ *FVO ODE then sol 0* $ *i else ab* $ *i*, *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*)]
unfolding *Vagree-def VSagree-def*
apply(*auto*)
subgoal for *i*
apply(*cases Inl i* ∈ *BVO ODE*)
using *bvo-to-fvo*[*of i ODE*] **apply** (*metis* (*no-types*, *lifting*))
apply(*erule allE*[**where** *x*=*i*])+
using *Inl-Inr-False imageE ode-to-fvo*
proof −
assume *a1*: (*aa*, *ba*) = *mk-v I ODE* (*ab*, *bb*) (*sol t*)
assume *a2*: (*Inl i* ∈ *BVO ODE* ⟶ *sol 0* $ *i* = *ab* $ *i*) ∧ (*Inl i* ∈ *Inl* ‘ *FVO ODE* ⟶ *sol 0* $ *i* = *ab* $ *i*) ∧ (*Inl i* ∈ *FVF* φ ⟶ *sol 0* $ *i* = *ab* $ *i*)
assume *a3*: (*Inl i*::′*c* + ′*c*) ∉ *Inl* ‘ *ODE-vars I ODE* ∧ *Inl i* ∉ *Inr* ‘ *ODE-vars I ODE* ⟶ *fst* (*mk-v I ODE* (*ab*, *bb*) (*sol t*)) $ *i* = *ab* $ *i*
assume *a4*: (*Inl i*::′*c* + ′*c*) ∉ *Inl* ‘ *ODE-vars I ODE* ∧ *Inl i* ∉ *Inr* ‘ *ODE-vars I ODE* ⟶ *fst* (*mk-v I ODE* (χ *i*. *if i* ∈ *FVO ODE then sol 0* $ *i else ab* $ *i*, *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*)) $ *i* = (*if* *i* ∈ *FVO ODE then sol 0* $ *i else ab* $ *i*)
assume *a5*: ((*Inl i*::′*c* + ′*c*) ∈ *Inl* ‘ *ODE-vars I ODE* ⟶ *fst* (*mk-v I ODE* (*ab*, *bb*) (*sol t*)) $ *i* = *sol t* $ *i*) ∧ (*Inl i* ∈ *Inr* ‘ *ODE-vars I ODE* ⟶ *fst* (*mk-v I ODE* (*ab*, *bb*) (*sol t*)) $ *i* = *sol t* $ *i*)
assume *a6*: ((*Inl i*::′*c* + ′*c*) ∈ *Inl* ‘ *ODE-vars I ODE* ⟶ *fst* (*mk-v I ODE* (χ *i*. *if* *i* ∈ *FVO ODE then sol 0* $ *i else ab* $ *i*, *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*)) $ *i* = (*if* *i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*)) ∧ (*Inl i* ∈ *Inr* ‘ *ODE-vars I ODE* ⟶ *fst* (*mk-v I ODE* (χ *i*. *if* *i* ∈ *FVO ODE then sol 0* $ *i else ab* $ *i*, *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*)) $ *i* = (*if* *i* ∈ *FVO ODE then sol t* $ *i else ab* $ *i*))
have *f7*: *fst* (*aa*, *ba*) $ *i* = *sol t* $ *i* ∨ (*Inl i*::′*c* + ′*c*) ∉ *Inl* ‘ *ODE-vars I ODE*
using *a5 a1* **by** *auto*
have *f8*: *fst* (*aa*, *ba*) $ *i* = *ab* $ *i* ∨ (*Inl i*::′*c* + ′*c*) ∈ *Inl* ‘ *ODE-vars I ODE*
using *a3 a1* **by** *fastforce*
moreover
{ **assume** *fst* (*mk-v I ODE* (χ *c*. *if* *c* ∈ *FVO ODE then sol 0* $ *c else ab* $ *c*, *bb*) (χ *c*. *if* *c* ∈ *FVO ODE then sol t* $ *c else ab* $ *c*)) $ *i* ≠ *ab* $ *i*
{ **assume** *fst* (*mk-v I ODE* (χ *c*. *if* *c* ∈ *FVO ODE then sol 0* $ *c else ab* $ *c*, *bb*) (χ *c*. *if* *c* ∈ *FVO ODE then sol t* $ *c else ab* $ *c*)) $ *i* ≠ *ab* $ *i* ∧ *Inl i* ∉ *Inr* ‘ *ODE-vars I ODE*
have *i* ∈ *FVO ODE* ∧ *fst* (*aa*, *ba*) $ *i* = *ab* $ *i* ⟶ *fst* (*mk-v I ODE* (χ *c*. *if* *c* ∈ *FVO ODE then sol 0* $ *c else ab* $ *c*, *bb*) (χ *c*. *if* *c* ∈ *FVO ODE then sol t* $ *c else ab* $ *c*)) $ *i* ≠ *sol t* $ *i* ∧ (*Inl i*::′*c* + ′*c*) ∈ *Inl* ‘ *ODE-vars I ODE* ∨ *fst* (*mk-v I ODE* (χ *c*. *if* *c* ∈ *FVO ODE then sol 0* $ *c else ab* $ *c*, *bb*) (χ *c*. *if* *c* ∈ *FVO ODE then sol t* $ *c else ab* $ *c*)) $ *i* = *ab* $ *i*

using *f7 a4 a2* **by** *force* **}**
then have $i \in FVO\ ODE \wedge fst\ (aa, ba)\ \$\ i = ab\ \$\ i \longrightarrow fst\ (mk\text{-}v\ I\ ODE\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ 0\ \$\ c\ else\ ab\ \$\ c, bb)\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ t\ \$\ c\ else\ ab\ \$\ c))\ \$\ i \neq sol\ t\ \$\ i \wedge (Inl\ i::'c + 'c) \in Inl\ `\ ODE\text{-}vars\ I\ ODE \vee fst\ (mk\text{-}v\ I\ ODE\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ 0\ \$\ c\ else\ ab\ \$\ c, bb)\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ t\ \$\ c\ else\ ab\ \$\ c))\ \$\ i = ab\ \$\ i$
by *blast* **}**
ultimately have $i \in FVO\ ODE \longrightarrow fst\ (mk\text{-}v\ I\ ODE\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ 0\ \$\ c\ else\ ab\ \$\ c, bb)\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ t\ \$\ c\ else\ ab\ \$\ c))\ \$\ i = fst\ (aa, ba)\ \$\ i$
using *f7 a6* **by** *fastforce*
then have $fst\ (mk\text{-}v\ I\ ODE\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ 0\ \$\ c\ else\ ab\ \$\ c, bb)\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ sol\ t\ \$\ c\ else\ ab\ \$\ c))\ \$\ i = fst\ (aa, ba)\ \$\ i$
using *f8 a4 ode-to-fvo* **by** *fastforce*
then show *?thesis*
using *a1* **by** *presburger*
qed
proof −
fix $i :: 'c$
assume *a1*: *osafe ODE*
assume *a2*: $(aa, ba) = mk\text{-}v\ I\ ODE\ (ab, bb)\ (sol\ t)$
assume *a3*: $\forall i.\ (Inr\ i \in Inl\ `\ ODE\text{-}vars\ I\ ODE \longrightarrow snd\ (mk\text{-}v\ I\ ODE\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ 0\ \$\ i\ else\ ab\ \$\ i, bb)\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ t\ \$\ i\ else\ ab\ \$\ i))\ \$\ i = ODE\text{-}sem\ I\ ODE\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ t\ \$\ i\ else\ ab\ \$\ i)\ \$\ i) \wedge ((Inr\ i::'c + 'c) \in Inr\ `\ ODE\text{-}vars\ I\ ODE \longrightarrow snd\ (mk\text{-}v\ I\ ODE\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ 0\ \$\ i\ else\ ab\ \$\ i, bb)\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ t\ \$\ i\ else\ ab\ \$\ i))\ \$\ i = ODE\text{-}sem\ I\ ODE\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ t\ \$\ i\ else\ ab\ \$\ i)\ \$\ i)$
assume *a4*: $\forall i.\ (Inr\ i \in Inl\ `\ ODE\text{-}vars\ I\ ODE \longrightarrow snd\ (mk\text{-}v\ I\ ODE\ (ab, bb)\ (sol\ t))\ \$\ i = ODE\text{-}sem\ I\ ODE\ (sol\ t)\ \$\ i) \wedge ((Inr\ i::'c + 'c) \in Inr\ `\ ODE\text{-}vars\ I\ ODE \longrightarrow snd\ (mk\text{-}v\ I\ ODE\ (ab, bb)\ (sol\ t))\ \$\ i = ODE\text{-}sem\ I\ ODE\ (sol\ t)\ \$\ i)$
assume *a5*: $\forall i.\ Inr\ i \notin Inl\ `\ ODE\text{-}vars\ I\ ODE \wedge (Inr\ i::'c + 'c) \notin Inr\ `\ ODE\text{-}vars\ I\ ODE \longrightarrow snd\ (mk\text{-}v\ I\ ODE\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ 0\ \$\ i\ else\ ab\ \$\ i, bb)\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ t\ \$\ i\ else\ ab\ \$\ i))\ \$\ i = bb\ \$\ i$
assume *a6*: $\forall i.\ Inr\ i \notin Inl\ `\ ODE\text{-}vars\ I\ ODE \wedge (Inr\ i::'c + 'c) \notin Inr\ `\ ODE\text{-}vars\ I\ ODE \longrightarrow snd\ (mk\text{-}v\ I\ ODE\ (ab, bb)\ (sol\ t))\ \$\ i = bb\ \$\ i$
have $\bigwedge i\ f\ r\ v.\ ODE\text{-}sem\ (i::('a, 'b, 'c)\ interp)\ ODE\ (\chi\ c.\ if\ \ c \in FVO\ ODE\ then\ f\ (r::real)\ \$\ c\ else\ v\ \$\ c) = ODE\text{-}sem\ i\ ODE\ (f\ r)$
using *a1* **by** (*metis* (*no-types*) *alt-sem-lemma*)
moreover
{ assume $(Inr\ i::'c + 'c) \notin Inr\ `\ ODE\text{-}vars\ I\ ODE$
moreover
{ assume $(Inr\ i::'c + 'c) \notin Inr\ `\ ODE\text{-}vars\ I\ ODE \wedge Inr\ i \notin Inl\ `\ ODE\text{-}vars\ I\ ODE \wedge (Inr\ i::'c + 'c) \notin Inr\ `\ ODE\text{-}vars\ I\ ODE \wedge Inr\ i \notin Inl\ `\ ODE\text{-}vars\ I\ ODE$
then have $snd\ (aa, ba)\ \$\ i = bb\ \$\ i \wedge (Inr\ i::'c + 'c) \notin Inr\ `\ ODE\text{-}vars\ I\ ODE \wedge Inr\ i \notin Inl\ `\ ODE\text{-}vars\ I\ ODE$

using *a6 a2* **by** *presburger*
then have *snd* (*mk-v I ODE* (χ *c. if* $c \in$ *FVO ODE then sol 0* \$ *c else ab* \$ *c*, *bb*) (χ *c. if* $c \in$ *FVO ODE then sol t* \$ *c else ab* \$ *c*)) \$ *i* = *snd* (*aa*, *ba*) \$ *i*
using *a5* **by** *presburger* **}**
ultimately have *snd* (*mk-v I ODE* (χ *c. if* $c \in$ *FVO ODE then sol 0* \$ *c else ab* \$ *c*, *bb*) (χ *c. if* $c \in$ *FVO ODE then sol t* \$ *c else ab* \$ *c*)) \$ *i* = *snd* (*aa*, *ba*) \$ *i*
by *blast* **}**
ultimately show *snd* (*mk-v I ODE* (*ab*, *bb*) (*sol t*)) \$ *i* = *snd* (*mk-v I ODE* (χ *c. if* $c \in$ *FVO ODE then sol 0* \$ *c else ab* \$ *c*, *bb*) (χ *c. if* $c \in$ *FVO ODE then sol t* \$ *c else ab* \$ *c*)) \$ *i*
using *a4 a3 a2* **by** *fastforce*
qed
apply(*rule conjI*)
subgoal by *auto*
apply(*auto simp only*: *solves-ode-def has-vderiv-on-def has-vector-derivative-def*)
apply (*rule has-derivative-vec*[*THEN has-derivative-eq-rhs*])
defer
apply (*rule ext*)
apply (*subst scaleR-vec-def*)
apply (*rule refl*)
subgoal for *x* **unfolding** *VSagree-def* **apply** *auto*
proof −
assume *osafe*:*osafe ODE*
and *fsafe*:*fsafe* φ
and *eqP*:*P* = φ
and *aaba*: (*aa*, *ba*) = *mk-v I ODE* (*ab*, *bb*) (*sol t*)
and *all*:∀ *i*. (*Inl i* ∈ *BVO ODE* ⟶ *sol 0* \$ *i* = *ab* \$ *i*) ∧ (*Inl i* ∈ *Inl* ' *FVO ODE* ⟶ *sol 0* \$ *i* = *ab* \$ *i*) ∧ (*Inl i* ∈ *FVF* φ ⟶ *sol 0* \$ *i* = *ab* \$ *i*)
and *allSol*:∀ *x*∈{*0..t*}. (*sol has-derivative* (λ*xa. xa* ∗$_R$ *ODE-sem I ODE* (*sol x*))) (*at x within* {*0..t*})
and *mkV*:*sol* ∈ {*0..t*} → {*x. mk-v I ODE* (*ab*, *bb*) *x* ∈ *fml-sem I* φ}
and *x*:*0* ≤ *x*
and *t*:*x* ≤ *t*
from *all* **have** *allT*:⋀*s. s* ≥ *0* ⟹ *s* ≤ *t* ⟹ *mk-v I ODE* (*ab*,*bb*) (*sol s*) ∈ *fml-sem I* φ
using *mkV* **by** *auto*
have *VA*:⋀*x. Vagree* (*mk-v I ODE* (*ab*, *bb*) (*sol x*)) (*mk-v I ODE* (*ab*, *bb*) (χ *i. if* $i \in$ *FVO ODE then sol x* \$ *i else ab* \$ *i*))
(*FVF* φ)
unfolding *Vagree-def*
apply(*auto*)
subgoal for *xa i*
using *mk-v-agree*[*of I ODE* (*ab*,*bb*) *sol xa*]
mk-v-agree[*of I ODE* (*ab*,*bb*) (χ *i. if* $i \in$ *FVO ODE then sol xa* \$ *i else ab* \$ *i*)]
apply(*cases i* ∈ *ODE-vars I ODE*)
using *ode-to-fvo* [*of i I ODE*] **unfolding** *Vagree-def*

apply *auto*
by *fastforce*
subgoal for *xa i*
using *mk-v-agree*[*of I ODE* (*ab*,*bb*) *sol xa*]
mk-v-agree[*of I ODE* (*ab*,*bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol xa* $ *i else ab* $ *i*)]
ODE-vars-lr
using *ode-to-fvo*[*of i I ODE*] **unfolding** *Vagree-def* **apply** *auto*
using *alt-sem-lemma osafe*
subgoal
proof −
assume *a1*: ∀ *i*. *Inr i* ∉ *Inl* ‘ *ODE-vars I ODE* ∧ (*Inr i*::′*c* + ′*c*) ∉ *Inr* ‘ *ODE-vars I ODE* ⟶ *snd* (*mk-v I ODE* (*ab*, *bb*) (*sol xa*)) $ *i* = *bb* $ *i*
assume *a2*: ∀ *i*. *Inr i* ∉ *Inl* ‘ *ODE-vars I ODE* ∧ (*Inr i*::′*c* + ′*c*) ∉ *Inr* ‘ *ODE-vars I ODE* ⟶ *snd* (*mk-v I ODE* (*ab*, *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol xa* $ *i else ab* $ *i*)) $ *i* = *bb* $ *i*
assume *a3*: ∀ *i*. (*Inr i* ∈ *Inl* ‘ *ODE-vars I ODE* ⟶ *snd* (*mk-v I ODE* (*ab*, *bb*) (*sol xa*)) $ *i* = *ODE-sem I ODE* (*sol xa*) $ *i*) ∧ ((*Inr i*::′*c* + ′*c*) ∈ *Inr* ‘ *ODE-vars I ODE* ⟶ *snd* (*mk-v I ODE* (*ab*, *bb*) (*sol xa*)) $ *i* = *ODE-sem I ODE* (*sol xa*) $ *i*)
assume *a4*: ∀ *i*. (*Inr i* ∈ *Inl* ‘ *ODE-vars I ODE* ⟶ *snd* (*mk-v I ODE* (*ab*, *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol xa* $ *i else ab* $ *i*)) $ *i* = *ODE-sem I ODE* (χ *i*. *if* *i* ∈ *FVO ODE then sol xa* $ *i else ab* $ *i*) $ *i*) ∧ ((*Inr i*::′*c* + ′*c*) ∈ *Inr* ‘ *ODE-vars I ODE* ⟶ *snd* (*mk-v I ODE* (*ab*, *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol xa* $ *i else ab* $ *i*)) $ *i* = *ODE-sem I ODE* (χ *i*. *if* *i* ∈ *FVO ODE then sol xa* $ *i else ab* $ *i*) $ *i*)
have *ODE-sem I ODE* (χ *c*. *if* *c* ∈ *FVO ODE then sol xa* $ *c else ab* $ *c*) $ *i* = *ODE-sem I ODE* (*sol xa*) $ *i*
by (*metis* (*no-types*) *alt-sem-lemma osafe*)
then have *Inr i* ∉ *Inl* ‘ *ODE-vars I ODE* ∧ (*Inr i*::′*c* + ′*c*) ∉ *Inr* ‘ *ODE-vars I ODE* ∨ *snd* (*mk-v I ODE* (*ab*, *bb*) (*sol xa*)) $ *i* = *snd* (*mk-v I ODE* (*ab*, *bb*) (χ *c*. *if* *c* ∈ *FVO ODE then sol xa* $ *c else ab* $ *c*)) $ *i*
using *a4 a3* **by** *fastforce*
then show *?thesis*
using *a2 a1* **by** *presburger*
qed
done
done
note *sem* = *IHF*[*OF Iagree-refl*[*of I*]]
have *VA1*:(∀ *i*. *Inl i* ∈ *FVF* φ ⟶
fst (*mk-v I ODE* ((χ *i*. *if* *i* ∈ *FVO ODE then sol 0* $ *i else ab* $ *i*), *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol x* $ *i else ab* $ *i*)) $ *i*
= *fst* (*mk-v I ODE* (*ab*, *bb*) (*sol x*)) $ *i*)
and *VA2*: (∀ *i*. *Inr i* ∈ *FVF* φ ⟶
snd (*mk-v I ODE* ((χ *i*. *if* *i* ∈ *FVO ODE then sol 0* $ *i else ab* $ *i*), *bb*) (χ *i*. *if* *i* ∈ *FVO ODE then sol x* $ *i else ab* $ *i*)) $ *i*
= *snd* (*mk-v I ODE* (*ab*, *bb*) (*sol x*)) $ *i*)
apply(*auto*)
subgoal for *i*

```
                using mk-v-agree[of I ODE ((χ i. if  i ∈ FVO ODE then sol 0 $
i else ab $ i),bb) (χ i. if  i ∈ FVO ODE then sol x $ i else ab $ i)]
                 using mk-v-agree[of I ODE (ab,bb) (sol x)] ODE-vars-lr[of i I
ODE]
                unfolding Vagree-def apply (auto)
                 apply(erule allE[where x=i])+
                 apply(cases  i ∈ FVO ODE)
                  apply(auto)
                  apply(cases  i ∈ FVO ODE)
                  apply(auto)
                  using ODE-vars-lr[of i I ODE] ode-to-fvo[of i I ODE]
                  apply auto
                 using all by meson
               subgoal for i
                using mk-v-agree[of I ODE ((χ i. if  i ∈ FVO ODE then sol 0 $
i else ab $ i),bb) (χ i. if  i ∈ FVO ODE then sol x $ i else ab $ i)]
                 using mk-v-agree[of I ODE (ab,bb) (sol x)] ODE-vars-lr[of i I
ODE]
                unfolding Vagree-def apply (auto)
                apply(erule allE[where x=i])+
                apply(cases  i ∈ FVO ODE)
                 apply(auto)
                 apply(cases  i ∈ FVO ODE)
                  apply(auto)
                 using ODE-vars-lr[of i I ODE] ode-to-fvo[of i I ODE]
                 apply(auto)
                 using alt-sem-lemma osafe
                 by (metis (no-types) alt-sem-lemma osafe)+
               done
             show mk-v I ODE (χ i. if  i ∈ FVO ODE then sol 0 $ i else ab $ i,
bb)
                                 (χ i. if  i ∈ FVO ODE then sol x $ i else ab $ i) ∈
fml-sem I φ
              using mk-v-agree[of I ODE (χ i. if  i ∈ FVO ODE then sol 0 $ i
else ab $ i, bb)
                                     (χ i. if  i ∈ FVO ODE then sol x $ i else ab $ i)]
                mk-v-agree[of I ODE (ab, bb) sol x]
               using sem[of mk-v I ODE (χ i. if  i ∈ FVO ODE then sol 0 $ i
else ab $ i, bb) (χ i. if  i ∈ FVO ODE then sol x $ i else ab $ i)
                           mk-v I ODE (ab, bb) (sol x)]
               VA1 VA2
               allT[of x] allT[of 0]
               unfolding Vagree-def
               apply auto
               using atLeastAtMost-iff mem-Collect-eq mkV t x
               apply(auto)
               using eqP VA sem
               by auto
             qed
```

proof −
fix x i
assume
$assms$:$osafe\ ODE$
$fsafe\ \varphi$
$0 \leq t$
$(aa, ba) = mk\text{-}v\ I\ ODE\ (ab, bb)\ (sol\ t)$
$VSagree\ (sol\ 0)\ ab\ \{x.\ Inl\ x \in BVO\ ODE \vee Inl\ x \in Inl\ `\ FVO\ ODE \vee Inl\ x \in FVF\ \varphi\}$
and $deriv$:$\forall\, x{\in}\{0..t\}.\ (sol\ has\text{-}derivative\ (\lambda xa.\ xa *_R ODE\text{-}sem\ I\ ODE\ (sol\ x)))\ (at\ x\ within\ \{0..t\})$
and sol:$sol \in \{0..t\} \to \{x.\ mk\text{-}v\ I\ ODE\ (ab, bb)\ x \in fml\text{-}sem\ I\ \varphi\}$
and mem:$x \in \{0..t\}$
from $deriv$
have $xDeriv$:$(sol\ has\text{-}derivative\ (\lambda xa.\ xa *_R ODE\text{-}sem\ I\ ODE\ (sol\ x)))\ (at\ x\ within\ \{0..t\})$
using mem **by** $blast$
have $silly1$:$(\lambda x.\ \chi\ i.\ sol\ x\ \$\ i) = sol$
by $(auto\ simp\ add{:}\ vec\text{-}eq\text{-}iff)$
have $silly2$:$(\lambda h.\ \chi\ i.\ h * ODE\text{-}sem\ I\ ODE\ (sol\ x)\ \$\ i) = (\lambda xa.\ xa *_R ODE\text{-}sem\ I\ ODE\ (sol\ x))$
by $(auto\ simp\ add{:}\ vec\text{-}eq\text{-}iff)$
from $xDeriv$ **have**
$xDeriv'$:$((\lambda x.\ \chi\ i.\ sol\ x\ \$\ i)\ has\text{-}derivative\ (\lambda h.\ \chi\ i.\ h * ODE\text{-}sem\ I\ ODE\ (sol\ x)\ \$\ i))\ (at\ x\ within\ \{0..t\})$
using $silly1\ silly2$ **apply** $auto$ **done**
from $xDeriv$ **have** $xDerivs$:$\bigwedge j.\ ((\lambda t.\ sol\ t\ \$\ j)\ has\text{-}derivative\ (\lambda xa.\ (xa *_R ODE\text{-}sem\ I\ ODE\ (sol\ x))\ \$\ j))\ (at\ x\ within\ \{0..t\})$
subgoal for j
using $silly1\ silly2\ has\text{-}derivative\text{-}proj[of\ (\lambda i.\ \lambda t.\ sol\ t\ \$\ i)\ (\lambda\ i.\ \lambda xa.\ (xa *_R ODE\text{-}sem\ I\ ODE\ (sol\ x))\ \$\ i)\ (at\ x\ within\ \{0..t\})\ j]$
apply $auto$
done
done
have $neato$:$\bigwedge \nu.\ \ i \notin FVO\ ODE \Longrightarrow ODE\text{-}sem\ I\ ODE\ \nu\ \$\ i = 0$
proof $(induction\ ODE)$
qed $auto$
show $((\lambda t.\ if\ \ i \in FVO\ ODE\ then\ sol\ t\ \$\ i\ else\ ab\ \$\ i)\ has\text{-}derivative\ (\lambda h.\ h *_R ODE\text{-}sem\ I\ ODE\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ x\ \$\ i\ else\ ab\ \$\ i)\ \$\ i))$
$(at\ x\ within\ \{0..t\})$
using $assms\ sol\ mem$
apply $auto$
apply $(rule\ has\text{-}derivative\text{-}eq\text{-}rhs)$
unfolding $VSagree\text{-}def$ **apply** $auto$
apply$(cases\ \ i \in FVO\ ODE)$
using $xDerivs[of\ i]$ **apply** $auto$
using $alt\text{-}sem\text{-}lemma\ neato[of\ (\chi\ i.\ if\ \ i \in FVO\ ODE\ then\ sol\ x\ \$\ i\ else\ ab\ \$\ i)]$ **apply** $auto$

```
            proof −
              assume a1: ((λt. sol t $ i) has-derivative (λxa. xa ∗ ODE-sem I
ODE (sol x) $ i)) (at x within {0..t})
             have ⋀i r. ODE-sem (i::('a, 'b, 'c) interp) ODE (χ c. if  c ∈ FVO
ODE then sol r $ c else ab $ c) = ODE-sem i ODE (sol r)
               by (metis (no-types) alt-sem-lemma assms(1))
             then show ((λr. sol r $ i) has-derivative (λr. r ∗ ODE-sem I ODE
(χ c. if  c ∈ FVO ODE then sol x $ c else ab $ c) $ i)) (at x within {0..t})
               using a1 by presburger
            qed
           qed
           proof −
            fix aa ba bb sol t
            assume osafe:osafe ODE
              and fsafe:fsafe φ
              and t:0 ≤ t
              and aaba:(aa, ba) = mk-v I ODE (sol 0, bb) (sol t)
               and sol:(sol solves-ode (λa. ODE-sem I ODE)) {0..t} {x. mk-v I
ODE (sol 0, bb) x ∈ fml-sem I φ}
            show∃ sola ta. mk-v I ODE (sol 0, bb) (sol t) = mk-v I ODE (sol 0,
bb) (sola ta) ∧
                        0 ≤ ta ∧
                        (sola solves-ode (λa. ODE-sem I ODE)) {0..ta} {x. mk-v I
ODE (sol 0, bb) x ∈ fml-sem I φ} ∧
                          VSagree (sola 0) (sol 0) {x. Inl x ∈ BVO ODE ∨ Inl x ∈
Inl ‘ FVO ODE ∨ Inl x ∈ FVF φ}
              apply(rule exI[where x=sol])
              apply(rule exI[where x=t])
              using fsafe t aaba sol apply auto
              unfolding VSagree-def by auto
            qed
           done
          done
           show ∀ I J. coincide-hp (EvolveODE ODE P) I J ∧ ode-sem-equiv
(EvolveODE ODE P) I
            proof (rule allI)+
              fix I J::('a,'b,'c) interp
            from equiv[of I]
            have equivI:
         {(ν, mk-v I ODE ν (sol t)) | ν sol t.
            t ≥ 0 ∧
            (sol solves-ode (λ-. ODE-sem I ODE)) {0..t} {x. mk-v I ODE ν x ∈
fml-sem I P} ∧
             VSagree (sol 0) (fst ν) {x | x. Inl x ∈ FVP (EvolveODE ODE P)}}
=
         {(ν, mk-v I ODE ν (sol t)) | ν sol t.
            t ≥ 0 ∧
            (sol solves-ode (λ-. ODE-sem I ODE)) {0..t} {x. mk-v I ODE ν x ∈
fml-sem I P} ∧
```

```
    (sol 0) = (fst ν)}
     unfolding ode-sem-equiv-def using osafe fsafe by blast

   from equiv[of J]
   have equivJ:
  {(ν, mk-v J ODE ν (sol t)) | ν sol t.
   t ≥ 0 ∧
  (sol solves-ode (λ-. ODE-sem J ODE)) {0..t} {x. mk-v J ODE ν x ∈
fml-sem J P} ∧
   VSagree (sol 0) (fst ν) {x | x. Inl x ∈ FVP (EvolveODE ODE P)}}
=
  {(ν, mk-v J ODE ν (sol t)) | ν sol t.
   t ≥ 0 ∧
  (sol solves-ode (λ-. ODE-sem J ODE)) {0..t} {x. mk-v J ODE ν x ∈
fml-sem J P} ∧
   (sol 0) = (fst ν)}
     unfolding ode-sem-equiv-def using osafe fsafe by blast
   from equivI
   have alt-ode-semI:prog-sem I (EvolveODE ODE P) =
     {(ν, mk-v I ODE ν (sol t)) | ν sol t.
   t ≥ 0 ∧
   (sol solves-ode (λ-. ODE-sem I ODE)) {0..t} {x. mk-v I ODE ν x ∈
fml-sem I P} ∧
    VSagree (sol 0) (fst ν) {x | x. Inl x ∈ FVP (EvolveODE ODE P)}}
by auto

   from equivJ
   have alt-ode-semJ:prog-sem J (EvolveODE ODE P) =
     {(ν, mk-v J ODE ν (sol t)) | ν sol t.
   t ≥ 0 ∧
  (sol solves-ode (λ-. ODE-sem J ODE)) {0..t} {x. mk-v J ODE ν x ∈
fml-sem J P} ∧
   VSagree (sol 0) (fst ν) {x | x. Inl x ∈ FVP (EvolveODE ODE P)}}
by auto

   have co-hp:coincide-hp (EvolveODE ODE P) I J
    apply(unfold coincide-hp-def)
      apply (auto simp del: prog-sem.simps(8) simp add: alt-ode-semI
alt-ode-semJ)
    proof −
   fix a b aa ba ab bb V sol t
   from IH have IHF:∀ a b aa ba . Iagree I J (SIGF P) ⟶ Vagree (a,
b) (aa, ba) (FVF P) ⟶ ((a, b) ∈ fml-sem I P) = ((aa, ba) ∈ fml-sem J P)
     unfolding coincide-fml-def by blast
    assume IA:Iagree I J (SIGF P ∪ {Inl x |x. Inl x ∈ SIGO ODE} ∪
{Inr (Inr x) |x. Inr x ∈ SIGO ODE})
   and VA:Vagree (a, b) (aa, ba) V
   and OVsub:BVO ODE ⊆ V
   and Osub:Inl ‘ FVO ODE ⊆ V
```

and *Fsub*:*FVF P* ⊆ *V*
and *veq*:(*ab*, *bb*) = *mk-v I ODE* (*a*, *b*) (*sol t*)
and *t*:*0* ≤ *t*
and *sol*:(*sol solves-ode* (λ*a*. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* (*a*, *b*) *x* ∈ *fml-sem I P*}
and *VSA*:*VSagree* (*sol 0*) *a* {*uu*. *Inl uu* ∈ *BVO ODE* ∨ *Inl uu* ∈ *Inl* ‘ *FVO ODE* ∨ *Inl uu* ∈ *FVF P*}
have *semBVsub*:(*semBV I ODE*) ⊆ *BVO ODE*
by (*induction ODE*, *auto*)
then have *OVsub''*:(*semBV I ODE*) ⊆ *V* **using** *OVsub* **by** *auto*
have *MBVBVsub*:(*Inl* ‘ *ODE-dom ODE* ∪ *Inr* ‘ *ODE-dom ODE*) ⊆ *BVO ODE*
apply(*induction ODE*)
by *auto*
from *OVsub* **and** *MBVBVsub* **have** *OVsub'*:(*Inl* ‘ *ODE-dom ODE* ∪ *Inr* ‘ *ODE-dom ODE*) ⊆ *V*
by *auto*
from *sol*
have *solSem*:⋀*x*. *0* ≤ *x* ⟹ *x* ≤ *t* ⟹ *mk-v I ODE* (*a*, *b*) (*sol x*) ∈ *fml-sem I P*
and *solDeriv*:⋀*x*. *0* ≤ *x* ⟹ *x* ≤ *t* ⟹ (*sol has-vector-derivative ODE-sem I ODE* (*sol x*)) (*at x within* {*0*..*t*})
unfolding *solves-ode-def has-vderiv-on-def* **by** *auto*
have *SIGFsub*:(*SIGF P*) ⊆ (*SIGF P* ∪ {*Inl x* |*x*. *Inl x* ∈ *SIGO ODE*} ∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE*}) **by** *auto*
from *IA* **have** *IAP*:*Iagree I J* (*SIGF P*)
using *Iagree-sub*[*OF SIGFsub*] **by** *auto*
from *IHF* **have** *IH'*:
∀ *a b aa ba*. *Vagree* (*a*, *b*) (*aa*, *ba*) (*FVF P*) ⟶ ((*a*, *b*) ∈ *fml-sem I P*) = ((*aa*, *ba*) ∈ *fml-sem J P*)
using *IAP* **by** *blast*
from *VA*
have *VAOV*:*Vagree* (*a*,*b*) (*aa*,*ba*) (*BVO ODE*)
using *agree-sub*[*OF OVsub*] **by** *auto*
have *ag*:⋀*s*. *Vagree* (*mk-v I ODE* (*a*, *b*) (*sol s*)) (*a*, *b*) (− *semBV I ODE*)
⋀*s*. *Vagree* (*mk-v I ODE* (*a*, *b*) (*sol s*)) (*mk-xode I ODE* (*sol s*)) (*semBV I ODE*)
⋀*s*. *Vagree* (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) (*aa*, *ba*) (− *semBV J ODE*)
⋀*s*. *Vagree* (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) (*mk-xode J ODE* (*sol s*)) (*semBV J ODE*)
subgoal for *s* **using** *mk-v-agree*[*of I ODE* (*a*,*b*) *sol s*] **by** *auto*
subgoal for *s* **using** *mk-v-agree*[*of I ODE* (*a*,*b*) *sol s*] **by** *auto*
subgoal for *s* **using** *mk-v-agree*[*of J ODE* (*aa*,*ba*) *sol s*] **by** *auto*
subgoal for *s* **using** *mk-v-agree*[*of J ODE* (*aa*,*ba*) *sol s*] **by** *auto*
done
have *sem-sub-BVO*:⋀*I*. *semBV I ODE* ⊆ *BVO ODE*
subgoal for *I*

apply(*induction ODE*)
by *auto*
done
have *MBV-sub-sem*:$\bigwedge I$. (*Inl* ‘ *ODE-dom ODE* ∪ *Inr* ‘ *ODE-dom ODE*) ⊆ *semBV I ODE*
subgoal for *I* **by** (*induction ODE*, *auto*) **done**
have *ag-BVO*:
$\bigwedge s$. *Vagree* (*mk-v I ODE* (*a*, *b*) (*sol s*)) (*a*, *b*) (− *BVO ODE*)
$\bigwedge s$. *Vagree* (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) (*aa*, *ba*) (− *BVO ODE*)
using *ag*(*1*) *ag*(*3*) *sem-sub-BVO*[*of I*] *sem-sub-BVO*[*of J*] *agree-sub*
by *blast*+
have *ag-semBV*:
$\bigwedge s$. *Vagree* (*mk-v I ODE* (*a*, *b*) (*sol s*)) (*mk-xode I ODE* (*sol s*)) (*Inl* ‘ *ODE-dom ODE* ∪ *Inr* ‘ *ODE-dom ODE*)
$\bigwedge s$. *Vagree* (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) (*mk-xode J ODE* (*sol s*)) (*Inl* ‘ *ODE-dom ODE* ∪ *Inr* ‘ *ODE-dom ODE*)
using *ag*(*2*) *ag*(*4*) *MBV-sub-sem*[*of I*] *MBV-sub-sem*[*of J*]
by (*simp add*: *agree-sub*)+
have *IOsub*:({*Inl x* |*x*. *Inl x* ∈ *SIGO ODE*} ∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE*}) ⊆ (*SIGF P* ∪ {*Inl x* |*x*. *Inl x* ∈ *SIGO ODE*} ∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE*})
by *auto*
from *IA*
have *IAO*:*Iagree I J* ({*Inl x* |*x*. *Inl x* ∈ *SIGO ODE*} ∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE*})
using *Iagree-sub*[*OF IOsub*] **by** *auto*
have *IOsub′*:({*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE*}) ⊆ ({*Inl x* |*x*. *Inl x* ∈ *SIGO ODE*} ∪ {*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE*})
by *auto*
from *IAO*
have *IAO′*:*Iagree I J* ({*Inr* (*Inr x*) |*x*. *Inr x* ∈ *SIGO ODE*})
using *Iagree-sub*[*OF IOsub′*] **by** *auto*
have *VAsol*:$\bigwedge s\ \nu'$. *Vagree* ((*sol s*), *ν′*) ((*sol s*), *ν′*) (*Inl* ‘*FVO ODE*) **unfolding** *Vagree-def* **by** *auto*
have *Osem*:$\bigwedge s$. $0 \le s \Longrightarrow s \le t \Longrightarrow$ *ODE-sem I ODE* (*sol s*) = *ODE-sem J ODE* (*sol s*)
subgoal for *s*
using *coincidence-ode*[*OF osafe VAsol*[*of s*] *IAO*] **by** *auto*
done
from *Osem*
have *Oag*:$\bigwedge s$. $0 \le s \Longrightarrow s \le t \Longrightarrow$ *VSagree* (*ODE-sem I ODE* (*sol s*)) (*ODE-sem J ODE* (*sol s*)) {*x*. *Inr x* ∈ *BVO ODE*}
unfolding *VSagree-def* **by** *auto*
from *Osem*
have *Oagsem*:$\bigwedge s$. $0 \le s \Longrightarrow s \le t \Longrightarrow$ *VSagree* (*ODE-sem I ODE* (*sol s*)) (*ODE-sem J ODE* (*sol s*)) {*x*. *Inr x* ∈ (*semBV I ODE*)}
unfolding *VSagree-def* **by** *auto*
from *Osem*
have *halp*:$\bigwedge s$. $0 \le s \Longrightarrow s \le t \Longrightarrow$ *Vagree* (*mk-xode I ODE* (*sol s*))

(mk-xode J ODE (sol s)) (semBV I ODE)
apply(*auto*)
using *Oag* **unfolding** *Vagree-def VSagree-def* **by** *blast*
then have *halpp*:$\bigwedge s.\ 0 \leq s \Longrightarrow s \leq t \Longrightarrow$ *Vagree (sol s, ODE-sem I ODE (sol s)) (sol s, ODE-sem J ODE (sol s)) (semBV I ODE)*
by *auto*
have *eqV*:$V = ((semBV\ I\ ODE)) \cup (V \cap (-(semBV\ I\ ODE)))$ **using** *OVsub''* **by** *auto*
have *neat*:$\bigwedge ODE.$ *Iagree I J* $(\{Inr\ (Inr\ x)\ |x.\ Inr\ x \in SIGO\ ODE\}) \Longrightarrow semBV\ I\ ODE = semBV\ J\ ODE$
subgoal for *ODE*
proof (*induction ODE*)
case (*OVar x*)
then show *?case* **unfolding** *Iagree-def* **by** *auto*
next
case (*OSing x1a x2*)
then show *?case* **by** *auto*
next
case (*OProd ODE1 ODE2*)
assume *IH1*:*Iagree I J* $\{Inr\ (Inr\ x)\ |x.\ Inr\ x \in SIGO\ ODE1\} \Longrightarrow semBV\ I\ ODE1 = semBV\ J\ ODE1$
assume *IH2*:*Iagree I J* $\{Inr\ (Inr\ x)\ |x.\ Inr\ x \in SIGO\ ODE2\} \Longrightarrow semBV\ I\ ODE2 = semBV\ J\ ODE2$
assume *agree*:*Iagree I J* $\{Inr\ (Inr\ x)\ |x.\ Inr\ x \in SIGO\ (OProd\ ODE1\ ODE2)\}$
from *agree* **have** *agree1*:*Iagree I J* $\{Inr\ (Inr\ x)\ |x.\ Inr\ x \in SIGO\ (\ ODE1\ )\}$ **and** *agree2*:*Iagree I J* $\{Inr\ (Inr\ x)\ |x.\ Inr\ x \in SIGO\ (\ ODE2)\}$
unfolding *Iagree-def* **by** *auto*
show *?case* **using** *IH1*[*OF agree1*] *IH2*[*OF agree2*] **by** *auto*
qed
done
note *semBVeq* = *neat*[*OF IAO'*]
then have *halpp'*:$\bigwedge s.\ 0 \leq s \Longrightarrow s \leq t \Longrightarrow$ *Vagree (mk-v I ODE (a, b) (sol s)) (mk-v J ODE (aa, ba) (sol s)) (semBV I ODE)*
subgoal for *s* **using** *ag*[*of s*] *ag-semBV*[*of s*] *Oagsem agree-trans semBVeq*
unfolding *Vagree-def* **by** (*auto simp add*: *semBVeq Osem*)
done
have *VAbar*:$\bigwedge s.\ 0 \leq s \Longrightarrow s \leq t \Longrightarrow$ *Vagree (mk-v I ODE (a, b) (sol s)) (mk-v J ODE (aa, ba) (sol s)) (V ∩ (−(semBV I ODE)))*
subgoal for *s*
apply(*unfold Vagree-def*)
apply(*rule conjI* | *rule allI*)+
subgoal for *i*
apply *auto*
using *VA ag*[*of s*] *semBVeq* **unfolding** *Vagree-def* **apply** *auto*
by (*metis Un-iff*)

apply(*rule allI*)+

subgoal for *i*
using *VA ag*[*of s*] *semBVeq* **unfolding** *Vagree-def* **by** *auto*
done
done
have *VAfoo*:⋀*s*. *0* ≤ *s* ⟹ *s* ≤ *t* ⟹ *Vagree* (*mk-v I ODE* (*a*, *b*) (*sol s*)) (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) *V*
using *agree-union*[*OF halpp*′ *VAbar*] *eqV* **by** *auto*
have *duhSub*:*FVF P* ⊆ *UNIV* **by** *auto*
from *VAfoo*
have *VA*′*foo*:⋀*s*. *0* ≤ *s* ⟹ *s* ≤ *t* ⟹ *Vagree* (*mk-v I ODE* (*a*, *b*) (*sol s*)) (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) *V*
using *agree-sub*[*OF duhSub*] **by** *auto*
then have *VA*″*foo*:⋀*s*. *0* ≤ *s* ⟹ *s* ≤ *t* ⟹ *Vagree* (*mk-v I ODE* (*a*, *b*) (*sol s*)) (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) (*FVF P*)
using *agree-sub*[*OF Fsub*] **by** *auto*
from *VA*″*foo IH*′
have *fmlSem*:⋀*s*. *0* ≤ *s* ⟹ *s* ≤ *t* ⟹ (*mk-v I ODE* (*a*, *b*) (*sol s*)) ∈ *fml-sem I P* ⟷ (*mk-v J ODE* (*aa*, *ba*) (*sol s*)) ∈ *fml-sem J P*
using *IAP coincide-fml-def hpsafe-Evolve.IH* **by** *blast*
from *VA*
have *VAO*:*Vagree* (*a*, *b*) (*aa*, *ba*) (*Inl* ‘*FVO ODE*)
using *agree-sub*[*OF Osub*] **by** *auto*
have *sol*′:(*sol solves-ode* (λ-. *ODE-sem J ODE*)) {*0*..*t*} {*x*. *mk-v J ODE* (*aa*, *ba*) *x* ∈ *fml-sem J P*}
apply(*auto simp add*: *solves-ode-def has-vderiv-on-def*)
subgoal for *s*
using *solDeriv*[*of s*] *Osem*[*of s*] **by** *auto*
subgoal for *s*
using *solSem*[*of s*] *fmlSem*[*of s*] **by** *auto*
done
have *VSA*′:*VSagree* (*sol 0*) *aa* {*uu*. *Inl uu* ∈ *BVO ODE* ∨ *Inl uu* ∈ *Inl* ‘*FVO ODE* ∨ *Inl uu* ∈ *FVF P*}
using *VSA VA OVsub* **unfolding** *VSagree-def Vagree-def*
apply *auto*
using *Osub* **apply** *blast*
using *Fsub* **by** *blast*
show
∃ *ab bb*. (∃ *sol t*. (*ab*, *bb*) = *mk-v J ODE* (*aa*, *ba*) (*sol t*) ∧
0 ≤ *t* ∧
(*sol solves-ode* (λ*a*. *ODE-sem J ODE*)) {*0*..*t*} {*x*. *mk-v J ODE* (*aa*, *ba*) *x* ∈ *fml-sem J P*} ∧
VSagree (*sol 0*) *aa* {*uu*. *Inl uu* ∈ *BVO ODE* ∨ *Inl uu* ∈ *Inl* ‘*FVO ODE* ∨ *Inl uu* ∈ *FVF P*}) ∧
Vagree (*mk-v I ODE* (*a*, *b*) (*sol t*)) (*ab*, *bb*) (*Inl* ‘ *ODE-dom ODE* ∪ *Inr* ‘ *ODE-dom ODE* ∪ *V*)
apply(*rule exI*[**where** *x*=*fst* (*mk-v J ODE* (*aa*, *ba*) (*sol t*))])
apply(*rule exI*[**where** *x*=*snd* (*mk-v J ODE* (*aa*, *ba*) (*sol t*))])
apply(*rule conjI*)
subgoal

apply(*rule exI*[**where** *x*=*sol*])
apply(*rule exI*[**where** *x*=*t*])
apply(*rule conjI*)
subgoal
apply(*auto*)
done
subgoal
apply(*rule conjI*)
subgoal by (*rule t*)
subgoal
apply(*rule conjI*)
subgoal by (*rule sol'*)
subgoal by (*rule VSA'*)
done
done
done
apply(*auto*)
using *mk-v-agree*[*of I ODE* (*a*,*b*) (*sol t*)]
mk-v-agree[*of J ODE* (*aa*,*ba*) (*sol t*)]
using *agree-refl t VA'foo*
OVsub' Un-absorb1 **by** (*auto simp add*: *OVsub' Un-absorb1*)
qed
show *coincide-hp* (*EvolveODE ODE P*) *I J* ∧ *ode-sem-equiv* (*EvolveODE ODE P*) *I* **using** *co-hp equiv*[*of I*] **by** *auto*
qed
qed
next
case (*hpsafe-Choice a b*)
then show *?case*
proof (*auto simp only*: *coincide-hp'-def coincide-hp-def*)
fix *I J*::('*a*,'*b*,'*c*) *interp* **and** *ν1 ν1' ν2 ν2' μ μ' V*
assume *safe*:*hpsafe a*
hpsafe b
and *IH1*:
∀ *I J*. (∀ *ν ν' μ V*.
Iagree I J (*SIGP a*) ⟶
Vagree ν ν' V ⟶ *FVP a* ⊆ *V* ⟶ (*ν*, *μ*) ∈ *prog-sem I a* ⟶ (∃ *μ'*. (*ν'*, *μ'*) ∈ *prog-sem J a* ∧ *Vagree μ μ'* (*MBV a* ∪ *V*)))
∧ *ode-sem-equiv a I*
and *IH2*:∀ *I J*. (∀ *ν ν' μ V*.
Iagree I J (*SIGP b*) ⟶
Vagree ν ν' V ⟶ *FVP b* ⊆ *V* ⟶ (*ν*, *μ*) ∈ *prog-sem I b* ⟶ (∃ *μ'*. (*ν'*, *μ'*) ∈ *prog-sem J b* ∧ *Vagree μ μ'* (*MBV b* ∪ *V*)))
∧ *ode-sem-equiv b I*
and *IA*:*Iagree I J* (*SIGP* (*a* ∪∪ *b*))
and *VA*:*Vagree* (*ν1*, *ν1'*) (*ν2*, *ν2'*) *V*
and *sub*:*FVP* (*a* ∪∪ *b*) ⊆ *V*
and *sem*:((*ν1*, *ν1'*), (*μ*, *μ'*)) ∈ *prog-sem I* (*a* ∪∪ *b*)
hence *eitherSem*:((*ν1*, *ν1'*), (*μ*, *μ'*)) ∈ *prog-sem I a* ∨ ((*ν1*, *ν1'*), (*μ*, *μ'*)) ∈

prog-sem I b
by *auto*
have *Ssub*:(*SIGP a*) ⊆ *SIGP* (*a* ∪∪ *b*) (*SIGP b*) ⊆ *SIGP* (*a* ∪∪ *b*)
unfolding *SIGP.simps* **by** *auto*
have *IA1*:*Iagree I J* (*SIGP a*) **and** *IA2*:*Iagree I J* (*SIGP b*)
using *IA Iagree-sub*[*OF Ssub*(*1*)] *Iagree-sub*[*OF Ssub*(*2*)] **by** *auto*
from *sub* **have** *sub1*:*FVP a* ⊆ *V* **and** *sub2*:*FVP b* ⊆ *V* **by** *auto*
then
show ∃ μ''. ((ν*2*, ν*2*′), μ'') ∈ *prog-sem J* (*a* ∪∪ *b*) ∧ *Vagree* (μ, μ′) μ'' (*MBV* (*a* ∪∪ *b*) ∪ *V*)
proof (*cases* ((ν*1*, ν*1*′), (μ, μ′)) ∈ *prog-sem I a*)
case *True*
then obtain μ'' **where** *prog-sem*:((ν*2*,ν*2*′), μ'') ∈ *prog-sem J a* **and** *agree*:*Vagree* (μ, μ′) μ'' (*MBV a* ∪ *V*)
using *IH1 VA sub1 IA1* **by** *blast*
from *agree* **have** *agree′*:*Vagree* (μ, μ′) μ'' (*MBV* (*a* ∪∪ *b*) ∪ *V*)
unfolding *Vagree-def MBV.simps* **by** *auto*
from *prog-sem* **have** *prog-sem′*:((ν*2*,ν*2*′), μ'') ∈ *prog-sem J* (*a* ∪∪ *b*)
unfolding *prog-sem.simps* **by** *blast*
from *agree′* **and** *prog-sem′* **show** *?thesis* **by** *blast*
next
case *False*
then have *sem2*:((ν*1*, ν*1*′), (μ, μ′)) ∈ *prog-sem I b* **using** *eitherSem* **by** *blast*
then obtain μ'' **where** *prog-sem*:((ν*2*,ν*2*′), μ'') ∈ *prog-sem J b* **and** *agree*:*Vagree* (μ, μ′) μ'' (*MBV b* ∪ *V*)
using *IH2 VA sub2 IA2* **by** *blast*
from *agree* **have** *agree′*:*Vagree* (μ, μ′) μ'' (*MBV* (*a* ∪∪ *b*) ∪ *V*)
unfolding *Vagree-def MBV.simps* **by** *auto*
from *prog-sem* **have** *prog-sem′*:((ν*2*,ν*2*′), μ'') ∈ *prog-sem J* (*a* ∪∪ *b*)
unfolding *prog-sem.simps* **by** *blast*
from *agree′* **and** *prog-sem′* **show** *?thesis* **by** *blast*
qed
next
fix *I*
assume *IHs*:
∀ *I J*. (∀ ν ν′ μ *V*.
Iagree I J (*SIGP a*) ⟶
Vagree ν ν′ *V* ⟶ *FVP a* ⊆ *V* ⟶ (ν, μ) ∈ *prog-sem I a* ⟶ (∃ μ′. (ν′, μ′) ∈ *prog-sem J a* ∧ *Vagree* μ μ′ (*MBV a* ∪ *V*))) ∧
ode-sem-equiv a I
∀ *I J*. (∀ ν ν′ μ *V*.
Iagree I J (*SIGP b*) ⟶
Vagree ν ν′ *V* ⟶ *FVP b* ⊆ *V* ⟶ (ν, μ) ∈ *prog-sem I b* ⟶ (∃ μ′. (ν′, μ′) ∈ *prog-sem J b* ∧ *Vagree* μ μ′ (*MBV b* ∪ *V*))) ∧
ode-sem-equiv b I
show *ode-sem-equiv* (*a* ∪∪ *b*) *I*
unfolding *ode-sem-equiv-def* **by** *auto*
qed
next

case (*hpsafe-Sequence a b*) **then show** *?case*
apply (*unfold coincide-hp'-def coincide-hp-def*)
apply (*rule allI*)+
apply (*rule conjI*)
prefer *2* **subgoal unfolding** *ode-sem-equiv-def* **by** *auto*
apply(*unfold prog-sem.simps SIGP.simps FVP.simps*)
apply(*rule allI*)+
apply(*auto*)
subgoal for *I J ν2 ν2' V ν1 ν1' μ μ' ω ω'*
proof −
assume *safe*:*hpsafe a hpsafe b*
assume (∀ *I*. ((∀ *J*. *Iagree I J* (*SIGP a*) ⟶ (∀ *aa b ab ba ac bb V*.
Vagree (*aa*, *b*) (*ab*, *ba*) *V* ⟶
FVP a ⊆ *V* ⟶ ((*aa*, *b*), *ac*, *bb*) ∈ *prog-sem I a* ⟶ (∃ *aa b*. ((*ab*, *ba*), *aa*, *b*) ∈ *prog-sem J a* ∧ *Vagree* (*ac*, *bb*) (*aa*, *b*) (*MBV a* ∪ *V*)))))
∧ *ode-sem-equiv a I*)
hence *IH1'*:⋀*aa b ab ba ac bb V*.
Iagree I J (*SIGP a*) ⟹
Vagree (*aa*, *b*) (*ab*, *ba*) *V* ⟹
FVP a ⊆ *V* ⟹ ((*aa*, *b*), *ac*, *bb*) ∈ *prog-sem I a* ⟹ (∃ *aa b*. ((*ab*, *ba*), *aa*, *b*) ∈ *prog-sem J a* ∧ *Vagree* (*ac*, *bb*) (*aa*, *b*) (*MBV a* ∪ *V*))
by *auto*
note *IH1* = *IH1'*[*of ν1 ν1' ν2 ν2' V μ μ'*]
assume *IH2''*:
∀ *I*. (∀ *J*. *Iagree I J* (*SIGP b*) ⟶ (∀ *a ba aa bb ab bc V*.
Vagree (*a*, *ba*) (*aa*, *bb*) *V* ⟶
FVP b ⊆ *V* ⟶ ((*a*, *ba*), *ab*, *bc*) ∈ *prog-sem I b* ⟶ (∃ *a ba*. ((*aa*, *bb*), *a*, *ba*) ∈ *prog-sem J b* ∧ *Vagree* (*ab*, *bc*) (*a*, *ba*) (*MBV b* ∪ *V*))))
∧ *ode-sem-equiv b I*
assume *IAab*:*Iagree I J* (*SIGP a* ∪ *SIGP b*)
have *IAsubs*:*SIGP a* ⊆ (*SIGP a* ∪ *SIGP b*) *SIGP b* ⊆ (*SIGP a* ∪ *SIGP b*) **by** *auto*
from *IAab* **have** *IA*:*Iagree I J* (*SIGP a*) *Iagree I J* (*SIGP b*) **using** *Iagree-sub*[*OF IAsubs*(*1*)] *Iagree-sub*[*OF IAsubs*(*2*)] **by** *auto*
from *IH2''* **have** *IH2'*:⋀*a ba aa bb ab bc V* .
Iagree I J (*SIGP b*) ⟹
Vagree (*a*, *ba*) (*aa*, *bb*) *V* ⟹
FVP b ⊆ *V* ⟹ ((*a*, *ba*), *ab*, *bc*) ∈ *prog-sem I b* ⟹ (∃ *a ba*. ((*aa*, *bb*), *a*, *ba*) ∈ *prog-sem J b* ∧ *Vagree* (*ab*, *bc*) (*a*, *ba*) (*MBV b* ∪ *V*))
using *IA* **by** *auto*
assume *VA*:*Vagree* (*ν1*, *ν1'*) (*ν2*, *ν2'*) *V*
assume *sub*:*FVP a* ⊆ *V FVP b* − *MBV a* ⊆ *V*
hence *sub'*:*FVP a* ⊆ *V* **by** *auto*
assume *sem*:((*ν1*, *ν1'*), (*μ*, *μ'*)) ∈ *prog-sem I a*
((*μ*, *μ'*), (*ω*, *ω'*)) ∈ *prog-sem I b*
obtain *ω1 ω1'* **where** *sem1*:((*ν2*, *ν2'*), (*ω1*, *ω1'*)) ∈ *prog-sem J a* **and** *VA1*:*Vagree* (*μ*, *μ'*) (*ω1*, *ω1'*) (*MBV a* ∪ *V*)
using *IH1*[*OF IA*(*1*) *VA sub' sem*(*1*)] **by** *auto*
note *IH2* = *IH2'*[*of μ μ' ω1 ω1' MBV a* ∪ *V ω ω'*]

have *sub2*:*FVP b* ⊆ *MBV a* ∪ *V* **using** *sub* **by** *auto*
obtain *ω2 ω2′* **where** *sem2*:((*ω1*, *ω1′*), (*ω2*, *ω2′*)) ∈ *prog-sem J b* **and** *VA2*:*Vagree* (*ω*, *ω′*) (*ω2*, *ω2′*) (*MBV b* ∪ (*MBV a* ∪ *V*))
using *IH2*[*OF IA*(*2*) *VA1 sub2 sem*(*2*)] **by** *auto*
show ∃ *ab bb*. ((*ν2*, *ν2′*), (*ab*, *bb*)) ∈ *prog-sem J a O prog-sem J b* ∧ *Vagree* (*ω*, *ω′*) (*ab*, *bb*) (*MBV a* ∪ *MBV b* ∪ *V*)
using *sem1 sem2 VA1 VA2*
by (*metis* (*no-types*, *lifting*) *Un-assoc Un-left-commute relcomp.relcompI*)
qed
done
next
case (*hpsafe-Loop a*) **then show** *?case*
apply(*unfold coincide-hp′-def coincide-hp-def*)
apply(*rule allI*)+
apply(*rule conjI*)
prefer *2* **subgoal unfolding** *ode-sem-equiv-def* **by** *auto*
apply(*rule allI* | *rule impI*)+
apply(*unfold prog-sem.simps FVP.simps MBV.simps SIGP.simps*)
subgoal for *I J ν ν′ μ V*
proof −
assume *safe*:*hpsafe a*
assume *IH*:(∀ *I J*. (∀ *ν ν′ μ V*.
Iagree I J (*SIGP a*) ⟶
Vagree ν ν′ V ⟶ *FVP a* ⊆ *V* ⟶ (*ν*, *μ*) ∈ *prog-sem I a* ⟶ (∃ *μ′*. (*ν′*, *μ′*) ∈ *prog-sem J a* ∧ *Vagree μ μ′* (*MBV a* ∪ *V*)))
∧ *ode-sem-equiv a I*)
assume *agree*:*Iagree I J* (*SIGP a*)
assume *VA*:*Vagree ν ν′ V*
assume *sub*:*FVP a* ⊆ *V*
have (*ν*, *μ*) ∈ (*prog-sem I a*)* ⟹ (⋀*ν′*. *Vagree ν ν′ V* ⟹ ∃ *μ′*. (*ν′*, *μ′*) ∈ (*prog-sem J a*)* ∧ *Vagree μ μ′* ({} ∪ *V*))
apply(*induction rule*: *converse-rtrancl-induct*)
apply(*auto*)
subgoal for *ω ω′ s s′ v v′*
proof −
assume *sem1*:((*ω*, *ω′*), (*s*, *s′*)) ∈ *prog-sem I a*
and *sem2*:((*s*, *s′*), *μ*) ∈ (*prog-sem I a*)*
and *IH2*:⋀*v v′*. (*Vagree* (*s*, *s′*) (*v*,*v′*) *V* ⟹ ∃ *ab ba*. ((*v*,*v′*), (*ab*, *ba*)) ∈ (*prog-sem J a*)* ∧ *Vagree μ* (*ab*, *ba*) *V*)
and *VA*:*Vagree* (*ω*, *ω′*) (*v*,*v′*) *V*
obtain *s″* **where** *sem″*:((*v*, *v′*), *s″*) ∈ *prog-sem J a* **and** *VA″*:*Vagree* (*s*,*s′*) *s″* (*MBV a* ∪ *V*)
using *IH agree VA sub sem1 agree-refl* **by** *blast*
then obtain *s′1* **and** *s′2* **where** *sem″*:((*v*, *v′*), (*s′1*, *s′2*)) ∈ *prog-sem J a* **and** *VA″*:*Vagree* (*s*,*s′*) (*s′1*, *s′2*) (*MBV a* ∪ *V*)
using *IH agree VA sub sem1 agree-refl* **by** (*cases s″*, *blast*)
from *VA″* **have** *VA″V*:*Vagree* (*s*,*s′*) (*s′1*, *s′2*) *V*
using *agree-sub* **by** *blast*
note *IH2′* = *IH2*[*of s′1 s′2*]

note *IH2''* = *IH2'*[*OF VA''V*]
then obtain *ab* **and** *ba* **where** *sem'''*:((*s'1*, *s'2*), (*ab*, *ba*)) ∈ (*prog-sem J a*)* **and** *VA'''*:*Vagree μ* (*ab*, *ba*) *V*
using *IH2''* **by** *auto*
from *sem''* *sem'''* **have** *sem*:((*v*, *v'*), (*ab*, *ba*)) ∈ (*prog-sem J a*)* **by** *auto*
show ∃ *μ'1 μ'2*. ((*v*, *v'*), (*μ'1*, *μ'2*)) ∈ (*prog-sem J a*)* ∧ *Vagree μ* (*μ'1*, *μ'2*) *V*
using *sem VA'''* **by** *blast*
qed
done
then show (*ν*, *μ*) ∈ (*prog-sem I a*)* ⟹ *Vagree ν ν' V* ⟹ ∃ *μ'*. (*ν'*, *μ'*) ∈ (*prog-sem J a*)* ∧ *Vagree μ μ'* ({} ∪ *V*)
by *auto*
qed
done
next
case (*fsafe-Geq t1 t2*)
then have *safe*:*dsafe t1 dsafe t2* **by** *auto*
have *almost*:⋀*ν ν'*. ⋀ *I J* :: ('*a*, '*b*, '*c*) *interp*. *Iagree I J* (*SIGF* (*Geq t1 t2*)) ⟹ *Vagree ν ν'* (*FVF* (*Geq t1 t2*)) ⟹ (*ν* ∈ *fml-sem I* (*Geq t1 t2*)) = (*ν'* ∈ *fml-sem J* (*Geq t1 t2*))
proof −
fix *ν ν'* **and** *I J* :: ('*a*, '*b*, '*c*) *interp*
assume *IA*:*Iagree I J* (*SIGF* (*Geq t1 t2*))
hence *IAs*:*Iagree I J* {*Inl x* | *x*. *x* ∈ (*SIGT t1*)}
Iagree I J {*Inl x* | *x*. *x* ∈ (*SIGT t2*)}
unfolding *SIGF.simps Iagree-def* **by** *auto*
assume *VA*:*Vagree ν ν'* (*FVF* (*Geq t1 t2*))
hence *VAs*:*Vagree ν ν'* (*FVT t1*) *Vagree ν ν'* (*FVT t2*)
unfolding *FVF.simps Vagree-def* **by** *auto*
have *sem1*:*dterm-sem I t1 ν* = *dterm-sem J t1 ν'*
by (*auto simp add*: *coincidence-dterm'*[*OF safe*(*1*) *VAs*(*1*) *IAs*(*1*)])
have *sem2*:*dterm-sem I t2 ν* = *dterm-sem J t2 ν'*
by (*auto simp add*: *coincidence-dterm'*[*OF safe*(*2*) *VAs*(*2*) *IAs*(*2*)])
show (*ν* ∈ *fml-sem I* (*Geq t1 t2*)) = (*ν'* ∈ *fml-sem J* (*Geq t1 t2*))
by (*simp add*: *sem1 sem2*)
qed
show *?case* **using** *almost* **unfolding** *coincide-fml-def* **by** *blast*
next
case (*fsafe-Prop args p*)
then have *safes*:⋀*arg*. *arg* ∈ *range args* ⟹ *dsafe arg* **using** *dfree-is-dsafe* **by** *auto*
have *almost*:⋀*ν ν'*. ⋀ *I J*::('*a*, '*b*, '*c*) *interp*. *Iagree I J* (*SIGF* (*Prop p args*)) ⟹ *Vagree ν ν'* (*FVF* (*Prop p args*)) ⟹ (*ν* ∈ *fml-sem I* (*Prop p args*)) = (*ν'* ∈ *fml-sem J* (*Prop p args*))
proof −
fix *ν ν'* **and** *I J* :: ('*a*, '*b*, '*c*) *interp*
assume *IA*:*Iagree I J* (*SIGF* (*Prop p args*))
have *subs*:⋀*i*. {*Inl x* | *x*. *x* ∈ *SIGT* (*args i*)} ⊆ (*SIGF* (*Prop p args*))

```
    by auto
  have IAs:⋀i. Iagree I J {Inl x | x. x ∈ SIGT (args i)}
    using IA apply(unfold SIGF.simps)
    subgoal for i
      using Iagree-sub[OF subs[of i]] by auto
    done
  have mem:Inr (Inr p) ∈ {Inr (Inr p)} ∪ {Inl x |x. x ∈ (⋃ i. SIGT (args i))}
    by auto
  from IA have pSame:Predicates I p = Predicates J p
    by (auto simp add: Iagree-Pred IA mem)
  assume VA:Vagree ν ν' (FVF (Prop p args))
  hence VAs:⋀i. Vagree ν ν' (FVT (args i))
    unfolding FVF.simps Vagree-def by auto
  have sems:⋀i. dterm-sem I (args i) ν = dterm-sem J (args i) ν'
    using IAs VAs coincidence-dterm' rangeI safes
    by (simp add: coincidence-dterm')
  hence vecSem:(χ i. dterm-sem I (args i) ν) = (χ i. dterm-sem J (args i) ν')
    by auto
  show (ν ∈ fml-sem I (Prop p args)) = (ν' ∈ fml-sem J (Prop p args))
    apply(unfold fml-sem.simps mem-Collect-eq)
    using IA vecSem pSame by (auto)
 qed
 then show ?case unfolding coincide-fml-def by blast
next
 case fsafe-Not then show ?case by auto
next
 case (fsafe-And p1 p2)
 then have safes:fsafe p1 fsafe p2
  and IH1:∀ ν ν' I J. Iagree I J (SIGF p1) ⟶ Vagree ν ν' (FVF p1) ⟶ (ν
∈ fml-sem I p1) = (ν' ∈ fml-sem J p1)
  and IH2:∀ ν ν' I J. Iagree I J (SIGF p2) ⟶ Vagree ν ν' (FVF p2) ⟶ (ν
∈ fml-sem I p2) = (ν' ∈ fml-sem J p2)
   by auto
 have almost:⋀ν ν' I J. Iagree I J (SIGF (And p1 p2)) ⟹ Vagree ν ν' (FVF
(And p1 p2)) ⟹ (ν ∈ fml-sem I (And p1 p2)) = (ν' ∈ fml-sem J (And p1 p2))
 proof −
  fix ν ν' I J
  assume IA:Iagree I J (SIGF (And p1 p2))
  have IAsubs:(SIGF p1) ⊆ (SIGF (And p1 p2)) (SIGF p2) ⊆ (SIGF (And p1
p2)) by auto
  from IA have IAs:Iagree I J (SIGF p1) Iagree I J (SIGF p2)
    using Iagree-sub[OF IAsubs(1)] Iagree-sub[OF IAsubs(2)] by auto
  assume VA:Vagree ν ν' (FVF (And p1 p2))
  hence VAs:Vagree ν ν' (FVF p1) Vagree ν ν' (FVF p2)
    unfolding FVF.simps Vagree-def by auto
  have eq1:(ν ∈ fml-sem I p1) = (ν' ∈ fml-sem J p1) using IH1 IAs VAs by
blast
  have eq2:(ν ∈ fml-sem I p2) = (ν' ∈ fml-sem J p2) using IH2 IAs VAs by
blast
```

```
    show (ν ∈ fml-sem I (And p1 p2)) = (ν' ∈ fml-sem J (And p1 p2))
      using eq1 eq2 by auto
  qed
  then show ?case unfolding coincide-fml-def by blast
next
  case (fsafe-Exists p x)
  then have safe:fsafe p
    and IH:∀ ν ν' I J. Iagree I J (SIGF p) ⟶ Vagree ν ν' (FVF p) ⟶ (ν ∈
fml-sem I p) = (ν' ∈ fml-sem J p)
    by auto
  have almost:⋀ν ν' I J. Iagree I J (SIGF (Exists x p)) ⟹ Vagree ν ν' (FVF
(Exists x p)) ⟹ (ν ∈ fml-sem I (Exists x p)) = (ν' ∈ fml-sem J (Exists x p))
  proof −
    fix ν ν' I J
    assume IA:Iagree I J (SIGF (Exists x p))
    hence IA':Iagree I J (SIGF p)
      unfolding SIGF.simps Iagree-def by auto
    assume VA:Vagree ν ν' (FVF (Exists x p))
    hence VA':Vagree ν ν' (FVF p − {Inl x}) by auto
    hence VA'':⋀r. Vagree (repv ν x r) (repv ν' x r) (FVF p)
      subgoal for r
        unfolding Vagree-def FVF.simps repv.simps
        by auto
      done
    have IH': ⋀r. Iagree I J (SIGF p) ⟹ Vagree (repv ν x r) (repv ν' x r) (FVF
p) ⟹ ((repv ν x r) ∈ fml-sem I p) = ((repv ν' x r) ∈ fml-sem J p)
      subgoal for r
        using IH apply(rule allE[where x = repv ν x r])
        apply(erule allE[where x = repv ν' x r])
        by (auto)
      done
    hence IH'':⋀r. ((repv ν x r) ∈ fml-sem I p) = ((repv ν' x r) ∈ fml-sem J p)
      subgoal for r
        using IA' VA'' by auto
      done
    have fact:⋀r. (repv ν x r ∈ fml-sem I p) = (repv ν' x r ∈ fml-sem J p)
      subgoal for r
        using IH'[OF IA' VA''] by auto
      done
    show (ν ∈ fml-sem I (Exists x p)) = (ν' ∈ fml-sem J (Exists x p))
      apply(simp only: fml-sem.simps mem-Collect-eq)
      using IH'' by auto
  qed
  then show ?case unfolding coincide-fml-def by blast
next
  case (fsafe-Diamond a p) then
  have hsafe:hpsafe a
    and psafe:fsafe p
    and IH1:∀ I J. (∀ν ν' μ V. Iagree I J (SIGP a) ⟶
```

Vagree $\nu\ \nu'\ V \longrightarrow$
FVP $a \subseteq V \longrightarrow (\nu,\ \mu) \in$ *prog-sem* $I\ a \longrightarrow (\exists \mu'.\ (\nu',\ \mu') \in$ *prog-sem* $J\ a \wedge$ *Vagree* $\mu\ \mu'\ (MBV\ a \cup V)))$
and *IH2*:$\forall \nu\ \nu'\ I\ J.$ *Iagree* $I\ J\ (SIGF\ p) \longrightarrow$ *Vagree* $\nu\ \nu'\ (FVF\ p) \longrightarrow (\nu \in$ *fml-sem* $I\ p) = (\nu' \in$ *fml-sem* $J\ p)$
unfolding *coincide-hp'-def coincide-hp-def coincide-fml-def* **apply** *auto* **done**
have *almost*:$\bigwedge \nu\ \nu'\ I\ J.$ *Iagree* $I\ J\ (SIGF\ (Diamond\ a\ p)) \Longrightarrow$ *Vagree* $\nu\ \nu'\ (FVF\ (Diamond\ a\ p)) \Longrightarrow (\nu \in$ *fml-sem* $I\ (Diamond\ a\ p)) = (\nu' \in$ *fml-sem* $J\ (Diamond\ a\ p))$
proof −
fix $\nu\ \nu'\ I\ J$
assume *IA*:*Iagree* $I\ J\ (SIGF\ (Diamond\ a\ p))$
have *IAsubs*:$(SIGP\ a) \subseteq (SIGF\ (Diamond\ a\ p))\ (SIGF\ p) \subseteq (SIGF\ (Diamond\ a\ p))$ **by** *auto*
from *IA* **have** *IAP*:*Iagree* $I\ J\ (SIGP\ a)$
and *IAF*:*Iagree* $I\ J\ (SIGF\ p)$ **using** *Iagree-sub*[*OF IAsubs*(*1*)] *Iagree-sub*[*OF IAsubs*(*2*)] **by** *auto*
from *IAP* **have** *IAP'*:*Iagree* $J\ I\ (SIGP\ a)$ **by** (*rule Iagree-comm*)
from *IAF* **have** *IAF'*:*Iagree* $J\ I\ (SIGF\ p)$ **by** (*rule Iagree-comm*)
assume *VA*: *Vagree* $\nu\ \nu'\ (FVF\ (Diamond\ a\ p))$
hence *VA'*: *Vagree* $\nu'\ \nu\ (FVF\ (Diamond\ a\ p))$ **by** (*rule agree-comm*)
have *dir1*:$\nu \in$ *fml-sem* $I\ (Diamond\ a\ p) \Longrightarrow \nu' \in$ *fml-sem* $J\ (Diamond\ a\ p)$
proof −
assume *sem*:$\nu \in$ *fml-sem* $I\ (Diamond\ a\ p)$
let *?V* = *FVF* $(Diamond\ a\ p)$
have *Vsup*:*FVP* $a \subseteq$ *?V* **by** *auto*
obtain μ **where** *prog*:$(\nu,\ \mu) \in$ *prog-sem* $I\ a$ **and** *fml*:$\mu \in$ *fml-sem* $I\ p$
using *sem* **by** *auto*
from *IH1* **have** *IH1'*:
Iagree $I\ J\ (SIGP\ a) \Longrightarrow$
Vagree $\nu\ \nu'$ *?V* $\Longrightarrow$
FVP $a \subseteq$ *?V* $\Longrightarrow (\nu,\ \mu) \in$ *prog-sem* $I\ a \Longrightarrow (\exists \mu'.\ (\nu',\ \mu') \in$ *prog-sem* $J\ a \wedge$ *Vagree* $\mu\ \mu'\ (MBV\ a \cup$ *?V*$))$
by *blast*
obtain μ' **where** *prog'*:$(\nu',\ \mu') \in$ *prog-sem* $J\ a$ **and** *agree*: *Vagree* $\mu\ \mu'\ (MBV\ a \cup$ *?V*$)$
using *IH1'*[*OF IAP VA Vsup prog*] **by** *blast*
from *IH2*
have *IH2'*:*Iagree* $I\ J\ (SIGF\ p) \Longrightarrow$ *Vagree* $\mu\ \mu'\ (FVF\ p) \Longrightarrow (\mu \in$ *fml-sem* $I\ p) = (\mu' \in$ *fml-sem* $J\ p)$
by *blast*
have *VAF*: *Vagree* $\mu\ \mu'\ (FVF\ p)$
using *agree VA* **by** (*auto simp only*: *Vagree-def FVF.simps*)
hence *IH2''*:$(\mu \in$ *fml-sem* $I\ p) = (\mu' \in$ *fml-sem* $J\ p)$
using *IH2'*[*OF IAF VAF*] **by** *auto*
have *fml'*:$\mu' \in$ *fml-sem* $J\ p$ **using** *IH2'' fml* **by** *auto*
have $\exists\ \mu'.\ (\nu',\ \mu') \in$ *prog-sem* $J\ a \wedge \mu' \in$ *fml-sem* $J\ p$ **using** *fml' prog'* **by** *blast*
then show $\nu' \in$ *fml-sem* $J\ (Diamond\ a\ p)$

unfolding *fml-sem.simps* **by** (*auto simp only*: *mem-Collect-eq*)
qed
have *dir2*:$\nu' \in$ *fml-sem J* (*Diamond a p*) $\Longrightarrow \nu \in$ *fml-sem I* (*Diamond a p*)
proof −
assume *sem*:$\nu' \in$ *fml-sem J* (*Diamond a p*)
let *?V* = *FVF* (*Diamond a p*)
have *Vsup*:*FVP a* ⊆ *?V* **by** *auto*
obtain μ **where** *prog*:(ν', μ) ∈ *prog-sem J a* **and** *fml*:$\mu \in$ *fml-sem J p*
using *sem* **by** *auto*
from *IH1* **have** *IH1'*:
Iagree J I (*SIGP a*) ⟹
Vagree ν' ν *?V* ⟹
FVP a ⊆ *?V* ⟹ (ν', μ) ∈ *prog-sem J a* ⟹ (∃ μ'. (ν, μ') ∈ *prog-sem I a* ∧ *Vagree* μ μ' (*MBV a* ∪ *?V*))
by *blast*
obtain μ' **where** *prog'*:(ν, μ') ∈ *prog-sem I a* **and** *agree*:*Vagree* μ μ' (*MBV a* ∪ *?V*)
using *IH1'*[*OF IAP' VA' Vsup prog*] **by** *blast*
from *IH2*
have *IH2'*:*Iagree J I* (*SIGF p*) ⟹ *Vagree* μ μ' (*FVF p*) ⟹ ($\mu \in$ *fml-sem J p*) = ($\mu' \in$ *fml-sem I p*)
by *blast*
have *VAF*: *Vagree* μ μ' (*FVF p*)
using *agree VA* **by** (*auto simp only*: *Vagree-def FVF.simps*)
hence *IH2''*:($\mu \in$ *fml-sem J p*) = ($\mu' \in$ *fml-sem I p*)
using *IH2'*[*OF IAF' VAF*] **by** *auto*
have *fml'*:$\mu' \in$ *fml-sem I p* **using** *IH2'' fml* **by** *auto*
have ∃ μ'. (ν, μ') ∈ *prog-sem I a* ∧ $\mu' \in$ *fml-sem I p* **using** *fml' prog'* **by** *blast*
then show $\nu \in$ *fml-sem I* (*Diamond a p*)
unfolding *fml-sem.simps* **by** (*auto simp only*: *mem-Collect-eq*)
qed
show ($\nu \in$ *fml-sem I* (*Diamond a p*)) = ($\nu' \in$ *fml-sem J* (*Diamond a p*))
using *dir1 dir2* **by** *auto*
qed
then show *?case* **unfolding** *coincide-fml-def* **by** *blast*
next
case (*fsafe-InContext* φ) **then**
have *safe*:*fsafe* φ
and *IH*:(∀ ν ν' *I J*. *Iagree I J* (*SIGF* φ) ⟶ *Vagree* ν ν' (*FVF* φ) ⟶ $\nu \in$ *fml-sem I* φ ⟷ $\nu' \in$ *fml-sem J* φ)
by (*unfold coincide-fml-def*)
hence *IH'*:⋀ν ν' *I J*. *Iagree I J* (*SIGF* φ) ⟹ *Vagree* ν ν' (*FVF* φ) ⟹ $\nu \in$ *fml-sem I* φ ⟷ $\nu' \in$ *fml-sem J* φ
by *auto*
hence *sem-eq*:⋀*I J*. *Iagree I J* (*SIGF* φ) ⟹ *fml-sem I* φ = *fml-sem J* φ
apply (*auto simp*: *Collect-cong Collect-mem-eq agree-refl*)
using *agree-refl* **by** *blast+*
have (⋀ ν ν' *I J C* . *Iagree I J* (*SIGF* (*InContext C* φ)) ⟹ *Vagree* ν ν'

(*FVF* (*InContext C* φ)) $\Longrightarrow$ $\nu \in$ *fml-sem I* (*InContext C* φ) $\longleftrightarrow$ $\nu' \in$ *fml-sem J* (*InContext C* φ))
proof −
fix ν ν' *I J C*
assume *IA*:*Iagree I J* (*SIGF* (*InContext C* φ))
then have *IA'*:*Iagree I J* (*SIGF* φ) **unfolding** *SIGF.simps Iagree-def* **by** *auto*
assume *VA*:*Vagree* ν ν' (*FVF* (*InContext C* φ))
then have *VAU*:*Vagree* ν ν' *UNIV* **unfolding** *FVF.simps Vagree-def* **by** *auto*
then have *VA'*:*Vagree* ν ν' (*FVF* φ) **unfolding** *FVF.simps Vagree-def* **by** *auto*
from *VAU* **have** *eq*:$\nu = \nu'$ **by** (*cases* ν, *cases* ν', *auto simp add*: *vec-eq-iff Vagree-def*)
from *IA* **have** *Cmem*:*Inr* (*Inl C*) $\in$ *SIGF* (*InContext C* φ)
by *simp*
have *Cagree*:*Contexts I C* = *Contexts J C* **by** (*rule Iagree-Contexts*[*OF IA Cmem*])
show $\nu \in$ *fml-sem I* (*InContext C* φ) $\longleftrightarrow$ $\nu' \in$ *fml-sem J* (*InContext C* φ)
using *Cagree eq sem-eq IA'* **by** (*auto*)
qed
then show *?case* **by** *simp*
qed

lemma *coincidence-formula*:$\bigwedge \nu$ ν' *I J*. *fsafe* (φ::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *formula*) $\Longrightarrow$ *Iagree I J* (*SIGF* φ) $\Longrightarrow$ *Vagree* ν ν' (*FVF* φ) $\Longrightarrow$ ($\nu \in$ *fml-sem I* φ $\longleftrightarrow$ $\nu' \in$ *fml-sem J* φ)
using *coincidence-hp-fml* **unfolding** *coincide-fml-def* **by** *blast*

lemma *coincidence-hp*:
fixes ν ν' μ *V I J*
assumes *safe*:*hpsafe* (α::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *hp*)
assumes *IA*:*Iagree I J* (*SIGP* α)
assumes *VA*:*Vagree* ν ν' *V*
assumes *sub*:*V* $\supseteq$ (*FVP* α)
assumes *sem*:(ν, μ) $\in$ *prog-sem I* α
shows ($\exists \mu'$. (ν', μ') $\in$ *prog-sem J* α $\wedge$ *Vagree* μ μ' (*MBV* α $\cup$ *V*))
proof −
have *thing*:($\forall$ *I J*. ($\forall \nu$ ν' μ *V*.
Iagree I J (*SIGP* α) $\longrightarrow$
Vagree ν ν' *V* $\longrightarrow$ *FVP* $\alpha \subseteq$ *V* $\longrightarrow$ (ν, μ) $\in$ *prog-sem I* α $\longrightarrow$ ($\exists \mu'$. (ν', μ') $\in$ *prog-sem J* α $\wedge$ *Vagree* μ μ' (*MBV* α $\cup$ *V*))) $\wedge$
ode-sem-equiv α *I*)
using *coincidence-hp-fml* **unfolding** *coincide-hp-def coincide-hp'-def*
using *safe* **by** *blast*
then have (*Iagree I J* (*SIGP* α) $\Longrightarrow$
Vagree ν ν' *V* $\Longrightarrow$ *FVP* $\alpha \subseteq$ *V* $\Longrightarrow$ (ν, μ) $\in$ *prog-sem I* α $\Longrightarrow$ ($\exists \mu'$. (ν', μ') $\in$ *prog-sem J* α $\wedge$ *Vagree* μ μ' (*MBV* α $\cup$ *V*)))
using *IA VA sub sem thing* **by** *blast*

then show ($\exists \mu'$. (ν', μ') $\in$ *prog-sem J* $\alpha \wedge$ *Vagree* μ μ' (*MBV* $\alpha \cup V$))
using *IA VA sub sem* **by** *auto*
qed

7.4 Corollaries: Alternate ODE semantics definition

lemma *ode-sem-eq*:
fixes *I*::('*a*::*finite*,'*b*::*finite*,'*c*::*finite*) *interp* **and** *ODE*::('*a*,'*c*) *ODE* **and** φ::('*a*,'*b*,'*c*) *formula*
assumes *osafe*:*osafe ODE*
assumes *fsafe*:*fsafe* φ
shows
({(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
t $\geq$ *0* $\wedge$
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
VSagree (*sol 0*) (*fst* ν) {*x* | *x*. *Inl x* $\in$ *FVP* (*EvolveODE ODE* φ)}}) =
({(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
t $\geq$ *0* $\wedge$
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
(*sol 0*) = (*fst* ν)})
proof −
have *hpsafe*:*hpsafe* (*EvolveODE ODE* φ) **using** *osafe fsafe* **by** (*auto intro*: *hp-safe-fsafe.intros*)
have *coincide-hp'*(*EvolveODE ODE* φ) **using** *coincidence-hp-fml hpsafe* **by** *blast*
hence *ode-sem-equiv* (*EvolveODE ODE* φ) *I* **unfolding** *coincide-hp'-def* **by** *auto*
then show *?thesis*
unfolding *ode-sem-equiv-def* **using** *osafe fsafe* **by** *auto*
qed

lemma *ode-alt-sem*:$\bigwedge$*I*::('*a*::*finite*,'*b*::*finite*,'*c*::*finite*) *interp*. $\bigwedge$*ODE*::('*a*,'*c*) *ODE*. $\bigwedge\varphi$::('*a*,'*b*,'*c*)*formula*. *osafe ODE* $\Longrightarrow$ *fsafe* φ $\Longrightarrow$
prog-sem I (*EvolveODE ODE* φ)
=
{(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
t $\geq$ *0* $\wedge$
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
VSagree (*sol 0*) (*fst* ν) {*x* | *x*. *Inl x* $\in$ *FVP* (*EvolveODE ODE* φ)}}

subgoal for *I ODE* φ
using *ode-sem-eq*[*of ODE* φ *I*] **by** *auto*
done
end
end
theory *Bound-Effect*
imports
Ordinary-Differential-Equations.ODE-Analysis

Ids
Lib
Syntax
Denotational-Semantics
Frechet-Correctness
Static-Semantics
Coincidence
begin

8 Bound Effect Theorem

The bound effect lemma says that a program can only modify its bound variables and nothing else. This is one of the major lemmas for showing correctness of uniform substitution.

context *ids* **begin**
lemma *bound-effect*:
fixes *I*::('*sf*,'*sc*,'*sz*) *interp*
assumes *good-interp*:*is-interp I*
shows $\bigwedge\nu$ :: '*sz state*. $\bigwedge\omega$ ::'*sz state*. *hpsafe* $\alpha \Longrightarrow (\nu, \omega) \in$ *prog-sem I* $\alpha \Longrightarrow$ *Vagree* $\nu\ \omega$ $(-$ *(BVP* $\alpha))$
proof (*induct rule*: *hp-induct*)
case *Var* **then show** *?case*
using *agree-nil Compl-UNIV-eq BVP.simps*(*1*) **by** *fastforce*
next
case *Test* **then show** *?case*
by *auto*(*simp add*: *agree-refl Compl-UNIV-eq Vagree-def*)
next
case (*Choice a b* $\nu\ \omega$)
assume *IH1*:$\bigwedge\nu'$. $\bigwedge\omega'$. *hpsafe a* $\Longrightarrow$((ν', ω') $\in$ *prog-sem I a* $\Longrightarrow$ *Vagree* $\nu'\ \omega'$ $(-$ *BVP a*))
assume *IH2*:$\bigwedge\nu'$. $\bigwedge\omega'$. *hpsafe b* $\Longrightarrow$((ν', ω') $\in$ *prog-sem I b* $\Longrightarrow$ *Vagree* $\nu'\ \omega'$ $(-$ *BVP b*))
assume *sem*:$(\nu, \omega) \in$ *prog-sem I* $(a \cup\cup b)$
assume *safe*:*hpsafe* (*Choice a b*)
from *safe* **have** *safes*:*hpsafe a hpsafe b* **by** (*auto dest*: *hpsafe.cases*)
have *sems*:$(\nu, \omega) \in$ *prog-sem I* $(a) \vee (\nu, \omega) \in$ *prog-sem I* (b) **using** *sem* **by** *auto*
have *agrees*:*Vagree* $\nu\ \omega$ $(-$ *BVP a*$) \vee$ *Vagree* $\nu\ \omega$ $(-$ *BVP b*) **using** *IH1 IH2 sems safes* **by** *blast*
have *sub1*:$-$(*BVP a*) $\supseteq$ $(-$ *BVP a* $\cap -$ *BVP b*) **by** *auto*
have *sub2*:$-$(*BVP a*) $\supseteq$ $(-$ *BVP a* $\cap -$ *BVP b*) **by** *auto*
have *res*:*Vagree* $\nu\ \omega$ $(-$ *BVP a* $\cap -$ *BVP b*) **using** *agrees sub1 sub2 agree-supset* **by** *blast*
then show *?case* **by** *auto*
next
case (*Compose a b* $\nu\ \omega$)
assume *IH1*:$\bigwedge\nu'$. $\bigwedge\omega'$. *hpsafe a* $\Longrightarrow$ $(\nu', \omega') \in$ *prog-sem I a* $\Longrightarrow$ *Vagree* $\nu'\ \omega'$ $(-$ *BVP a*)
assume *IH2*:$\bigwedge\nu'$. $\bigwedge\omega'$. *hpsafe b* $\Longrightarrow$ $(\nu', \omega') \in$ *prog-sem I b* $\Longrightarrow$ *Vagree* $\nu'\ \omega'$ $(-$

```
BVP b)
  assume sem:(ν, ω) ∈ prog-sem I (a ;; b)
  assume safe:hpsafe (a ;; b)
  from safe have safes:hpsafe a hpsafe b by (auto dest: hpsafe.cases)
  then show ?case
    using agree-trans IH1 IH2 sem safes by fastforce
next
  fix ODE::('sf,'sz) ODE and P::('sf,'sc,'sz) formula and ν ω
  assume safe:hpsafe (EvolveODE ODE P)
  from safe have osafe:osafe ODE and fsafe:fsafe P by (auto dest: hpsafe.cases)
   show (ν, ω) ∈ prog-sem I (EvolveODE ODE P) ⟹ Vagree ν ω (− BVP
(EvolveODE ODE P))
  proof −
    assume sem:(ν, ω) ∈ prog-sem I (EvolveODE ODE P)
    from sem have agree:Vagree ν ω (− BVO ODE)
      apply(simp only: prog-sem.simps(8) mem-Collect-eq osafe fsafe)
      apply(erule exE)+
    proof −
      fix ν' sol t
      assume assm:
        (ν, ω) = (ν', mk-v I ODE ν' (sol t)) ∧
         0 ≤ t ∧
           (sol solves-ode (λ-. ODE-sem I ODE)) {0..t} {x. mk-v I ODE ν' x ∈
fml-sem I P} ∧ (sol 0) = (fst ν')
      have semBV:−BVO ODE ⊆ −semBV I ODE
        by(induction ODE, auto)
       from assm have Vagree ω ν (− BVO ODE) using mk-v-agree[of I ODE ν
(sol t)]
         using agree-sub[OF semBV] by auto
      thus Vagree ν ω (− BVO ODE) by (rule agree-comm)
    qed
    thus Vagree ν ω (− BVP (EvolveODE ODE P)) by auto
  qed
next
  case (Star a ν ω) then
  have IH:(⋀ν ω. hpsafe a ⟹ (ν, ω) ∈ prog-sem I a ⟹ Vagree ν ω (− BVP a))
    and safe:hpsafe a**
    and sem:(ν, ω) ∈ prog-sem I a**
    by auto
  from safe have asafe:hpsafe a by (auto dest: hpsafe.cases)
  show Vagree ν ω (− BVP a**)
    using sem apply (simp only: prog-sem.simps)
    apply (erule converse-rtrancl-induct)
     subgoal by(rule agree-refl)
   subgoal for y z using IH[of y z, OF asafe] sem by (auto simp add: Vagree-def)
    done
qed (auto simp add: Vagree-def)
end end
theory Differential-Axioms
```

imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
Frechet-Correctness
Axioms
Coincidence
begin context *ids* **begin**

9 Differential Axioms

Differential axioms fall into two categories: Axioms for computing the derivatives of terms and axioms for proving properties of ODEs. The derivative axioms are all corollaries of the frechet correctness theorem. The ODE axioms are more involved, often requiring extensive use of the ODE libraries.

9.1 Derivative Axioms

definition *diff-const-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*diff-const-axiom* ≡ *Equals* (*Differential* ($*f fid1 empty*)) (*Const 0*)

definition *diff-var-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*diff-var-axiom* ≡ *Equals* (*Differential* (*Var vid1*)) (*DiffVar vid1*)

definition *state-fun* ::*'sf* ⇒ (*'sf*, *'sz*) *trm*
where [*axiom-defs*]:*state-fun f* = ($*f f* (λ*i*. *Var i*))

definition *diff-plus-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*diff-plus-axiom* ≡ *Equals* (*Differential* (*Plus* (*state-fun fid1*) (*state-fun fid2*)))
(*Plus* (*Differential* (*state-fun fid1*)) (*Differential* (*state-fun fid2*)))

definition *diff-times-axiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*diff-times-axiom* ≡ *Equals* (*Differential* (*Times* (*state-fun fid1*) (*state-fun fid2*)))
(*Plus* (*Times* (*Differential* (*state-fun fid1*)) (*state-fun fid2*))
(*Times* (*state-fun fid1*) (*Differential* (*state-fun fid2*))))

— [*y*=*g*(*x*)][*y'*=*1*](*f*(*g*(*x*))' = *f*(*y*)')
definition *diff-chain-axiom*::(*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*diff-chain-axiom* ≡ [[*Assign vid2* (*f1 fid2 vid1*)]]([[*DiffAssign vid2* (*Const 1*)]]
(*Equals* (*Differential* ($*f fid1* (*singleton* (*f1 fid2 vid1*)))) (*Times* (*Differential* (*f1 fid1 vid2*)) (*Differential* (*f1 fid2 vid1*)))))

9.2 ODE Axioms

definition *DWaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*DWaxiom* = ([[*EvolveODE* (*OVar vid1*) (*Predicational pid1*)]](*Predicational pid1*))

definition *DWaxiom'* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*DWaxiom'* = ([[*EvolveODE* (*OSing vid1* (*Function fid1* (*singleton* (*Var vid1*)))) (*Prop vid2* (*singleton* (*Var vid1*)))]](*Prop vid2* (*singleton* (*Var vid1*))))

definition *DCaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*DCaxiom* = (
([[*EvolveODE* (*OVar vid1*) (*Predicational pid1*)]]*Predicational pid3*) →
(([[*EvolveODE* (*OVar vid1*) (*Predicational pid1*)]](*Predicational pid2*))
↔
([[*EvolveODE* (*OVar vid1*) (*And* (*Predicational pid1*) (*Predicational pid3*))]]*Predicational pid2*)))

definition *DEaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*DEaxiom* =
(([[*EvolveODE* (*OSing vid1* (*f1 fid1 vid1*)) (*p1 vid2 vid1*)]] (*P pid1*))
↔
([[*EvolveODE* (*OSing vid1* (*f1 fid1 vid1*)) (*p1 vid2 vid1*)]]
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*))

definition *DSaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*DSaxiom* =
(([[*EvolveODE* (*OSing vid1* (*f0 fid1*)) (*p1 vid2 vid1*)]]*p1 vid3 vid1*)
↔
(*Forall vid2*
(*Implies* (*Geq* (*Var vid2*) (*Const 0*))
(*Implies*
(*Forall vid3*
(*Implies* (*And* (*Geq* (*Var vid3*) (*Const 0*)) (*Geq* (*Var vid2*) (*Var vid3*)))
(*Prop vid2* (*singleton* (*Plus* (*Var vid1*) (*Times* (*f0 fid1*) (*Var vid3*)))))))
([[*Assign vid1* (*Plus* (*Var vid1*) (*Times* (*f0 fid1*) (*Var vid2*)))]]*p1 vid3 vid1*)))))

— ($Q \rightarrow [c\&Q](f(x)' \geq g(x)')$)
— →
— ($[c\&Q](f(x) \geq g(x))$) −−> ($Q \rightarrow (f(x) \geq g(x))$
definition *DIGeqaxiom* :: (*'sf*, *'sc*, *'sz*) *formula*
where [*axiom-defs*]:*DIGeqaxiom* =
Implies
(*Implies* (*Prop vid1 empty*) ([[*EvolveODE* (*OVar vid1*) (*Prop vid1 empty*)]](*Geq* (*Differential* (*f1 fid1 vid1*)) (*Differential* (*f1 fid2 vid1*)))))
(*Implies*
(*Implies*(*Prop vid1 empty*) (*Geq* (*f1 fid1 vid1*) (*f1 fid2 vid1*)))
([[*EvolveODE* (*OVar vid1*) (*Prop vid1 empty*)]](*Geq* (*f1 fid1 vid1*) (*f1 fid2 vid1*)))))

— $g(x) > h(x) \rightarrow [x'=f(x),\ c\ \&\ p(x)](g(x)' \geq h(x)') \rightarrow [x'=f(x),\ c\ \&\ p(x)]g(x) > h(x)$

— $(Q \rightarrow [c\&Q](f(x)' \geq g(x)')$
— $\rightarrow$
— $([c\&Q](f(x) > g(x))) <-> (Q \rightarrow (f(x) > g(x))$
definition *DIGraxiom* :: $('sf, 'sc, 'sz)$ *formula*
where [*axiom-defs*]:*DIGraxiom* =
Implies
(*Implies* (*Prop vid1 empty*) ([[*EvolveODE* (*OVar vid1*) (*Prop vid1 empty*)]](*Geq* (*Differential* (*f1 fid1 vid1*)) (*Differential* (*f1 fid2 vid1*)))))
(*Implies*
(*Implies*(*Prop vid1 empty*) (*Greater* (*f1 fid1 vid1*) (*f1 fid2 vid1*)))
([[*EvolveODE* (*OVar vid1*) (*Prop vid1 empty*)]](*Greater* (*f1 fid1 vid1*) (*f1 fid2 vid1*))))

— $[\{1' = 1(1)\ \&\ 1(1)\}]2(1) <->$
— $\exists\, 2.\ [\{1'=1(1),\ 2' = 2(1)*2 + 3(1)\ \&\ 1(1)\}]2(1)*)$
definition *DGaxiom* :: $('sf, 'sc, 'sz)$ *formula*
where [*axiom-defs*]:*DGaxiom* = (([[*EvolveODE* (*OSing vid1* (*f1 fid1 vid1*)) (*p1 vid1 vid1*)]]*p1 vid2 vid1*) $\leftrightarrow$
(*Exists vid2*
([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*)) (*OSing vid2* (*Plus* (*Times* (*f1 fid2 vid1*) (*Var vid2*)) (*f1 fid3 vid1*)))) (*p1 vid1 vid1*)]]
p1 vid2 vid1)))

9.3 Proofs for Derivative Axioms

lemma *constant-deriv-inner*:
assumes *interp*:$\forall\, x\ i.$ (*Functions I i has-derivative FunctionFrechet I i x*) (*at x*)
shows *FunctionFrechet I id1* (*vec-lambda* ($\lambda i.$ *sterm-sem I* (*empty i*) (*fst* ν))) (*vec-lambda*($\lambda i.$ *frechet I* (*empty i*) (*fst* ν) (*snd* ν)))= *0*
proof −
have *empty-zero*:(*vec-lambda*($\lambda i.$ *frechet I* (*empty i*) (*fst* ν) (*snd* ν))) = *0*
using *local.empty-def Cart-lambda-cong frechet.simps*(*5*) *zero-vec-def*
apply *auto*
apply(*rule vec-extensionality*)
using *local.empty-def Cart-lambda-cong frechet.simps*(*5*) *zero-vec-def*
by (*simp add*: *local.empty-def*)
let *?x* = (*vec-lambda* ($\lambda i.$ *sterm-sem I* (*empty i*) (*fst* ν)))
from *interp*
have *has-deriv*:(*Functions I id1 has-derivative FunctionFrechet I id1 ?x*) (*at ?x*)
by *auto*
then have *f-linear*:*linear* (*FunctionFrechet I id1 ?x*)
using *Deriv.has-derivative-linear* **by** *auto*
then show *?thesis* **using** *empty-zero f-linear linear-0* **by** (*auto*)
qed

lemma *constant-deriv-zero*:*is-interp I* $\Longrightarrow$ *directional-derivative I* (*$f id1 empty*)
$\nu = 0$
apply(*simp only*: *is-interp-def directional-derivative-def frechet.simps frechet-correctness*)
apply(*rule constant-deriv-inner*)
apply(*auto*)
done

theorem *diff-const-axiom-valid*: *valid diff-const-axiom*
apply(*simp only*: *valid-def diff-const-axiom-def equals-sem*)
apply(*rule allI* | *rule impI*)+
apply(*simp only*: *dterm-sem.simps constant-deriv-zero sterm-sem.simps*)
done

theorem *diff-var-axiom-valid*: *valid diff-var-axiom*
apply(*auto simp add*: *diff-var-axiom-def valid-def directional-derivative-def*)
by (*metis inner-prod-eq*)

theorem *diff-plus-axiom-valid*: *valid diff-plus-axiom*
apply(*auto simp add*: *diff-plus-axiom-def valid-def*)
subgoal for *I a b*
using *frechet-correctness*[*of I* (*Plus* (*state-fun fid1*) (*state-fun fid2*)) *b*]
unfolding *state-fun-def* **apply** (*auto intro*: *dfree.intros*)
unfolding *directional-derivative-def* **by** *auto*
done

theorem *diff-times-axiom-valid*: *valid diff-times-axiom*
apply(*auto simp add*: *diff-times-axiom-def valid-def*)
subgoal for *I a b*
using *frechet-correctness*[*of I* (*Times* (*state-fun fid1*) (*state-fun fid2*)) *b*]
unfolding *state-fun-def* **apply** (*auto intro*: *dfree.intros*)
unfolding *directional-derivative-def* **by** *auto*
done

9.4 Proofs for ODE Axioms

lemma *DW-valid*:*valid DWaxiom*
apply(*unfold DWaxiom-def valid-def Let-def impl-sem*)
apply(*safe*)
apply(*auto simp only*: *fml-sem.simps prog-sem.simps box-sem*)
subgoal for *I aa ba ab bb sol t* **using** *mk-v-agree*[*of I* (*OVar vid1*) (*ab*,*bb*) *sol t*]
Vagree-univ[*of aa ba sol t ODEs I vid1* (*sol t*)] *solves-ode-domainD*
by (*fastforce*)
done

lemma *DE-lemma*:
fixes *ab bb*::*'sz simple-state*
and *sol*::*real* $\Rightarrow$ *'sz simple-state*
and *I*::(*'sf*, *'sc*, *'sz*) *interp*
shows

repd (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1* (*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)))
= *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)
proof
have *set-eq*: {*Inl vid1*, *Inr vid1*} = {*Inr vid1*, *Inl vid1*} **by** *auto*
have *agree*:*Vagree* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) (*mk-xode I* (*OSing vid1* (*f1 fid1 vid1*)) (*sol t*))
{*Inl vid1*, *Inr vid1*}
using *mk-v-agree*[*of I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)]
unfolding *semBV.simps* **using** *set-eq* **by** *auto*
have *fact*:*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*))
= *snd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) $ *vid1*
using *agree* **unfolding** *Vagree-def dterm-sem.simps f1-def mk-xode.simps*
proof −
assume *alls*:(∀ *i*. *Inl i* ∈ {*Inl vid1*, *Inr vid1*} ⟶
fst (*mk-v I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*ab*, *bb*) (*sol t*)) $ *i* =
fst (*sol t*, *ODE-sem I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*sol t*)) $ *i*) ∧
(∀ *i*. *Inr i* ∈ {*Inl vid1*, *Inr vid1*} ⟶
snd (*mk-v I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*ab*, *bb*) (*sol t*)) $ *i* =
snd (*sol t*, *ODE-sem I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*sol t*)) $ *i*)
hence *atVid''*:*snd* (*mk-v I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*ab*, *bb*) (*sol t*)) $ *vid1* = *sterm-sem I* ($*f fid1* (*singleton* (*trm.Var vid1*))) (*sol t*)
by *auto*
have *argsEq*:(χ *i*. *dterm-sem I* (*singleton* (*trm.Var vid1*) *i*)
(*mk-v I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*ab*, *bb*) (*sol t*)))
= (χ *i*. *sterm-sem I* (*singleton* (*trm.Var vid1*) *i*) (*sol t*))
using *alls f1-def* **by** *auto*
thus *Functions I fid1* (χ *i*. *dterm-sem I* (*singleton* (*trm.Var vid1*) *i*)
(*mk-v I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*ab*, *bb*) (*sol t*)))

= *snd* (*mk-v I* (*OSing vid1* ($*f fid1* (*singleton* (*trm.Var vid1*)))) (*ab*, *bb*) (*sol t*)) $ *vid1*
by (*simp only*: *atVid'' ODE-sem.simps sterm-sem.simps dterm-sem.simps*)
qed
have *eqSnd*:(χ *y*. *if vid1* = *y then snd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) $ *vid1*
else snd (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) $ *y*) = *snd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*))
by (*simp add*: *vec-extensionality*)
have *truth*:*repd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)))
= *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)
using *fact* **by** (*auto simp only*: *eqSnd repd.simps fact prod.collapse split*: *if-split*)

thus *fst* (*repd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)))) =
fst (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*))

snd (*repd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)))) =
snd (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*))
by *auto*
qed

lemma *DE-valid*:*valid DEaxiom*
proof −
have *dsafe*:*dsafe* (*$f fid1* (*singleton* (*trm.Var vid1*))) **unfolding** *singleton-def* **by**(*auto intro*: *dsafe.intros*)
have *osafe*:*osafe*(*OSing vid1* (*f1 fid1 vid1*)) **unfolding** *f1-def empty-def singleton-def* **using** *dsafe osafe.intros dsafe.intros*
by (*simp add*: *osafe-Sing dfree-Const*)
have *fsafe*:*fsafe* (*p1 vid2 vid1*) **unfolding** *p1-def singleton-def* **using** *hpsafe-fsafe.intros*(*10*)
using *dsafe dsafe-Fun-simps image-iff*
by (*simp add*: *dfree-Const*)
show *valid DEaxiom*
apply(*auto simp only*: *DEaxiom-def valid-def Let-def iff-sem impl-sem*)
apply(*auto simp only*: *fml-sem.simps prog-sem.simps mem-Collect-eq box-sem*)
proof −
fix *I*::(*'sf*,*'sc*,*'sz*) *interp*
and *aa ba ab bb sol*
and *t*::*real*
and *ac bc*
assume *is-interp I*
assume *allw*:∀ *ω*. (∃ *ν sol t*.
((*ab*, *bb*), *ω*) = (*ν*, *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) *ν* (*sol t*)) ∧
0 ≤ *t* ∧
(*sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f1 fid1 vid1*)))) {*0*..*t*}
{*x*. *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) *ν x* ∈ *fml-sem I* (*p1 vid2 vid1*)} ∧
(*sol 0*) = (*fst ν*)) ⟶
ω ∈ *fml-sem I* (*P pid1*)
assume *t*:*0* ≤ *t*
assume *aaba*:(*aa*, *ba*) = *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)
assume *solve*: (*sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f1 fid1 vid1*)))) {*0*..*t*}
{*x*. *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) *x* ∈ *fml-sem I* (*p1 vid2 vid1*)}
assume *sol0*: (*sol 0*) = (*fst* (*ab*, *bb*))
assume *rep*: (*ac*, *bc*) =
repd (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*)

(*sol t*)))
have *aaba-sem*:(*aa*,*ba*) ∈ *fml-sem I* (*P pid1*) **using** *allw t aaba solve sol0 rep* **by** *blast*
have *truth*:*repd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)))
= *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)
using *DE-lemma* **by** *auto*
show
repd (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)))
∈ *fml-sem I* (*P pid1*) **using** *aaba aaba-sem truth* **by** (*auto*)
next
fix *I*::(*'sf*,*'sc*,*'sz*) *interp* **and** *aa ba ab bb sol* **and** *t*::*real*
assume *is-interp I*
assume *all*:∀ ω. (∃ ν *sol t*.
((*ab*, *bb*), ω) = (ν, *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) ν (*sol t*)) ∧
0 ≤ *t* ∧
(*sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f1 fid1 vid1*)))) {0..*t*}
{*x*. *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) ν *x* ∈ *fml-sem I* (*p1 vid2 vid1*)} ∧
(*sol* 0) = (*fst* ν)) ⟶
(∀ ω'. ω' = *repd* ω *vid1* (*dterm-sem I* (*f1 fid1 vid1*) ω) ⟶ ω' ∈ *fml-sem I* (*P pid1*))
hence *justW*:(∃ ν *sol t*.
((*ab*, *bb*), (*aa*, *ba*)) = (ν, *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) ν (*sol t*)) ∧
0 ≤ *t* ∧
(*sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f1 fid1 vid1*)))) {0..*t*}
{*x*. *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) ν *x* ∈ *fml-sem I* (*p1 vid2 vid1*)} ∧
(*sol* 0) = (*fst* ν)) ⟶
(∀ ω'. ω' = *repd* (*aa*, *ba*) *vid1* (*dterm-sem I* (*f1 fid1 vid1*) (*aa*, *ba*)) ⟶ ω' ∈ *fml-sem I* (*P pid1*))
by (*rule allE*)
assume *t*:0 ≤ *t*
assume *aaba*:(*aa*, *ba*) = *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)
assume *sol*:(*sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f1 fid1 vid1*)))) {0..*t*}
{*x*. *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) *x* ∈ *fml-sem I* (*p1 vid2 vid1*)}
assume *sol0*: (*sol* 0) = (*fst* (*ab*, *bb*))
have *repd* (*aa*, *ba*) *vid1* (*dterm-sem I* (*f1 fid1 vid1*) (*aa*, *ba*)) ∈ *fml-sem I* (*P pid1*)
using *justW t aaba sol sol0* **by** *auto*
hence *foo*:*repd* (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1* (*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*))) ∈ *fml-sem I* (*P pid1*)
using *aaba* **by** *auto*

```
        hence repd (mk-v I (OSing vid1 (f1 fid1 vid1)) (ab, bb) (sol t)) vid1
(dterm-sem I (f1 fid1 vid1) (mk-v I (OSing vid1 (f1 fid1 vid1)) (ab, bb) (sol t)))
            = mk-v I (OSing vid1 (f1 fid1 vid1)) (ab, bb) (sol t) using DE-lemma
by auto
      thus mk-v I (OSing vid1 (f1 fid1 vid1)) (ab, bb) (sol t) ∈ fml-sem I (P pid1)
using foo by auto
  qed
qed

lemma ODE-zero:⋀i. Inl i ∉ BVO ODE ⟹ Inr i ∉ BVO ODE ⟹ ODE-sem I
ODE ν $ i= 0
  by(induction ODE, auto)

lemma DE-sys-valid:
  assumes disj:{Inl vid1, Inr vid1} ∩ BVO ODE = {}
  shows valid (([[EvolveODE (OProd  (OSing vid1 (f1 fid1 vid1)) ODE) (p1 vid2
vid1)]] (P pid1)) ↔
 ([[EvolveODE ((OProd  (OSing vid1 (f1 fid1 vid1))ODE)) (p1 vid2 vid1)]]
    [[DiffAssign vid1 (f1 fid1 vid1)]]P pid1))
proof −
   have dsafe:dsafe ($f fid1 (singleton (trm.Var vid1))) unfolding singleton-def
by(auto intro: dsafe.intros)
  have osafe:osafe(OSing vid1 (f1 fid1 vid1)) unfolding f1-def empty-def single-
ton-def using dsafe osafe.intros dsafe.intros
    by (simp add: osafe-Sing dfree-Const)
 have fsafe:fsafe (p1 vid2 vid1) unfolding p1-def singleton-def using hpsafe-fsafe.intros(10)
    using dsafe dsafe-Fun-simps image-iff
    by (simp add: dfree-Const)
  show valid (([[EvolveODE (OProd (OSing vid1 (f1 fid1 vid1)) ODE) (p1 vid2
vid1)]] (P pid1)) ↔
 ([[EvolveODE ((OProd (OSing vid1 (f1 fid1 vid1)) ODE)) (p1 vid2 vid1)]]
    [[DiffAssign vid1 (f1 fid1 vid1)]]P pid1))
    apply(auto simp only: DEaxiom-def valid-def Let-def iff-sem impl-sem)
    apply(auto simp only: fml-sem.simps prog-sem.simps mem-Collect-eq box-sem
f1-def p1-def P-def expand-singleton)
   proof −
     fix I ::('sf,'sc,'sz) interp
       and aa ba ab bb sol
       and t::real
       and ac bc
     assume good:is-interp I
     assume bigAll:
     ∀ω. (∃ν sol t. ((ab, bb), ω) = (ν, mk-v I (OProd (OSing vid1 ($f fid1 (λi. if
i = vid1 then trm.Var vid1 else Const 0))) ODE) ν (sol t)) ∧
                  0 ≤ t ∧
                  (sol solves-ode (λ-. ODE-sem I (OProd(OSing vid1 ($f fid1 (λi. if
i = vid1 then trm.Var vid1 else Const 0))) ODE ))) {0..t}
                   {x. Predicates I vid2
                      (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
```

```
                    (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1
then trm.Var vid1 else Const 0)))ODE) ν x))} ∧
              sol 0 = fst ν) ⟶
        ω ∈ fml-sem I (Pc pid1)
    let ?myω = mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0)))ODE) (ab,bb) (sol t)
    assume t:0 ≤ t
    assume aaba:(aa, ba) = mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1
then trm.Var vid1 else Const 0)))ODE) (ab, bb) (sol t)
    assume sol:(sol solves-ode (λ-. ODE-sem I (OProd (OSing vid1 ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0)))ODE))) {0..t}
     {x. Predicates I vid2
         (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
               (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0)))ODE) (ab, bb) x))}
    assume sol0:sol 0 = fst (ab, bb)
    assume acbc:(ac, bc) =
    repd (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1
else Const 0)))ODE) (ab, bb) (sol t)) vid1
     (dterm-sem I ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))
      (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0)))ODE) (ab, bb) (sol t)))
    have bigEx:(∃ν sol t. ((ab, bb), ?myω) = (ν, mk-v I (OProd (OSing vid1 ($f
fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))ODE) ν (sol t)) ∧
                  0 ≤ t ∧
                  (sol solves-ode (λ-. ODE-sem I (OProd (OSing vid1 ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0)))ODE))) {0..t}
                   {x. Predicates I vid2
                       (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
                             (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1
then trm.Var vid1 else Const 0)))ODE) ν x))} ∧
                  sol 0 = fst ν)
      apply(rule exI[where x=(ab, bb)])
      apply(rule exI[where x=sol])
      apply(rule exI[where x=t])
      apply(rule conjI)
       apply(rule refl)
      apply(rule conjI)
       apply(rule t)
      apply(rule conjI)
       using sol apply blast
      by (rule sol0)
    have bigRes:?myω ∈ fml-sem I (Pc pid1) using bigAll bigEx by blast
    have notin1:Inl vid1 ∉ BVO ODE using disj by auto
    have notin2:Inr vid1 ∉ BVO ODE using disj by auto
    have ODE-sem:ODE-sem I ODE (sol t) $ vid1 = 0
     using ODE-zero notin1 notin2
     by blast
    have vec-eq:(χ i. sterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
```

```
(sol t)) =
          (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
           (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1
else Const 0)))ODE) (ab, bb) (sol t)))
      apply(rule vec-extensionality)
      apply simp
       using mk-v-agree[of I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then
trm.Var vid1 else Const 0)))ODE) (ab, bb) (sol t)]
      by(simp add: Vagree-def)
    have sem-eq:(?myω ∈ fml-sem I (Pc pid1)) = ((repd (mk-v I (OProd (OSing
vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))ODE) (ab, bb) (sol
t)) vid1
     (dterm-sem I ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))
      (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0)))ODE) (ab, bb) (sol t)))) ∈ fml-sem I (Pc pid1))
      apply(rule coincidence-formula)
       subgoal by simp
      subgoal by (rule Iagree-refl)
       using mk-v-agree[of I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then
trm.Var vid1 else Const 0)))ODE) (ab, bb) (sol t)]
      unfolding Vagree-def
      apply simp
      apply(erule conjE)+
      apply(erule allE[where x=vid1])+
      apply(simp add: ODE-sem)
      using vec-eq by simp
    show repd (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0)))ODE) (ab, bb) (sol t)) vid1
     (dterm-sem I ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))
      (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0)))ODE) (ab, bb) (sol t)))
    ∈ fml-sem I (Pc pid1)
      using bigRes sem-eq by blast
  next
    fix I::('sf,'sc,'sz)interp
    and aa ba ab bb sol
    and t::real
    assume good-interp:is-interp I
    assume all:∀ω. (∃ν sol t. ((ab, bb), ω) = (ν, mk-v I (OProd (OSing vid1 ($f
fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))ODE) ν (sol t)) ∧
                       0 ≤ t ∧
                        (sol solves-ode (λ-. ODE-sem I (OProd (OSing vid1 ($f fid1
(λi. if i = vid1 then trm.Var vid1 else Const 0)))ODE))) {0..t}
                       {x. Predicates I vid2
                          (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const
0)
                                (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1
then trm.Var vid1 else Const 0)))ODE) ν x))} ∧
                       sol 0 = fst ν) ⟶
```

$(\forall \omega'.\ \omega' = repd\ \omega\ vid1\ (dterm\text{-}sem\ I\ (\$f\ fid1\ (\lambda i.\ if\ i = vid1\ then\ trm.Var\ vid1\ else\ Const\ 0))\ \omega) \longrightarrow \omega' \in fml\text{-}sem\ I\ (Pc\ pid1))$

let *?myω* = *mk-v I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*)

assume *t*:*0* ≤ *t*

assume *aaba*:(*aa*, *ba*) = *mk-v I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*)

assume *sol*:
(*sol solves-ode* (*λ-. ODE-sem I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*))) {*0..t*}
{*x. Predicates I vid2*
(*χ i. dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) *x*))}

assume *sol0*:*sol 0* = *fst* (*ab*, *bb*)

have *bigEx*:(∃ *ν sol t.* ((*ab*, *bb*), *?myω*) = (*ν*, *mk-v I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*) *ν* (*sol t*)) ∧
0 ≤ *t* ∧
(*sol solves-ode* (*λ-. ODE-sem I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*))) {*0..t*}
{*x. Predicates I vid2*
(*χ i. dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*) *ν x*))} ∧
sol 0 = *fst ν*)

apply(*rule exI*[**where** *x*=(*ab*, *bb*)])
apply(*rule exI*[**where** *x*=*sol*])
apply(*rule exI*[**where** *x*=*t*])
apply(*rule conjI*)
apply(*rule refl*)
apply(*rule conjI*)
apply(*rule t*)
apply(*rule conjI*)
using *sol sol0* **by**(*blast*)+

have *rep-sem-eq*:*repd* (*mk-v I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))
(*mk-v I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*))) ∈ *fml-sem I* (*Pc pid1*)
= (*repd ?myω vid1* (*dterm-sem I* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)) *?myω*) ∈ *fml-sem I* (*Pc pid1*))

apply(*rule coincidence-formula*)
subgoal by *simp*
subgoal by (*rule Iagree-refl*)
by(*simp add*: *Vagree-def*)

have *notin1*:*Inl vid1* ∉ *BVO ODE* **using** *disj* **by** *auto*
have *notin2*:*Inr vid1* ∉ *BVO ODE* **using** *disj* **by** *auto*

have *ODE-sem*:*ODE-sem I ODE* (*sol t*) $ *vid1* = *0*
using *ODE-zero notin1 notin2*
by *blast*
have *vec-eq*:
(*χ i. dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OProd* (*OSing vid1* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*))) =
(*χ i. sterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*) (*sol t*))
apply(*rule vec-extensionality*)
using *mk-v-agree*[*of I* (*OProd* (*OSing vid1* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*)]
by (*simp add*: *Vagree-def*)
have *sem-eq*:
(*repd ?myω vid1* (*dterm-sem I* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)) *?myω*) ∈ *fml-sem I* (*Pc pid1*))
= (*mk-v I* (*OProd* (*OSing vid1* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*) ∈ *fml-sem I* (*Pc pid1*))
apply(*rule coincidence-formula*)
subgoal by *simp*
subgoal by (*rule Iagree-refl*)
using *mk-v-agree*[*of I* (*OProd* (*OSing vid1* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*)]
unfolding *Vagree-def* **apply** *simp*
apply(*erule conjE*)+
apply(*erule allE*[**where** *x*=*vid1*])+
by (*simp add*: *ODE-sem vec-eq*)
have *some-sem*:*repd* (*mk-v I* (*OProd* (*OSing vid1* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*)) *vid1*
(*dterm-sem I* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*))
(*mk-v I* (*OProd* (*OSing vid1* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*))) ∈ *fml-sem I* (*Pc pid1*)
using *rep-sem-eq*
using *all bigEx* **by** *blast*
have *bigImp*:(∀ *ω′*. *ω′* = *repd ?myω vid1* (*dterm-sem I* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)) *?myω*) ⟶ *ω′* ∈ *fml-sem I* (*Pc pid1*))
apply(*rule allI*)
apply(*rule impI*)
apply *auto*
using *some-sem* **by** *auto*
have *fml-sem*:*repd ?myω vid1* (*dterm-sem I* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)) *?myω*) ∈ *fml-sem I* (*Pc pid1*)
using *sem-eq bigImp* **by** *blast*
show *mk-v I* (*OProd* (*OSing vid1* ($*f fid1* (*λi. if i* = *vid1 then trm.Var vid1 else Const 0*)))*ODE*) (*ab*, *bb*) (*sol t*) ∈ *fml-sem I* (*Pc pid1*)
using *fml-sem sem-eq* **by** *blast*
qed
qed

lemma *DC-valid*:*valid DCaxiom*

proof (*auto simp only*: *fml-sem.simps prog-sem.simps DCaxiom-def valid-def iff-sem impl-sem box-sem*, *auto*)
fix *I*::(*'sf*,*'sc*,*'sz*) *interp* **and** *aa ba bb sol t*
assume *is-interp I*
and *all3*:∀ *a b*. (∃ *sola*. *sol 0* = *sola 0* ∧
(∃ *t*. (*a*, *b*) = *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) (*sola t*) ∧
0 ≤ *t* ∧ (*sola solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*t*} {*x*. *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) *x* ∈ *Contexts I pid1 UNIV*})) ⟶
(*a*, *b*) ∈ *Contexts I pid3 UNIV*
and *all2*:∀ *a b*. (∃ *sola*. *sol 0* = *sola 0* ∧
(∃ *t*. (*a*, *b*) = *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) (*sola t*) ∧
0 ≤ *t* ∧ (*sola solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*t*} {*x*. *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) *x* ∈ *Contexts I pid1 UNIV*})) ⟶
(*a*, *b*) ∈ *Contexts I pid2 UNIV*
and *t*:*0* ≤ *t*
and *aaba*:(*aa*, *ba*) = *mk-v I* (*OVar vid1*) (*sol 0*, *bb*) (*sol t*)
and *sol*:(*sol solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*t*}
{*x*. *mk-v I* (*OVar vid1*) (*sol 0*, *bb*) *x* ∈ *Contexts I pid1 UNIV* ∧ *mk-v I* (*OVar vid1*) (*sol 0*, *bb*) *x* ∈ *Contexts I pid3 UNIV*}
from *sol* **have**
sol1:(*sol solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*t*} {*x*. *mk-v I* (*OVar vid1*) (*sol 0*, *bb*) *x* ∈ *Contexts I pid1 UNIV*}
by (*metis* (*mono-tags*, *lifting*) *Collect-mono solves-ode-supset-range*)
from *all2* **have** *all2'*:⋀*v*. (∃ *sola*. *sol 0* = *sola 0* ∧
(∃ *t*. *v* = *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) (*sola t*) ∧
0 ≤ *t* ∧ (*sola solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*t*} {*x*. *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) *x* ∈ *Contexts I pid1 UNIV*})) ⟹
v ∈ *Contexts I pid2 UNIV* **by** *auto*
show *mk-v I* (*OVar vid1*) (*sol 0*, *bb*) (*sol t*) ∈ *Contexts I pid2 UNIV*
apply(*rule all2'*[*of mk-v I* (*OVar vid1*) (*sol 0*, *bb*) (*sol t*)])
apply(*rule exI*[**where** *x*=*sol*])
apply(*rule conjI*)
subgoal by (*rule refl*)
subgoal using *t sol1* **by** *auto*
done
next
fix *I*::(*'sf*,*'sc*,*'sz*) *interp* **and** *aa ba bb sol t*
assume *is-interp I*
and *all3*:∀ *a b*. (∃ *sola*. *sol 0* = *sola 0* ∧
(∃ *t*. (*a*, *b*) = *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) (*sola t*) ∧
0 ≤ *t* ∧ (*sola solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*t*} {*x*. *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) *x* ∈ *Contexts I pid1 UNIV*})) ⟶
(*a*, *b*) ∈ *Contexts I pid3 UNIV*
and *all2*:∀ *a b*. (∃ *sola*. *sol 0* = *sola 0* ∧
(∃ *t*. (*a*, *b*) = *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) (*sola t*) ∧
0 ≤ *t* ∧
(*sola solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*t*}
{*x*. *mk-v I* (*OVar vid1*) (*sola 0*, *bb*) *x* ∈ *Contexts I pid1 UNIV* ∧
mk-v I (*OVar vid1*) (*sola 0*, *bb*) *x* ∈ *Contexts I pid3 UNIV*}))

$\longrightarrow$
$(a, b) \in Contexts\ I\ pid2\ UNIV$
and $t{:}0 \leq t$
and $aaba{:}(aa, ba) = mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ (sol\ t)$
and $sol{:}(sol\ solves\text{-}ode\ (\lambda a.\ ODEs\ I\ vid1))\ \{0..t\}\ \{x.\ mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ x \in Contexts\ I\ pid1\ UNIV\}$
from *all2*
have $all2'{:}\bigwedge v.\ (\exists sola.\ sol\ 0 = sola\ 0 \wedge (\exists t.\ v = mk\text{-}v\ I\ (OVar\ vid1)\ (sola\ 0, bb)\ (sola\ t) \wedge 0 \leq t \wedge (sola\ solves\text{-}ode\ (\lambda a.\ ODEs\ I\ vid1))\ \{0..t\}\ \{x.\ mk\text{-}v\ I\ (OVar\ vid1)\ (sola\ 0, bb)\ x \in Contexts\ I\ pid1\ UNIV \wedge mk\text{-}v\ I\ (OVar\ vid1)\ (sola\ 0, bb)\ x \in Contexts\ I\ pid3\ UNIV\}))$
$\Longrightarrow$
$v \in Contexts\ I\ pid2\ UNIV$
by *auto*
from *all3*
have $all3'{:}\bigwedge v.\ (\exists sola.\ sol\ 0 = sola\ 0 \wedge (\exists t.\ v = mk\text{-}v\ I\ (OVar\ vid1)\ (sola\ 0, bb)\ (sola\ t) \wedge 0 \leq t \wedge (sola\ solves\text{-}ode\ (\lambda a.\ ODEs\ I\ vid1))\ \{0..t\}\ \{x.\ mk\text{-}v\ I\ (OVar\ vid1)\ (sola\ 0, bb)\ x \in Contexts\ I\ pid1\ UNIV\})) \Longrightarrow$
$v \in Contexts\ I\ pid3\ UNIV$
by *auto*
have $inp1{:}\bigwedge s.\ 0 \leq s \Longrightarrow s \leq t \Longrightarrow mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ (sol\ s) \in Contexts\ I\ pid1\ UNIV$
using *sol solves-odeD atLeastAtMost-iff* **by** *blast*
have $inp3{:}\bigwedge s.\ 0 \leq s \Longrightarrow s \leq t \Longrightarrow mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ (sol\ s) \in Contexts\ I\ pid3\ UNIV$
apply(*rule all3'*)
subgoal for *s*
apply(*rule exI* [**where** *x*=*sol*])
apply(*rule conjI*)
subgoal by (*rule refl*)
apply(*rule exI* [**where** *x*=*s*])
apply(*rule conjI*)
subgoal by (*rule refl*)
apply(*rule conjI*)
subgoal by *assumption*
subgoal using *sol* **by** (*meson atLeastatMost-subset-iff order-refl solves-ode-subset*)
done
done
have $inp13{:}\bigwedge s.\ 0 \leq s \Longrightarrow s \leq t \Longrightarrow mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ (sol\ s) \in Contexts\ I\ pid1\ UNIV \wedge mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ (sol\ s) \in Contexts\ I\ pid3\ UNIV$
using *inp1 inp3* **by** *auto*
have $sol13{:}(sol\ solves\text{-}ode\ (\lambda a.\ ODEs\ I\ vid1))\ \{0..t\}$
$\{x.\ mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ x \in Contexts\ I\ pid1\ UNIV \wedge mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0, bb)\ x \in Contexts\ I\ pid3\ UNIV\}$
apply(*rule solves-odeI*)

```
    subgoal using sol by (rule solves-odeD)
   subgoal for s using inp13[of s] by auto
   done
 show mk-v I (OVar vid1) (sol 0, bb) (sol t) ∈ Contexts I pid2 UNIV
  using t sol13 all2'[of mk-v I (OVar vid1) (sol 0, bb) (sol t)] by auto
qed

lemma DS-valid:valid DSaxiom
proof −
 have dsafe:dsafe($f fid1 (λi. Const 0))
  using dsafe-Const by auto
 have osafe:osafe(OSing vid1 (f0 fid1))
  unfolding f0-def empty-def
  using dsafe osafe.intros
  by (simp add: osafe-Sing dfree-Const)
 have fsafe:fsafe(p1 vid2 vid1)
  unfolding p1-def
  apply(rule fsafe-Prop)
  using singleton.simps dsafe-Const by (auto intro: dfree.intros)
 show valid DSaxiom
  apply(auto simp only: DSaxiom-def valid-def Let-def iff-sem impl-sem box-sem)
   apply(auto simp only: fml-sem.simps prog-sem.simps mem-Collect-eq iff-sem
impl-sem box-sem forall-sem)
 proof −
  fix I::('sf,'sc,'sz) interp
    and a b r aa ba
  assume good-interp:is-interp I
  assume allW:∀ω. (∃ν sol t.
          ((a, b), ω) = (ν, mk-v I (OSing vid1 (f0 fid1)) ν (sol t)) ∧
          0 ≤ t ∧
          (sol solves-ode (λ-. ODE-sem I (OSing vid1 (f0 fid1)))) {0..t}
           {x. mk-v I (OSing vid1 (f0 fid1)) ν x ∈ fml-sem I (p1 vid2 vid1)} ∧
           (sol 0) = (fst ν)) ⟶
       ω ∈ fml-sem I (p1 vid3 vid1)
   assume dterm-sem I (Const 0) (repv (a, b) vid2 r) ≤ dterm-sem I (trm.Var
vid2) (repv (a, b) vid2 r)
   hence leq:0 ≤ r by (auto)
   assume ∀ra. repv (repv (a, b) vid2 r) vid3 ra
       ∈ {v. dterm-sem I (Const 0) v ≤ dterm-sem I (trm.Var vid3) v} ∩
         {v. dterm-sem I (trm.Var vid3) v ≤ dterm-sem I (trm.Var vid2) v} ⟶
        Predicates I vid2
       (χ i. dterm-sem I (singleton (Plus (trm.Var vid1) (Times (f0 fid1) (trm.Var
vid3))) i)
              (repv (repv (a, b) vid2 r) vid3 ra))
   hence constraint:∀ra. (0 ≤ ra ∧ ra ≤ r) ⟶
       (repv (repv (a, b) vid2 r) vid3 ra)
      ∈ fml-sem I (Prop vid2 (singleton (Plus (Var vid1) (Times (f0 fid1) (Var
vid3)))))
    using leq by auto
```

```
    assume aaba: (aa, ba) =
     repv (repv (a, b) vid2 r) vid1
      (dterm-sem I (Plus (trm.Var vid1) (Times (f0 fid1) (trm.Var vid2))) (repv
(a, b) vid2 r))
    let ?abba = repv (repd (a, b) vid1 (Functions I fid1 (χ i. 0))) vid1
      (dterm-sem I (Plus (trm.Var vid1) (Times (f0 fid1) (trm.Var vid2))) (repv
(a, b) vid2 r))
    from allW have thisW:(∃ν sol t.
          ((a, b), ?abba) = (ν, mk-v I (OSing vid1 (f0 fid1)) ν (sol t)) ∧
          0 ≤ t ∧
          (sol solves-ode (λ-. ODE-sem I (OSing vid1 (f0 fid1)))) {0..t}
           {x. mk-v I (OSing vid1 (f0 fid1)) ν x ∈ fml-sem I (p1 vid2 vid1)} ∧
           (sol 0) = (fst ν)) ⟶
       ?abba ∈ fml-sem I (p1 vid3 vid1) by blast
    let ?c = Functions I fid1 (χ -. 0)
    let ?sol = (λt. χ i. if i = vid1 then (a $ i) + ?c * t else (a $ i))
     have agrees:Vagree (mk-v I (OSing vid1 (f0 fid1)) (a, b) (?sol r)) (a, b) (−
semBV I (OSing vid1 (f0 fid1)))
  ∧ Vagree (mk-v I (OSing vid1 (f0 fid1)) (a, b) (?sol r))
   (mk-xode I (OSing vid1 (f0 fid1)) (?sol r)) (semBV I (OSing vid1 (f0 fid1)))
       using mk-v-agree[of I (OSing vid1 (f0 fid1)) (a,b) (?sol r)] by auto
    have prereq1a:fst ?abba
     = fst (mk-v I (OSing vid1 (f0 fid1)) (a,b) (?sol r))
      using agrees aaba
      apply (auto simp add: aaba Vagree-def)
       apply (rule vec-extensionality)
       subgoal for i
         apply (cases i = vid1)
        using vne12 agrees Vagree-def apply (auto simp add: aaba f0-def empty-def)
          done
      apply (rule vec-extensionality)
      subgoal for i
        apply (cases i = vid1)
         apply(auto  simp add: f0-def empty-def)
      done
    done
  have prereq1b:snd (?abba) = snd (mk-v I (OSing vid1 (f0 fid1)) (a,b) (?sol r))
    using agrees aaba
    apply (auto simp add: aaba Vagree-def)
    apply (rule vec-extensionality)
    subgoal for i
      apply (cases i = vid1)
       using vne12 agrees Vagree-def apply (auto simp add: aaba f0-def empty-def
)
      done
    done
  have ?abba = mk-v I (OSing vid1 (f0 fid1)) (a,b) (?sol r)
    using prod-eq-iff prereq1a prereq1b by blast
  hence req1:((a, b), ?abba) = ((a, b), mk-v I (OSing vid1 (f0 fid1)) (a,b) (?sol
```

```
r)) by auto
  have sterm-sem I ($f fid1 (λi. Const 0)) b = Functions I fid1 (χ i. 0) by auto
  hence vec-simp:(λa b. χ i. if i = vid1 then sterm-sem I ($f fid1 (λi. Const 0))
b else 0)
      = (λa b. χ i. if i = vid1 then Functions I fid1 (χ i. 0) else 0)
    by (auto simp add: vec-eq-iff cong: if-cong)
  have sub: {0..r} ⊆ UNIV by auto
  have sub2:{x. mk-v I (OSing vid1 (f0 fid1)) (a,b) x ∈ fml-sem I (p1 vid2 vid1)}
⊆ UNIV by auto
  have req3:(?sol solves-ode (λ-. ODE-sem I (OSing vid1 (f0 fid1)))) {0..r}
          {x. mk-v I (OSing vid1 (f0 fid1)) (a,b) x ∈ fml-sem I (p1 vid2 vid1)}
    apply(auto simp add: f0-def empty-def vec-simp)
    apply(rule solves-odeI)
     apply(auto simp only: has-vderiv-on-def has-vector-derivative-def box-sem)
     apply (rule has-derivative-vec[THEN has-derivative-eq-rhs])
      defer
      apply (rule ext)
      apply (subst scaleR-vec-def)
      apply (rule refl)
     apply (auto intro!: derivative-eq-intros)
    — Domain constraint satisfied
    using constraint apply (auto)
    subgoal for t
      apply(erule allE[where x=t])
      apply(auto simp add: p1-def)
    proof −
      have eq:(χ i. dterm-sem I (if i = vid1 then Plus (trm.Var vid1) (Times (f0
fid1) (trm.Var vid3)) else Const 0)
          (χ y. if vid3 = y then t else fst (χ y. if vid2 = y then r else fst (a, b) $
y, b) $ y, b)) =
          (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
            (mk-v I (OSing vid1 ($f fid1 (λi. Const 0))) (a, b)
              (χ i. if i = vid1 then a $ i + Functions I fid1 (χ -. 0) ∗ t else a $ i)))
        using vne12 vne13 mk-v-agree[of I (OSing vid1 ($f fid1 (λi. Const 0))) (a,
b) (χ i. if i = vid1 then a $ i + Functions I fid1 (χ -. 0) ∗ t else a $ i)]
        by (auto simp add: vec-eq-iff f0-def empty-def Vagree-def)
      show 0 ≤ t ⟹
    t ≤ r ⟹
    Predicates I vid2
      (χ i. dterm-sem I (if i = vid1 then Plus (trm.Var vid1) (Times (f0 fid1)
(trm.Var vid3)) else Const 0)
          (χ y. if vid3 = y then t else fst (χ y. if vid2 = y then r else fst (a, b) $
y, b) $ y, b)) ⟹
    Predicates I vid2
     (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
          (mk-v I (OSing vid1 ($f fid1 (λi. Const 0))) (a, b)
            (χ i. if i = vid1 then a $ i + Functions I fid1 (χ -. 0) ∗ t else a $ i)))
       using eq by auto
    qed
```

```
    done
  have req4':?sol 0 = fst (a,b) by (auto simp: vec-eq-iff)
  then have req4:  (?sol 0) = (fst (a,b))
    using VSagree-refl[of a] req4' unfolding VSagree-def by auto
  have inPred:?abba ∈ fml-sem I (p1 vid3 vid1)
    using req1 leq req3 req4 thisW by fastforce
  have sem-eq:?abba ∈ fml-sem I (p1 vid3 vid1) ⟷ (aa,ba) ∈ fml-sem I (p1 vid3
vid1)
    apply (rule coincidence-formula)
      apply (auto simp add: aaba Vagree-def p1-def f0-def empty-def)
    subgoal using Iagree-refl by auto
    done
  from inPred sem-eq have  inPred':(aa,ba) ∈ fml-sem I (p1 vid3 vid1)
    by auto
  — thus by lemma 6 consequence for formulas
  show repv (repv (a, b) vid2 r) vid1
       (dterm-sem I (Plus (trm.Var vid1) (Times (f0 fid1) (trm.Var vid2))) (repv
(a, b) vid2 r))
      ∈ fml-sem I (p1 vid3 vid1)
    using aaba inPred' by (auto)
next
  fix I::('sf,'sc,'sz) interp
  and aa ba ab bb sol
  and t:: real
  assume good-interp:is-interp I
  assume all:
       ∀ r. dterm-sem I (Const 0) (repv (ab, bb) vid2 r) ≤ dterm-sem I (trm.Var
vid2) (repv (ab, bb) vid2 r) ⟶
          (∀ ra. repv (repv (ab, bb) vid2 r) vid3 ra
               ∈ {v. dterm-sem I (Const 0) v ≤ dterm-sem I (trm.Var vid3) v} ∩
                 {v. dterm-sem I (trm.Var vid3) v ≤ dterm-sem I (trm.Var vid2)
v} ⟶
                Predicates I vid2
                 (χ i. dterm-sem I (singleton (Plus (trm.Var vid1) (Times (f0 fid1)
(trm.Var vid3))) i)
                       (repv (repv (ab, bb) vid2 r) vid3 ra))) ⟶

          (∀ ω. ω = repv (repv (ab, bb) vid2 r) vid1
                     (dterm-sem I (Plus (trm.Var vid1) (Times (f0 fid1) (trm.Var
vid2))) (repv (ab, bb) vid2 r)) ⟶
                ω ∈ fml-sem I (p1 vid3 vid1))
  assume t:0 ≤ t
  assume aaba:(aa, ba) = mk-v I (OSing vid1 (f0 fid1)) (ab, bb) (sol t)
  assume sol:(sol solves-ode (λ-. ODE-sem I (OSing vid1 (f0 fid1)))) {0..t}
       {x. mk-v I (OSing vid1 (f0 fid1)) (ab, bb) x ∈ fml-sem I (p1 vid2 vid1)}
  hence constraint:⋀s. s ∈ {0 .. t} ⟹ sol s ∈ {x. mk-v I (OSing vid1 (f0 fid1))
(ab, bb) x ∈ fml-sem I (p1 vid2 vid1)}
    using solves-ode-domainD by fastforce
  — sol 0 = fst (ab, bb)
```

assume *sol0*: (*sol 0*) = (*fst* (*ab*, *bb*))
have *impl*:*dterm-sem I* (*Const 0*) (*repv* (*ab*, *bb*) *vid2 t*) ≤ *dterm-sem I* (*trm.Var vid2*) (*repv* (*ab*, *bb*) *vid2 t*) ⟶
(∀ *ra*. *repv* (*repv* (*ab*, *bb*) *vid2 t*) *vid3 ra*
∈ {*v*. *dterm-sem I* (*Const 0*) *v* ≤ *dterm-sem I* (*trm.Var vid3*) *v*} ∩
{*v*. *dterm-sem I* (*trm.Var vid3*) *v* ≤ *dterm-sem I* (*trm.Var vid2*) *v*} ⟶
Predicates I vid2
(χ *i*. *dterm-sem I* (*singleton* (*Plus* (*trm.Var vid1*) (*Times* (*f0 fid1*) (*trm.Var vid3*))) *i*)
(*repv* (*repv* (*ab*, *bb*) *vid2 t*) *vid3 ra*))) ⟶
(∀ ω. ω = *repv* (*repv* (*ab*, *bb*) *vid2 t*) *vid1*
(*dterm-sem I* (*Plus* (*trm.Var vid1*) (*Times* (*f0 fid1*) (*trm.Var vid2*))) (*repv* (*ab*, *bb*) *vid2 t*)) ⟶
ω ∈ *fml-sem I* (*p1 vid3 vid1*)) **using** *all* **by** *auto*
interpret *ll*:*ll-on-open-it UNIV* (λ-. *ODE-sem I* (*OSing vid1* (*f0 fid1*))) *UNIV 0*
apply(*standard*)
apply(*auto*)
unfolding *local-lipschitz-def f0-def empty-def sterm-sem.simps*
using *gt-ex lipschitz-on-constant* **by** *blast*
have *eq-UNIV*:*ll.existence-ivl 0* (*sol 0*) = *UNIV*
apply(*rule ll.existence-ivl-eq-domain*)
apply(*auto*)
subgoal for *tm tM t*
apply(*unfold f0-def empty-def sterm-sem.simps*)
by(*metis add.right-neutral mult-zero-left order-refl*)
done
— Combine with *flow-usolves-ode* and *equals-flowI* to get uniqueness of solution
let *?f* = (λ-. *ODE-sem I* (*OSing vid1* (*f0 fid1*)))
have *sol-UNIV*: ⋀*t x*. (*ll.flow 0 x usolves-ode ?f from 0*) (*ll.existence-ivl 0 x*) *UNIV*
using *ll.flow-usolves-ode* **by** *auto*
from *sol* **have** *sol'*:
(*sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f0 fid1*)))) {*0..t*} *UNIV*
apply (*rule solves-ode-supset-range*)
by *auto*
from *sol'* **have** *sol''*:⋀*s*. *s* ≥ *0* ⟹ *s* ≤ *t* ⟹ (*sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f0 fid1*)))) {*0..s*} *UNIV*
by (*simp add*: *solves-ode-subset*)
have *sol0-eq*:*sol 0* = *ll.flow 0* (*sol 0*) *0*
using *ll.general.flow-initial-time-if* **by** *auto*
have *isFlow*:⋀*s*. *s* ≥ *0* ⟹ *s* ≤ *t* ⟹ *sol s* = *ll.flow 0* (*sol 0*) *s*
apply(*rule ll.equals-flowI*)
apply(*auto*)
subgoal using *eq-UNIV* **by** *auto*
subgoal using *sol'' closed-segment-eq-real-ivl t* **by** (*auto simp add*: *solves-ode-singleton*)
subgoal using *eq-UNIV sol sol0-eq* **by** *auto*
done

let *?c = Functions I fid1* (χ -. *0*)
let *?sol* = ($\lambda t.$ χ *i. if i = vid1 then* (*ab* $ *i*) + *?c* * *t else* (*ab* $ *i*))
have *vec-simp*:(λa *b.* χ *i. if i = vid1 then sterm-sem I* ($*f fid1* ($\lambda i.$ *Const 0*)) *b else 0*)
= (λa *b.* χ *i. if i = vid1 then Functions I fid1* (χ *i. 0*) *else 0*)
by (*auto simp add*: *vec-eq-iff cong*: *if-cong*)
have *exp-sol*:(*?sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f0 fid1*)))) {*0..t*} *UNIV*
apply(*auto simp add*: *f0-def empty-def vec-simp*)
apply(*rule solves-odeI*)
apply(*auto simp only*: *has-vderiv-on-def has-vector-derivative-def box-sem*)
apply (*rule has-derivative-vec*[*THEN has-derivative-eq-rhs*])
defer
apply (*rule ext*)
apply (*subst scaleR-vec-def*)
apply (*rule refl*)
apply (*auto intro*!: *derivative-eq-intros*)
done
from *exp-sol* **have** *exp-sol'*:$\bigwedge s.$ *s* $\geq$ *0* $\Longrightarrow$ *s* $\leq$ *t* $\Longrightarrow$ (*?sol solves-ode* (λ-. *ODE-sem I* (*OSing vid1* (*f0 fid1*)))) {*0..s*} *UNIV*
by (*simp add*: *solves-ode-subset*)
have *exp-sol0-eq*:*?sol 0 = ll.flow 0* (*?sol 0*) *0*
using *ll.general.flow-initial-time-if* **by** *auto*
have *more-eq*:(χ *i. if i = vid1 then ab* $ *i* + *Functions I fid1* (χ -. *0*) * *0 else ab* $ *i*) = *sol 0*
using *sol0*
apply *auto*
apply(*rule vec-extensionality*)
by(*auto*)
have *exp-isFlow*:$\bigwedge s.$ *s* $\geq$ *0* $\Longrightarrow$ *s* $\leq$ *t* $\Longrightarrow$ *?sol s = ll.flow 0* (*sol 0*) *s*
apply(*rule ll.equals-flowI*)
apply(*auto*)
subgoal using *eq-UNIV* **by** *auto*
defer
subgoal for *s*
using *eq-UNIV* **apply** *auto*
subgoal using *exp-sol exp-sol0-eq more-eq*
apply(*auto*)
done
done
using *exp-sol' closed-segment-eq-real-ivl t* **apply**(*auto*)
by (*simp add*: *solves-ode-singleton*)
have *sol-eq-exp*:$\bigwedge s.$ *s* $\geq$ *0* $\Longrightarrow$ *s* $\leq$ *t* $\Longrightarrow$ *?sol s = sol s*
unfolding *exp-isFlow isFlow* **by** *auto*
then have *sol-eq-exp-t*:*?sol t = sol t*
using *t* **by** *auto*
then have *sol-eq-exp-t'*:*sol t* $ *vid1 = ?sol t* $ *vid1* **by** *auto*
then have *useful*:*?sol t* $ *vid1 = ab* $ *vid1 + Functions I fid1* (χ *i. 0*) * *t*
by *auto*

from *sol-eq-exp-t′ useful* **have** *useful′*:*sol t $ vid1 = ab $ vid1 + Functions I fid1 (χ i. 0) ∗ t*
by *auto*
have *sol-int*:((*ll.flow 0 (sol 0)*) *usolves-ode ?f from 0*) {*0..t*} {*x. mk-v I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) *x* ∈ *fml-sem I* (*p1 vid2 vid1*)}
apply (*rule usolves-ode-subset-range*[*of* (*ll.flow 0* (*sol 0*)) *?f 0* {*0..t*} *UNIV* {*x. mk-v I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) *x* ∈ *fml-sem I* (*p1 vid2 vid1*)}])
subgoal using *eq-UNIV sol-UNIV*[*of* (*sol 0*)] **apply** (*auto*)
apply (*rule usolves-ode-subset*)
using *t* **by**(*auto*)
apply(*auto*)
using *sol* **apply**(*auto dest*!: *solves-ode-domainD*)
subgoal for *xa* **using** *isFlow*[*of xa*] **by**(*auto*)
done
have *thing*:⋀*s. 0* ≤ *s* ⟹ *s* ≤ *t* ⟹ *fst* (*mk-v I* (*OSing vid1* ($*f fid1* (λ*i. Const 0*))) (*ab*, *bb*) (*?sol s*)) $ *vid1* = *ab* $ *vid1* + *Functions I fid1* (χ *i. 0*) ∗ *s*
subgoal for *s*
using *mk-v-agree*[*of I* (*OSing vid1* ($*f fid1* (λ*i. Const 0*))) (*ab*, *bb*) (*?sol s*)] **apply** *auto*
unfolding *Vagree-def* **by** *auto*
done
have *thing′*:⋀*s. 0* ≤ *s* ⟹ *s* ≤ *t* ⟹ *fst* (*mk-v I* (*OSing vid1* ($*f fid1* (λ*i. Const 0*))) (*ab*, *bb*) (*sol s*)) $ *vid1* = *ab* $ *vid1* + *Functions I fid1* (χ *i. 0*) ∗ *s*
subgoal for *s* **using** *thing*[*of s*] *sol-eq-exp*[*of s*] **by** *auto* **done**
have *another-eq*:⋀*i s. 0* ≤ *s* ⟹ *s* ≤ *t* ⟹ *dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) (*sol s*))

= *dterm-sem I* (*if i* = *vid1 then Plus* (*trm.Var vid1*) (*Times* (*f0 fid1*) (*trm.Var vid3*)) *else Const 0*)
(χ *y. if vid3* = *y then s else fst* (χ *y. if vid2* = *y then s else fst* (*ab*, *bb*) $ *y*, *bb*) $ *y*, *bb*)
using *mk-v-agree*[*of I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) (*sol s*)] *vne12 vne23 vne13*
apply(*auto simp add*: *f0-def p1-def empty-def*)
unfolding *Vagree-def* **apply**(*simp add*: *f0-def empty-def*)
subgoal for *s* **using** *thing′* **by** *auto*
done
have *allRa′*:(∀ *ra. repv* (*repv* (*ab*, *bb*) *vid2 t*) *vid3 ra*
∈ {*v. dterm-sem I* (*Const 0*) *v* ≤ *dterm-sem I* (*trm.Var vid3*) *v*} ∩
{*v. dterm-sem I* (*trm.Var vid3*) *v* ≤ *dterm-sem I* (*trm.Var vid2*) *v*}
⟶
Predicates I vid2
(χ *i. dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) (*sol ra*))))
apply(*rule allI*)
subgoal for *ra*
using *mk-v-agree*[*of I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) (*sol ra*)]
vne23 constraint[*of ra*] **apply**(*auto simp add*: *Vagree-def p1-def*)

done
done
have *anotherFact*:$\bigwedge$*ra. 0* $\leq$ *ra* $\Longrightarrow$ *ra* $\leq$ *t* $\Longrightarrow$ (χ *i. if i = vid1 then ab* \$ *i* + *Functions I fid1* (χ *-. 0*) $*$ *ra else ab* \$ *i*) \$ *vid1* =
ab \$ *vid1* + *dterm-sem I* (*f0 fid1*) (χ *y. if vid3 = y then ra else fst* (χ *y. if vid2 = y then t else fst* (*ab*, *bb*) \$ *y*, *bb*) \$ *y*, *bb*) $*$ *ra*
subgoal for *ra*
apply *simp*
apply(*rule disjI2*)
by (*auto simp add*: *f0-def empty-def*)
done
have *thing'*:$\bigwedge$*ra i. 0* $\leq$ *ra* $\Longrightarrow$ *ra* $\leq$ *t* $\Longrightarrow$ *dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*) (*mk-v I* (*OSing vid1* (\$*f fid1* (λ*i. Const 0*))) (*ab*, *bb*) (*sol ra*))
= *dterm-sem I* (*if i = vid1 then Plus* (*trm.Var vid1*) (*Times* (*f0 fid1*) (*trm.Var vid3*)) *else Const 0*)
(χ *y. if vid3 = y then ra else fst* (χ *y. if vid2 = y then t else fst* (*ab*, *bb*) \$ *y*, *bb*) \$ *y*, *bb*)
subgoal for *ra i*
using *vne12 vne13 mk-v-agree*[*of I OSing vid1* (\$*f fid1* (λ*i. Const 0*)) (*ab*,*bb*) (*sol ra*)]
apply (*auto*)
unfolding *Vagree-def* **apply**(*safe*)
apply(*erule allE*[**where** *x*=*vid1*])+
using *sol-eq-exp*[*of ra*] *anotherFact*[*of ra*] **by** *auto*
done
have *allRa*:($\forall$ *ra. repv* (*repv* (*ab*, *bb*) *vid2 t*) *vid3 ra*
$\in$ {*v. dterm-sem I* (*Const 0*) *v* $\leq$ *dterm-sem I* (*trm.Var vid3*) *v*} $\cap$
{*v. dterm-sem I* (*trm.Var vid3*) *v* $\leq$ *dterm-sem I* (*trm.Var vid2*) *v*}
$\longrightarrow$
Predicates I vid2
(χ *i. dterm-sem I* (*singleton* (*Plus* (*trm.Var vid1*) (*Times* (*f0 fid1*) (*trm.Var vid3*))) *i*)
(*repv* (*repv* (*ab*, *bb*) *vid2 t*) *vid3 ra*)))
apply(*rule allI*)
subgoal for *ra*
using *mk-v-agree*[*of I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) (*sol ra*)]
vne23 constraint[*of ra*] **apply**(*auto simp add*: *Vagree-def p1-def*)
using *sol-eq-exp*[*of ra*] **apply** (*auto simp add*: *f0-def empty-def Vagree-def vec-eq-iff*)
using *thing'* **by** *auto*
done
have *fml3*:$\bigwedge$*ra. 0* $\leq$ *ra* $\Longrightarrow$ *ra* $\leq$ *t* $\Longrightarrow$
($\forall$ ω. ω = *repv* (*repv* (*ab*, *bb*) *vid2 t*) *vid1*
(*dterm-sem I* (*Plus* (*trm.Var vid1*) (*Times* (*f0 fid1*) (*trm.Var vid2*))) (*repv* (*ab*, *bb*) *vid2 t*)) $\longrightarrow$
ω $\in$ *fml-sem I* (*p1 vid3 vid1*))
using *impl allRa* **by** *auto*
have *someEq*:(χ *i. dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(χ *y. if vid1 = y then* (*if vid2 = vid1 then t else fst* (*ab*, *bb*) \$ *vid1*) +

Functions I fid1 (χ *i. 0*) $*$ *t*
else fst (χ *y. if vid2* $=$ *y then t else fst* (*ab*, *bb*) $ *y*, *bb*) $ *y*, *bb*))
$=$ (χ *i. dterm-sem I* (*if i* $=$ *vid1 then trm.Var vid1 else Const 0*) (*mk-v I* (*OSing vid1* ($f *fid1* ($\lambda$*i. Const 0*))) (*ab*, *bb*) (*sol t*)))
apply(*rule vec-extensionality*)
using *vne12 sol-eq-exp t thing* **by** *auto*
show *mk-v I* (*OSing vid1* (*f0 fid1*)) (*ab*, *bb*) (*sol t*) $\in$ *fml-sem I* (*p1 vid3 vid1*)
using *mk-v-agree*[*of I OSing vid1* (*f0 fid1*) (*ab*, *bb*) *sol t*] *fml3*[*of t*]
unfolding *f0-def p1-def empty-def Vagree-def*
using *someEq* **by**(*auto simp add*: *sol-eq-exp-t' t vec-extensionality vne12*)
qed qed

lemma *MVT0-within*:
fixes *f* ::*real* $\Rightarrow$ *real*
and *f'*::*real* $\Rightarrow$ *real* $\Rightarrow$ *real*
and *s t* :: *real*
assumes *f'*:$\bigwedge$*x. x* $\in$ {*0..t*} $\Longrightarrow$ (*f has-derivative* (*f' x*)) (*at x within* {*0..t*})
assumes *geq'*:$\bigwedge$*x. x* $\in$ {*0..t*} $\Longrightarrow$ *f' x s* $\geq$ *0*
assumes *int-s*:*s* $>$ *0* $\wedge$ *s* $\leq$ *t*
assumes *t*: *0* $<$ *t*
shows *f s* $\geq$ *f 0*
proof $-$
have *f 0* $+$ *0* $\leq$ *f s*
apply (*rule Lib.MVT-ivl'*[*OF f'*, *of 0 s 0*])
subgoal for *x* **by** *assumption*
subgoal for *x* **using** *geq'* **by** *auto*
using *t int-s t* **apply** *auto*
subgoal for *x*
by (*metis int-s mult.commute mult.right-neutral order.trans mult-le-cancel-left-pos*)
done
then show *?thesis* **by** *auto*
qed

lemma *MVT'*:
fixes *f g* ::*real* $\Rightarrow$ *real*
fixes *f' g'*::*real* $\Rightarrow$ *real* $\Rightarrow$ *real*
fixes *s t* ::*real*
assumes *f'*:$\bigwedge$*s. s* $\in$ {*0..t*} $\Longrightarrow$ (*f has-derivative* (*f' s*)) (*at s within* {*0..t*})
assumes *g'*:$\bigwedge$*s. s* $\in$ {*0..t*} $\Longrightarrow$ (*g has-derivative* (*g' s*)) (*at s within* {*0..t*})
assumes *geq'*:$\bigwedge$*x. x* $\in$ {*0..t*} $\Longrightarrow$ *f' x s* $\geq$ *g' x s*
assumes *geq0*:*f 0* $\geq$ *g 0*
assumes *int-s*:*s* $>$ *0* $\wedge$ *s* $\leq$ *t*
assumes *t*:*t* $>$ *0*
shows *f s* $\geq$ *g s*
proof $-$
let *?h* $=$ (λ*x. f x* $-$ *g x*)
let *?h'* $=$ (λ*s x. f' s x* $-$ *g' s x*)
have *?h s* $\geq$ *?h 0*

apply(*rule MVT0-within*[*of t ?h ?h′ s*])
subgoal for *s* **using** *f′*[*of s*] *g′*[*of s*] **by** *auto*
subgoal for *sa* **using** *geq′*[*of sa*] **by** *auto*
subgoal using *int-s* **by** *auto*
subgoal using *t* **by** *auto*
done
then show *?thesis* **using** *geq0* **by** *auto*
qed

lemma *MVT′-gr*:
fixes *f g* ::*real* ⇒ *real*
fixes *f′ g′*::*real* ⇒ *real* ⇒ *real*
fixes *s t* ::*real*
assumes *f′*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ (*f has-derivative* (*f′ s*)) (*at s within* {*0*..*t*})
assumes *g′*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ (*g has-derivative* (*g′ s*)) (*at s within* {*0*..*t*})
assumes *geq′*:⋀*x*. *x* ∈ {*0*..*t*} ⟹ *f′ x s* ≥ *g′ x s*
assumes *geq0*:*f 0* > *g 0*
assumes *int-s*:*s* > *0* ∧ *s* ≤ *t*
assumes *t*:*t* > *0*
shows *f s* > *g s*
proof −
let *?h* = (λ*x*. *f x* − *g x*)
let *?h′* = (λ*s x*. *f′ s x* − *g′ s x*)
have *?h s* ≥ *?h 0*
apply(*rule MVT0-within*[*of t ?h ?h′ s*])
subgoal for *s* **using** *f′*[*of s*] *g′*[*of s*] **by** *auto*
subgoal for *sa* **using** *geq′*[*of sa*] **by** *auto*
subgoal using *int-s* **by** *auto*
subgoal using *t* **by** *auto*
done
then show *?thesis* **using** *geq0* **by** *auto*
qed

lemma *frech-linear*:
fixes *x ϑ ν ν′ I*
assumes *good-interp*:*is-interp I*
assumes *free*:*dfree ϑ*
shows *x* ∗ *frechet I ϑ ν ν′* = *frechet I ϑ ν* (*x* ∗*R ν′*)
using *frechet-linear*[*OF good-interp free*]
by (*simp add*: *linear-simps*)

lemma *rift-in-space-time*:
fixes *sol I ODE ψ ϑ t s b*
assumes *good-interp*:*is-interp I*
assumes *free*:*dfree ϑ*
assumes *osafe*:*osafe ODE*
assumes *sol*:(*sol solves-ode* (λ- *ν′*. *ODE-sem I ODE ν′*)) {*0*..*t*}
{*x*. *mk-v I ODE* (*sol 0*, *b*) *x* ∈ *fml-sem I ψ*}
assumes *FVT*:*FVT ϑ* ⊆ *semBV I ODE*

assumes *ivl*:$s \in \{0..t\}$
shows $((\lambda t.\ sterm\text{-}sem\ I\ \vartheta\ (fst\ (mk\text{-}v\ I\ ODE\ (sol\ 0,\ b)\ (sol\ t))))$
— This is Frechet derivative, so equivalent to:
— *has-real-derivative frechet I* ϑ (*fst*((*mk-v I ODE* (*sol 0*, *b*) (*sol s*)))) (*snd* (*mk-v I ODE* (*sol 0*, *b*) (*sol s*))))) (*at s within* $\{0..t\}$)
has-derivative $(\lambda t'.\ t' * frechet\ I\ \vartheta\ (fst((mk\text{-}v\ I\ ODE\ (sol\ 0,\ b)\ (sol\ s))))\ (snd$ (*mk-v I ODE* (*sol 0*, *b*) (*sol s*))))) (*at s within* $\{0..t\}$)
proof −
let $?\varphi = (\lambda t.\ (mk\text{-}v\ I\ ODE\ (sol\ 0,\ b)\ (sol\ t)))$
let $?\varphi s = (\lambda t.\ fst\ (?\varphi\ t))$
have *sol-deriv*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow (sol\ has\text{-}derivative\ (\lambda xa.\ xa *_R ODE\text{-}sem\ I$ *ODE* (*sol s*))) (*at s within* $\{0..t\}$)
using *sol* **apply** *simp*
apply (*drule solves-odeD*(*1*))
unfolding *has-vderiv-on-def has-vector-derivative-def*
by *auto*
have *sol-dom*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow ?\varphi\ s \in fml\text{-}sem\ I\ \psi$
using *sol* **apply** *simp*
apply (*drule solves-odeD*(*2*))
by *auto*
let $?h = (\lambda t.\ sterm\text{-}sem\ I\ \vartheta\ (?\varphi s\ t))$
let $?g = (\lambda \nu.\ sterm\text{-}sem\ I\ \vartheta\ \nu)$
let $?f = ?\varphi s$
let $?f' = (\lambda t'.\ t' *_R (\chi\ i.\ if\ i \in ODE\text{-}vars\ I\ ODE\ then\ ODE\text{-}sem\ I\ ODE\ (sol\ s)$ \$ *i else 0*))
let $?g' = (frechet\ I\ \vartheta\ (?\varphi s\ s))$
have *heq*:$?h = ?g \circ ?f$ **by** (*auto*)
have *fact1*:$\bigwedge i.\ i \in ODE\text{-}vars\ I\ ODE \Longrightarrow (\lambda t.\ ?\varphi s(t)\ \$\ i) = (\lambda t.\ sol\ t\ \$\ i)$
subgoal for *i*
apply(*rule ext*)
subgoal for *t*
using *mk-v-agree*[*of I ODE* (*sol 0*, *b*) *sol t*]
unfolding *Vagree-def* **by** *auto*
done done
have *fact2*:$\bigwedge i.\ i \in ODE\text{-}vars\ I\ ODE \Longrightarrow (\lambda t.\ if\ i \in ODE\text{-}vars\ I\ ODE\ then$ *ODE-sem I ODE* (*sol t*) \$ *i else 0*) = $(\lambda t.\ ODE\text{-}sem\ I\ ODE\ (sol\ t)\ \$\ i)$
subgoal for *i*
apply(*rule ext*)
subgoal for *t*
using *mk-v-agree*[*of I ODE* (*sol 0*, *b*) *sol t*]
unfolding *Vagree-def* **by** *auto*
done done
have *fact3*:$\bigwedge i.\ i \in (-ODE\text{-}vars\ I\ ODE) \Longrightarrow (\lambda t.\ ?\varphi s(t)\ \$\ i) = (\lambda t.\ sol\ 0\ \$\ i)$
subgoal for *i*
apply(*rule ext*)
subgoal for *t*
using *mk-v-agree*[*of I ODE* (*sol 0*, *b*) *sol t*]
unfolding *Vagree-def* **by** *auto*
done done

have *fact4*:$\bigwedge i.\ i \in (-ODE\text{-}vars\ I\ ODE) \Longrightarrow (\lambda t.\ if\ i \in ODE\text{-}vars\ I\ ODE\ then\ ODE\text{-}sem\ I\ ODE\ (sol\ t)\ \$\ i\ else\ 0) = (\lambda t.\ 0)$
subgoal for *i*
apply(*rule ext*)
subgoal for *t*
using *mk-v-agree*[*of I ODE* (*sol 0*, *b*) *sol t*]
unfolding *Vagree-def* **by** *auto*
done done
have *some-eq*:$(\lambda v'.\ \chi\ i.\ v' *_R ODE\text{-}sem\ I\ ODE\ (sol\ s)\ \$\ i) = (\lambda v'.\ v' *_R ODE\text{-}sem\ I\ ODE\ (sol\ s))$
apply(*rule ext*)
apply(*rule vec-extensionality*)
by *auto*
have *some-sol*:$(sol\ has\text{-}derivative\ (\lambda v'.\ v' *_R ODE\text{-}sem\ I\ ODE\ (sol\ s)))\ (at\ s\ within\ \{0..t\})$
using *sol ivl* **unfolding** *solves-ode-def has-vderiv-on-def has-vector-derivative-def* **by** *auto*
have *some-eta*:$(\lambda t.\ \chi\ i.\ sol\ t\ \$\ i) = sol$ **by** (*rule ext*, *rule vec-extensionality*, *auto*)
have *ode-deriv*:$\bigwedge i.\ i \in ODE\text{-}vars\ I\ ODE \Longrightarrow$
$((\lambda t.\ sol\ t\ \$\ i)\ has\text{-}derivative\ (\lambda\ v'.\ v' *_R ODE\text{-}sem\ I\ ODE\ (sol\ s)\ \$\ i))\ (at\ s\ within\ \{0..t\})$
subgoal for *i*
apply(*rule has-derivative-proj*)
using *some-eq some-sol some-eta* **by** *auto*
done
have *eta*:$(\lambda t.\ (\chi\ i.\ ?f\ t\ \$\ i)) = ?f$ **by**(*rule ext*, *rule vec-extensionality*, *auto*)
have *eta-esque*:$(\lambda t'.\ \chi\ i.\ t' * (if\ i \in ODE\text{-}vars\ I\ ODE\ then\ ODE\text{-}sem\ I\ ODE\ (sol\ s)\ \$\ i\ else\ 0)) =$
$(\lambda t'.\ t' *_R (\chi\ i.\ if\ i \in ODE\text{-}vars\ I\ ODE\ then\ ODE\text{-}sem\ I\ ODE\ (sol\ s)\ \$\ i\ else\ 0))$
apply(*rule ext* | *rule vec-extensionality*)+
subgoal for *t' i* **by** *auto* **done**
have $((\lambda t.\ (\chi\ i.\ ?f\ t\ \$\ i))\ has\text{-}derivative\ (\lambda t'.\ (\chi\ i.\ ?f'\ t'\ \$\ i)))\ (at\ s\ within\ \{0..t\})$
apply (*rule has-derivative-vec*)
subgoal for *i*
apply(*cases i ∈ ODE-vars I ODE*)
subgoal using *fact1*[*of i*] *fact2*[*of i*] *ode-deriv*[*of i*] **by** *auto*
subgoal using *fact3*[*of i*] *fact4*[*of i*] **by** *auto*
done
done
then have *fderiv*:$(?f\ has\text{-}derivative\ ?f')\ (at\ s\ within\ \{0..t\})$ **using** *eta eta-esque* **by** *auto*
have *gderiv*:$(?g\ has\text{-}derivative\ ?g')\ (at\ (?f\ s)\ within\ ?f\ `\ \{0..t\})$
using *has-derivative-at-withinI*
using *frechet-correctness free good-interp*
by *blast*
have *chain*:$((?g \circ ?f)\ has\text{-}derivative\ (?g' \circ ?f'))\ (at\ s\ within\ \{0..t\})$

```
    using fderiv gderiv diff-chain-within by blast
  let ?coν1 = (fst (mk-v I ODE (sol 0, b) (sol s)), ODE-sem I ODE (fst (mk-v I
ODE (sol 0, b) (sol s))))
  let ?coν2 = (fst (mk-v I ODE (sol 0, b) (sol s)), snd (mk-v I ODE (sol 0, b)
(sol s)))
  have sub-cont:⋀a .a ∉ ODE-vars I ODE ⟹ Inl a ∈ FVT ϑ ⟹ False
    using FVT by auto
  have sub-cont2:⋀a .a ∉ ODE-vars I ODE ⟹ Inr a ∈ FVT ϑ ⟹ False
    using FVT by auto
  have Vagree (mk-v I ODE (sol 0, b) (sol s)) (sol s, b) (Inl ‘ ODE-vars I ODE)
    using mk-v-agree[of I ODE (sol 0, b) sol s]
    unfolding Vagree-def by auto
  let ?co′ν1 = (λx. (fst (mk-v I ODE (sol 0, b) (sol s)), x ∗R (χ i. if i ∈ ODE-vars
I ODE then ODE-sem I ODE (sol s) $ i else 0)))
  let ?co′ν2 = (λx. (fst (mk-v I ODE (sol 0, b) (sol s)), x ∗R snd (mk-v I ODE
(sol 0, b) (sol s))))
  have co-agree-sem:⋀s. Vagree (?co′ν1 s) (?co′ν2 s) (semBV I ODE)
    subgoal for sa
      using mk-v-agree[of I ODE (sol 0, b) sol s]
      unfolding Vagree-def by auto
    done
  have co-agree-help:⋀s. Vagree (?co′ν1 s) (?co′ν2 s) (FVT ϑ)
    using agree-sub[OF FVT co-agree-sem] by auto
  have co-agree′:⋀s. Vagree (?co′ν1 s) (?co′ν2 s) (FVDiff ϑ)
    subgoal for s
      using mk-v-agree[of I ODE (sol 0, b) sol s]
      unfolding Vagree-def apply auto
      subgoal for i x
        apply(cases x)
        subgoal for a
          apply(cases a ∈ ODE-vars I ODE)
         by (simp | metis (no-types, lifting) FVT ODE-vars-lr Vagree-def mk-v-agree
mk-xode.elims subsetD snd-conv)+
        subgoal for a
          apply(cases a ∈ ODE-vars I ODE)
        by (simp | metis (no-types, lifting) FVT Vagree-def mk-v-agree mk-xode.elims
subsetD snd-conv)+
        done
      subgoal for i x
        apply(cases x)
        subgoal for a
          apply(cases a ∈ ODE-vars I ODE)
            using FVT ODE-vars-lr Vagree-def mk-v-agree mk-xode.elims subsetD
snd-conv
           by auto
        subgoal for a
          apply(cases a ∈ ODE-vars I ODE)
           apply(erule allE[where x=i])+
            using FVT ODE-vars-lr Vagree-def mk-v-agree mk-xode.elims subsetD
```

```
snd-conv
          by auto
        done
      done
    done
  have heq'':(?g' ∘ ?f') = (λt'. t' *R frechet I ϑ (?φs s) (snd (?φ s)))
    using mk-v-agree[of I ODE (sol 0, b) sol s]
    unfolding comp-def
    apply auto
    apply(rule ext | rule vec-extensionality)+
    subgoal for x
      using frech-linear[of I ϑ x (fst (mk-v I ODE (sol 0, b) (sol s))) (snd (mk-v I
ODE (sol 0, b) (sol s))), OF good-interp free]
       using coincidence-frechet[OF free, of (?co'ν1 x) (?co'ν2 x), OF co-agree'[of
x], of I]
      by auto
    done
  have ((?g ∘ ?f) has-derivative (?g' ∘ ?f')) (at s within {0..t})
    using chain by auto
  then have ((?g ∘ ?f) has-derivative (λt'. t' * frechet I ϑ (?φs s) (snd (?φ s))))
(at s within {0..t})
    using heq'' by auto
  then have result:((λt. sterm-sem I ϑ (?φs t))  has-derivative (λt. t * frechet I
ϑ (?φs s) (snd (?φ s)))) (at s within {0..t})
    using heq by auto
  then show ?thesis by auto
qed

lemma dterm-sterm-dfree:
   dfree ϑ ⟹ (⋀ν ν'. sterm-sem I ϑ ν = dterm-sem I ϑ (ν, ν'))
  by(induction rule: dfree.induct, auto)

— g(x)≥ h(x) →  [x'=f(x), c & p(x)](g(x)' ≥ h(x)') → [x'=f(x), c]g(x) ≥ h(x)
lemma DIGeq-valid:valid DIGeqaxiom
  unfolding DIGeqaxiom-def
  apply(unfold DIGeqaxiom-def valid-def impl-sem iff-sem)
  apply(auto)
  proof −
    fix I::('sf,'sc,'sz) interp and  b aa ba
      and sol::real ⇒ 'sz simple-state
      and t::real
    let ?ODE = OVar vid1
    let ?φ = (λt. mk-v I (?ODE) (sol 0, b) (sol t))
    assume t:0 ≤ t
      and sol:(sol solves-ode (λa. ODEs I vid1)) {0..t}
       {x. Predicates I vid1 (χ i. dterm-sem I (empty i) (mk-v I ?ODE (sol 0, b)
x))}
      and notin: ¬(Predicates I vid1 (χ i. dterm-sem I (empty i) (sol 0, b)))
    have fsafe:fsafe (Prop vid1 empty) by (auto simp add: empty-def)
```

```
    from sol have Predicates I vid1 (χ i. dterm-sem I (empty i) (?φ 0))
      using t solves-ode-domainD[of sol (λa. ODEs I vid1) {0..t}] by auto
    then have incon:Predicates I vid1 (χ i. dterm-sem I (empty i) ((sol 0, b)))
      using coincidence-formula[OF fsafe Iagree-refl[of I], of (sol 0, b) ?φ 0]
      unfolding Vagree-def by (auto simp add: empty-def)
     show dterm-sem I (f1 fid2 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t)) ≤
dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
      using notin incon by auto
  next
    fix I::('sf,'sc,'sz) interp and b aa ba
      and sol::real ⇒ 'sz simple-state
      and t::real
    let ?ODE = OVar vid1
    let ?φ = (λt. mk-v I (?ODE) (sol 0, b) (sol t))
    assume t:0 ≤ t
      and sol:(sol solves-ode (λa. ODEs I vid1)) {0..t}
       {x. Predicates I vid1 (χ i. dterm-sem I (empty i) (mk-v I ?ODE (sol 0, b)
x))}
      and notin: ¬(Predicates I vid1 (χ i. dterm-sem I (empty i) (sol 0, b)))
    have fsafe:fsafe (Prop vid1 empty) by (auto simp add: empty-def)
    from sol have Predicates I vid1 (χ i. dterm-sem I (empty i) (?φ 0))
      using t solves-ode-domainD[of sol (λa. ODEs I vid1) {0..t}] by auto
    then have incon:Predicates I vid1 (χ i. dterm-sem I (empty i) ((sol 0, b)))
      using coincidence-formula[OF fsafe Iagree-refl[of I], of (sol 0, b) ?φ 0]
      unfolding Vagree-def by (auto simp add: empty-def)
     show dterm-sem I (f1 fid2 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t)) ≤
dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
      using notin incon by auto
  next
    fix I::('sf,'sc,'sz) interp and b aa ba
      and sol::real ⇒ 'sz simple-state
      and t::real
    let ?ODE = OVar vid1
    let ?φ = (λt. mk-v I (?ODE) (sol 0, b) (sol t))
    assume t:0 ≤ t
    assume sol:(sol solves-ode (λa. ODEs I vid1)) {0..t}
       {x. Predicates I vid1 (χ i. dterm-sem I (local.empty i) (mk-v I (OVar vid1)
(sol 0, b) x))}
    assume notin:¬ Predicates I vid1 (χ i. dterm-sem I (local.empty i) (sol 0, b))
    have fsafe:fsafe (Prop vid1 empty) by (auto simp add: empty-def)
    from sol have Predicates I vid1 (χ i. dterm-sem I (empty i) (?φ 0))
      using t solves-ode-domainD[of sol (λa. ODEs I vid1) {0..t}] by auto
    then have incon:Predicates I vid1 (χ i. dterm-sem I (empty i) ((sol 0, b)))
      using coincidence-formula[OF fsafe Iagree-refl[of I], of (sol 0, b) ?φ 0]
      unfolding Vagree-def by (auto simp add: empty-def)
    show dterm-sem I (f1 fid2 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
      ≤ dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
      using incon notin by auto
next
```

fix I::$('sf, 'sc, 'sz)$ *interp* **and** b *aa ba*
and *sol*::*real* ⇒ *'sz simple-state*
and t::*real*
let *?ODE* = *OVar vid1*
let $?\varphi$ = $(\lambda t.$ *mk-v I* (*?ODE*) (*sol 0*, *b*) (*sol t*))
assume *good-interp*:*is-interp I*
assume *aaba*:(*aa*, *ba*) = *mk-v I* (*OVar vid1*) (*sol 0*, *b*) (*sol t*)
assume *t*:*0* ≤ *t*
assume *sol*:(*sol solves-ode* (λ*a. ODEs I vid1*)) {*0*..*t*}
{*x. Predicates I vid1* (χ *i. dterm-sem I* (*local.empty i*) (*mk-v I* (*OVar vid1*) (*sol 0*, *b*) *x*))}
assume *box*:∀ *a ba.* (∃ *sola. sol 0* = *sola 0* ∧
(∃ *t.* (*a*, *ba*) = *mk-v I* (*OVar vid1*) (*sola 0*, *b*) (*sola t*) ∧
0 ≤ *t* ∧
(*sola solves-ode* (λ*a. ODEs I vid1*)) {*0*..*t*}
{*x. Predicates I vid1*
(χ *i. dterm-sem I* (*local.empty i*) (*mk-v I* (*OVar vid1*) (*sola 0*, *b*) *x*))})) ⟶
directional-derivative I (*f1 fid2 vid1*) (*a*, *ba*) ≤ *directional-derivative I* (*f1 fid1 vid1*) (*a*, *ba*)
assume *geq0*:*dterm-sem I* (*f1 fid2 vid1*) (*sol 0*, *b*) ≤ *dterm-sem I* (*f1 fid1 vid1*) (*sol 0*, *b*)
have *free1*:*dfree* ($*f fid2* (λ*i. if i* = *vid1 then trm.Var vid1 else Const 0*))
by (*auto intro*: *dfree.intros*)
have *free2*:*dfree* ($*f fid1* (λ*i. if i* = *vid1 then trm.Var vid1 else Const 0*))
by (*auto intro*: *dfree.intros*)
from *geq0*
have *geq0'*:*sterm-sem I* (*f1 fid2 vid1*) (*sol 0*) ≤ *sterm-sem I* (*f1 fid1 vid1*) (*sol 0*)
unfolding *f1-def* **using** *dterm-sterm-dfree*[*OF free1*, *of I sol 0 b*] *dterm-sterm-dfree*[*OF free2*, *of I sol 0 b*]
by *auto*
let $?\varphi s$ = λ*x. fst* ($?\varphi$ *x*)
let $?\varphi t$ = λ*x. snd* ($?\varphi$ *x*)
let *?df1* = $(\lambda t.$ *dterm-sem I* (*f1 fid2 vid1*) ($?\varphi$ *t*))
let *?f1* = $(\lambda t.$ *sterm-sem I* (*f1 fid2 vid1*) ($?\varphi s$ *t*))
let *?f1'* = $(\lambda$ *s t'. t'* ∗ *frechet I* (*f1 fid2 vid1*) ($?\varphi s$ *s*) ($?\varphi t$ *s*))
have *dfeq*:*?df1* = *?f1*
apply(*rule ext*)
subgoal for *t*
using *dterm-sterm-dfree*[*OF free1*, *of I* $?\varphi s$ *t snd* ($?\varphi$ *t*)] **unfolding** *f1-def expand-singleton* **by** *auto*
done
have *free3*:*dfree* (*f1 fid2 vid1*) **unfolding** *f1-def* **by** (*auto intro*: *dfree.intros*)
let *?df2* = $(\lambda t.$ *dterm-sem I* (*f1 fid1 vid1*) ($?\varphi$ *t*))
let *?f2* = $(\lambda t.$ *sterm-sem I* (*f1 fid1 vid1*) ($?\varphi s$ *t*))
let *?f2'* = $(\lambda s\ t'\ .\ t'$ ∗ *frechet I* (*f1 fid1 vid1*) ($?\varphi s$ *s*) ($?\varphi t$ *s*))
let *?int* = {*0*..*t*}
have *bluh*:⋀*x i.* (*Functions I i has-derivative* (*THE f'.* ∀ *x.* (*Functions I i*

has-derivative f′ x) (*at x*)) *x*) (*at x*)
using *good-interp* **unfolding** *is-interp-def* **by** *auto*
have *blah*:(*Functions I fid2 has-derivative* (*THE f′*. ∀*x*. (*Functions I fid2 has-derivative f′ x*) (*at x*)) (χ *i*. *if i* = *vid1 then sol t* $ *vid1 else 0*)) (*at* (χ *i*. *if i* = *vid1 then sol t* $ *vid1 else 0*))
using *bluh* **by** *auto*
have *bigEx*:⋀*s*. *s* ∈ {*0*..*t*} ⟹(∃ *sola*. *sol 0* = *sola 0* ∧
(∃ *ta*. (*fst* (*?φ s*),
snd (*?φ s*)) =
mk-v I (*OVar vid1*) (*sola 0*, *b*) (*sola ta*) ∧
0 ≤ *ta* ∧
(*sola solves-ode* (λ*a*. *ODEs I vid1*)) {*0*..*ta*}
{*x*. *Predicates I vid1* (χ *i*. *dterm-sem I* (*local.empty i*) (*mk-v I* (*OVar vid1*) (*sol 0*, *b*) *x*))}))
subgoal for *s*
apply(*rule exI*[**where** *x*=*sol*])
apply(*rule conjI*)
subgoal by (*rule refl*)
apply(*rule exI*[**where** *x*=*s*])
apply(*rule conjI*)
subgoal by *auto*
apply(*rule conjI*)
subgoal by *auto*
using *sol*
using *atLeastAtMost-iff atLeastatMost-subset-iff order-refl solves-ode-on-subset*
by (*metis* (*no-types*, *lifting*) *subsetI*)
done
have *box′*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ *directional-derivative I* (*f1 fid2 vid1*) (*?φs s*, *?φt s*)
≤ *directional-derivative I* (*f1 fid1 vid1*) (*?φs s*, *?φt s*)
subgoal for *s*
using *box*
apply *simp*
apply (*erule allE*[**where** *x*=*?φs s*])
apply (*erule allE*[**where** *x*=*?φt s*])
using *bigEx*[*of s*] **by** *auto*
done
have *dsafe1*:*dsafe* (*f1 fid2 vid1*) **unfolding** *f1-def* **by** (*auto intro*: *dsafe.intros*)
have *dsafe2*:*dsafe* (*f1 fid1 vid1*) **unfolding** *f1-def* **by** (*auto intro*: *dsafe.intros*)
have *agree1*:*Vagree* (*sol 0*, *b*) (*?φ 0*) (*FVT* (*f1 fid2 vid1*))
using *mk-v-agree*[*of I* (*OVar vid1*) (*sol 0*, *b*) *fst* (*?φ 0*)]
unfolding *f1-def Vagree-def expand-singleton*
apply *auto*
by (*metis* (*no-types*, *lifting*) *Compl-iff Vagree-def fst-conv mk-v-agree mk-xode.simps semBV.simps*)
have *agree2*:*Vagree* (*sol 0*, *b*) (*?φ 0*) (*FVT* (*f1 fid1 vid1*))
using *mk-v-agree*[*of I* (*OVar vid1*) (*sol 0*, *b*) *fst* (*?φ 0*)]
unfolding *f1-def Vagree-def expand-singleton*
apply *auto*

by (*metis* (*no-types*, *lifting*) *Compl-iff Vagree-def fst-conv mk-v-agree mk-xode.simps semBV.simps*)
have *sem-eq1*:*dterm-sem I* (*f1 fid2 vid1*) (*sol 0*, *b*) = *dterm-sem I* (*f1 fid2 vid1*) (*?φ 0*)
using *coincidence-dterm*[*OF dsafe1 agree1*] **by** *auto*
then have *sem-eq1'*:*sterm-sem I* (*f1 fid2 vid1*) (*sol 0*) = *sterm-sem I* (*f1 fid2 vid1*) (*?φs 0*)
using *dterm-sterm-dfree*[*OF free1*, *of I sol 0 b*]
dterm-sterm-dfree[*OF free1*, *of I* (*?φs 0*) *snd* (*?φ 0*)]
unfolding *f1-def expand-singleton* **by** *auto*
have *sem-eq2*:*dterm-sem I* (*f1 fid1 vid1*) (*sol 0*, *b*) = *dterm-sem I* (*f1 fid1 vid1*) (*?φ 0*)
using *coincidence-dterm*[*OF dsafe2 agree2*] **by** *auto*
then have *sem-eq2'*:*sterm-sem I* (*f1 fid1 vid1*) (*sol 0*) = *sterm-sem I* (*f1 fid1 vid1*) (*?φs 0*)
using *dterm-sterm-dfree*[*OF free2*, *of I sol 0 b*] *dterm-sterm-dfree*[*OF free2*, *of I* (*?φs 0*) *snd* (*?φ 0*)]
unfolding *f1-def expand-singleton* **by** *auto*
have *good-interp'*:⋀*i x*. (*Functions I i has-derivative* (*THE f'*. ∀ *x*. (*Functions I i has-derivative f' x*) (*at x*)) *x*) (*at x*)
using *good-interp* **unfolding** *is-interp-def* **by** *auto*
have *chain* :
⋀*f f' g g' x s*.
(*f has-derivative f'*) (*at x within s*) ⟹
(*g has-derivative g'*) (*at* (*f x*) *within f ' s*) ⟹ (*g* ∘ *f has-derivative g'* ∘ *f'*) (*at x within s*)
by(*auto intro*: *derivative-intros*)
have *sol1*:(*sol solves-ode* (λ-. *ODE-sem I* (*OVar vid1*))) {*0..t*} {*x*. *mk-v I* (*OVar vid1*) (*sol 0*, *b*) *x* ∈ *fml-sem I* (*Prop vid1 empty*)}
using *sol* **unfolding** *p1-def singleton-def empty-def* **by** *auto*
have *FVTsub1*:*vid1* ∈ *ODE-vars I* (*OVar vid1*) ⟹ *FVT* (*$f fid2* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)) ⊆ *semBV I* ((*OVar vid1*))
apply *auto*
subgoal for *x xa*
apply(*cases xa* = *vid1*)
by *auto*
done
have *FVTsub2*:*vid1* ∈ *ODE-vars I* (*OVar vid1*) ⟹ *FVT* (*$f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)) ⊆ *semBV I* ((*OVar vid1*))
apply *auto*
subgoal for *x xa*
apply(*cases xa* = *vid1*)
by *auto*
done
have *osafe*:*osafe* (*OVar vid1*)
by *auto*
have *deriv1*:⋀*s*. *vid1* ∈ *ODE-vars I* (*OVar vid1*) ⟹ *s* ∈ *?int* ⟹ (*?f1 has-derivative* (*?f1' s*)) (*at s within* {*0..t*})
subgoal for *s*

```
      using  rift-in-space-time[OF good-interp free1 osafe sol1 FVTsub1, of s]
      unfolding f1-def expand-singleton directional-derivative-def
      by blast
    done
    have deriv2:⋀s. vid1 ∈ ODE-vars I (OVar vid1) ⟹ s ∈ ?int ⟹ (?f2
has-derivative (?f2' s)) (at s within {0..t})
    subgoal for s
      using rift-in-space-time[OF good-interp free2 osafe sol1 FVTsub2, of s]
      unfolding f1-def expand-singleton directional-derivative-def
      by blast
    done
   have leq:⋀s . s ∈ ?int ⟹ ?f1' s 1 ≤ ?f2' s 1
    subgoal for s using box'[of s]
      by (simp add: directional-derivative-def)
    done
   have preserve-agree1:vid1 ∉ ODE-vars I (OVar vid1) ⟹ VSagree (sol 0) (fst
(mk-v I (OVar vid1) (sol 0, b) (sol t))) {vid1}
    using mk-v-agree[of I OVar vid1 (sol 0, b) sol t]
    unfolding Vagree-def VSagree-def
    by auto
   have preserve-coincide1:
    vid1 ∉ ODE-vars I (OVar vid1) ⟹
    sterm-sem I (f1 fid2 vid1) (fst (mk-v I (OVar vid1) (sol 0, b) (sol t)))
   = sterm-sem I (f1 fid2 vid1) (sol 0)
    using coincidence-sterm[of (sol 0, b) (mk-v I (OVar vid1) (sol 0, b) (sol t))
f1 fid2 vid1 I]
    preserve-agree1 unfolding VSagree-def Vagree-def f1-def by auto
   have preserve-coincide2:
    vid1 ∉ ODE-vars I (OVar vid1) ⟹
    sterm-sem I (f1 fid1 vid1) (fst (mk-v I (OVar vid1) (sol 0, b) (sol t)))
   = sterm-sem I (f1 fid1 vid1) (sol 0)
    using coincidence-sterm[of (sol 0, b) (mk-v I (OVar vid1) (sol 0, b) (sol t))
f1 fid1 vid1 I]
    preserve-agree1 unfolding VSagree-def Vagree-def f1-def by auto
   have ?f1 t ≤ ?f2 t
    apply(cases t = 0)
     subgoal using geq0' sem-eq1' sem-eq2' by auto
    subgoal
     apply(cases vid1 ∈ ODE-vars I (OVar vid1))
     subgoal
      apply (rule MVT'[OF deriv2 deriv1, of t])
            subgoal by auto
           subgoal by auto
          subgoal for s using deriv2[of s] using leq by auto
         using t leq geq0' sem-eq1' sem-eq2' by auto
     subgoal
      using geq0
      using dterm-sterm-dfree[OF free1, of I sol 0 b]
      using dterm-sterm-dfree[OF free2, of I sol 0 b]
```

```
        using preserve-coincide1 preserve-coincide2
        by(simp add: f1-def)
      done
    done
  then
  show      dterm-sem I (f1 fid2 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
    ≤ dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))

    using t
    dterm-sterm-dfree[OF free2, of I ?φs t snd (?φ t)]
    dterm-sterm-dfree[OF free1, of I ?φs t snd (?φ t)]
    by (simp add: f1-def)
qed


lemma DIGr-valid:valid DIGraxiom
  unfolding DIGraxiom-def
  apply(unfold DIGraxiom-def valid-def impl-sem iff-sem)
  apply(auto)
proof −
  fix I::('sf,'sc,'sz) interp and  b aa ba
    and sol::real ⇒ 'sz simple-state
    and t::real
  let ?ODE = OVar vid1
  let ?φ = (λt. mk-v I (?ODE) (sol 0, b) (sol t))
  assume t:0 ≤ t
    and sol:(sol solves-ode (λa. ODEs I vid1)) {0..t}
    {x. Predicates I vid1 (χ i. dterm-sem I (empty i) (mk-v I ?ODE (sol 0, b) x))}
    and notin: ¬(Predicates I vid1 (χ i. dterm-sem I (empty i) (sol 0, b)))
  have fsafe:fsafe (Prop vid1 empty) by (auto simp add: empty-def)
  from sol have Predicates I vid1 (χ i. dterm-sem I (empty i) (?φ 0))
    using t solves-ode-domainD[of sol (λa. ODEs I vid1) {0..t}]  by auto
  then have incon:Predicates I vid1 (χ i. dterm-sem I (empty i) ((sol 0, b)))
    using coincidence-formula[OF fsafe Iagree-refl[of I], of (sol 0, b) ?φ 0]
    unfolding Vagree-def by (auto simp add: empty-def)
  show dterm-sem I (f1 fid2 vid1)  (mk-v I (OVar vid1) (sol 0, b) (sol t)) <
dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
    using notin incon by auto
next
  fix I::('sf,'sc,'sz) interp and  b aa ba
    and sol::real ⇒ 'sz simple-state
    and t::real
  let ?ODE = OVar vid1
  let ?φ = (λt. mk-v I (?ODE) (sol 0, b) (sol t))
  assume t:0 ≤ t
    and sol:(sol solves-ode (λa. ODEs I vid1)) {0..t}
    {x. Predicates I vid1 (χ i. dterm-sem I (empty i) (mk-v I ?ODE (sol 0, b) x))}
    and notin: ¬(Predicates I vid1 (χ i. dterm-sem I (empty i) (sol 0, b)))
  have fsafe:fsafe (Prop vid1 empty) by (auto simp add: empty-def)
```

```
  from sol have Predicates I vid1 (χ i. dterm-sem I (empty i) (?φ 0))
    using t solves-ode-domainD[of sol (λa. ODEs I vid1) {0..t}] by auto
  then have incon:Predicates I vid1 (χ i. dterm-sem I (empty i) ((sol 0, b)))
    using coincidence-formula[OF fsafe Iagree-refl[of I], of (sol 0, b) ?φ 0]
    unfolding Vagree-def by (auto simp add: empty-def)
  show dterm-sem I (f1 fid2 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t)) <
dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
    using notin incon by auto
next
  fix I::('sf,'sc,'sz) interp and b aa ba
    and sol::real ⇒ 'sz simple-state
    and t::real
  let ?ODE = OVar vid1
  let ?φ = (λt. mk-v I (?ODE) (sol 0, b) (sol t))
  assume t:0 ≤ t
  assume sol:(sol solves-ode (λa. ODEs I vid1)) {0..t}
      {x. Predicates I vid1 (χ i. dterm-sem I (local.empty i) (mk-v I (OVar vid1)
(sol 0, b) x))}
  assume notin:¬ Predicates I vid1 (χ i. dterm-sem I (local.empty i) (sol 0, b))
  have fsafe:fsafe (Prop vid1 empty) by (auto simp add: empty-def)
  from sol have Predicates I vid1 (χ i. dterm-sem I (empty i) (?φ 0))
    using t solves-ode-domainD[of sol (λa. ODEs I vid1) {0..t}] by auto
  then have incon:Predicates I vid1 (χ i. dterm-sem I (empty i) ((sol 0, b)))
    using coincidence-formula[OF fsafe Iagree-refl[of I], of (sol 0, b) ?φ 0]
    unfolding Vagree-def by (auto simp add: empty-def)
  show dterm-sem I (f1 fid2 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
    < dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
    using incon notin by auto
next
  fix I::('sf,'sc,'sz) interp and b aa ba
  and sol::real ⇒ 'sz simple-state
  and t::real
  let ?ODE = OVar vid1
  let ?φ = (λt. mk-v I (?ODE) (sol 0, b) (sol t))
  assume good-interp:is-interp I
  assume aaba:(aa, ba) = mk-v I (OVar vid1) (sol 0, b) (sol t)
  assume t:0 ≤ t
  assume sol:(sol solves-ode (λa. ODEs I vid1)) {0..t}
      {x. Predicates I vid1 (χ i. dterm-sem I (local.empty i) (mk-v I (OVar vid1)
(sol 0, b) x))}
  assume box:∀ a ba. (∃ sola. sol 0 = sola 0 ∧
                  (∃ t. (a, ba) = mk-v I (OVar vid1) (sola 0, b) (sola t) ∧
                      0 ≤ t ∧
                      (sola solves-ode (λa. ODEs I vid1)) {0..t}
                       {x. Predicates I vid1
                            (χ i. dterm-sem I (local.empty i) (mk-v I (OVar vid1)
(sola 0, b) x))})) ⟶
            directional-derivative I (f1 fid2 vid1) (a, ba) ≤ directional-derivative I
(f1 fid1 vid1) (a, ba)
```

assume *geq0*:*dterm-sem I* (*f1 fid2 vid1*) (*sol 0*, *b*) < *dterm-sem I* (*f1 fid1 vid1*) (*sol 0*, *b*)
have *free1*:*dfree* ($*f fid2* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*))
by (*auto intro*: *dfree.intros*)
have *free2*:*dfree* ($*f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*))
by (*auto intro*: *dfree.intros*)
from *geq0*
have *geq0'*:*sterm-sem I* (*f1 fid2 vid1*) (*sol 0*) < *sterm-sem I* (*f1 fid1 vid1*) (*sol 0*)
unfolding *f1-def* **using** *dterm-sterm-dfree*[*OF free1*, *of I sol 0 b*] *dterm-sterm-dfree*[*OF free2*, *of I sol 0 b*]
by *auto*
let *?φs* = λ*x*. *fst* (*?φ x*)
let *?φt* = λ*x*. *snd* (*?φ x*)
let *?df1* = (λ*t*. *dterm-sem I* (*f1 fid2 vid1*) (*?φ t*))
let *?f1* = (λ*t*. *sterm-sem I* (*f1 fid2 vid1*) (*?φs t*))
let *?f1'* = (λ *s t'*. *t'* ∗ *frechet I* (*f1 fid2 vid1*) (*?φs s*) (*?φt s*))
have *dfeq*:*?df1* = *?f1*
apply(*rule ext*)
subgoal for *t*
using *dterm-sterm-dfree*[*OF free1*, *of I ?φs t snd* (*?φ t*)] **unfolding** *f1-def expand-singleton* **by** *auto*
done
have *free3*:*dfree* (*f1 fid2 vid1*) **unfolding** *f1-def* **by** (*auto intro*: *dfree.intros*)
let *?df2* = (λ*t*. *dterm-sem I* (*f1 fid1 vid1*) (*?φ t*))
let *?f2* = (λ*t*. *sterm-sem I* (*f1 fid1 vid1*) (*?φs t*))
let *?f2'* = (λ*s t'* . *t'* ∗ *frechet I* (*f1 fid1 vid1*) (*?φs s*) (*?φt s*))
let *?int* = {*0..t*}
have *bluh*:⋀*x i*. (*Functions I i has-derivative* (*THE f'*. ∀ *x*. (*Functions I i has-derivative f'* *x*) (*at x*)) *x*) (*at x*)
using *good-interp* **unfolding** *is-interp-def* **by** *auto*
have *blah*:(*Functions I fid2 has-derivative* (*THE f'*. ∀ *x*. (*Functions I fid2 has-derivative f'* *x*) (*at x*)) (χ *i*. *if i* = *vid1 then sol t* $ *vid1 else 0*)) (*at* (χ *i*. *if i* = *vid1 then sol t* $ *vid1 else 0*))
using *bluh* **by** *auto*
have *bigEx*:⋀*s*. *s* ∈ {*0..t*} ⟹(∃ *sola*. *sol 0* = *sola 0* ∧
(∃ *ta*. (*fst* (*?φ s*),
snd (*?φ s*)) =
mk-v I (*OVar vid1*) (*sola 0*, *b*) (*sola ta*) ∧
0 ≤ *ta* ∧
(*sola solves-ode* (λ*a*. *ODEs I vid1*)) {*0..ta*}
{*x*. *Predicates I vid1* (χ *i*. *dterm-sem I* (*local.empty i*) (*mk-v I* (*OVar vid1*) (*sol 0*, *b*) *x*))}))
subgoal for *s*
apply(*rule exI*[**where** *x*=*sol*])
apply(*rule conjI*)
subgoal by (*rule refl*)
apply(*rule exI*[**where** *x*=*s*])
apply(*rule conjI*)

subgoal by *auto*
apply(*rule conjI*)
subgoal by *auto*
using *sol*
using *atLeastAtMost-iff atLeastatMost-subset-iff order-refl solves-ode-on-subset*
by (*metis* (*no-types*, *lifting*) *subsetI*)
done
have *box'*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ *directional-derivative I* (*f1 fid2 vid1*) (*?φs s*, *?φt s*)
≤ *directional-derivative I* (*f1 fid1 vid1*) (*?φs s*, *?φt s*)
subgoal for *s*
using *box*
apply *simp*
apply (*erule allE*[**where** *x*=*?φs s*])
apply (*erule allE*[**where** *x*=*?φt s*])
using *bigEx*[*of s*] **by** *auto*
done
have *dsafe1*:*dsafe* (*f1 fid2 vid1*) **unfolding** *f1-def* **by** (*auto intro*: *dsafe.intros*)
have *dsafe2*:*dsafe* (*f1 fid1 vid1*) **unfolding** *f1-def* **by** (*auto intro*: *dsafe.intros*)
have *agree1*:*Vagree* (*sol 0*, *b*) (*?φ 0*) (*FVT* (*f1 fid2 vid1*))
using *mk-v-agree*[*of I* (*OVar vid1*) (*sol 0*, *b*) *fst* (*?φ 0*)]
unfolding *f1-def Vagree-def expand-singleton*
apply *auto*
by (*metis* (*no-types*, *lifting*) *Compl-iff Vagree-def fst-conv mk-v-agree mk-xode.simps semBV.simps*)
have *agree2*:*Vagree* (*sol 0*, *b*) (*?φ 0*) (*FVT* (*f1 fid1 vid1*))
using *mk-v-agree*[*of I* (*OVar vid1*) (*sol 0*, *b*) *fst* (*?φ 0*)]
unfolding *f1-def Vagree-def expand-singleton*
apply *auto*
by (*metis* (*no-types*, *lifting*) *Compl-iff Vagree-def fst-conv mk-v-agree mk-xode.simps semBV.simps*)
have *sem-eq1*:*dterm-sem I* (*f1 fid2 vid1*) (*sol 0*, *b*) = *dterm-sem I* (*f1 fid2 vid1*) (*?φ 0*)
using *coincidence-dterm*[*OF dsafe1 agree1*] **by** *auto*
then have *sem-eq1'*:*sterm-sem I* (*f1 fid2 vid1*) (*sol 0*) = *sterm-sem I* (*f1 fid2 vid1*) (*?φs 0*)
using *dterm-sterm-dfree*[*OF free1*, *of I sol 0 b*]
dterm-sterm-dfree[*OF free1*, *of I* (*?φs 0*) *snd* (*?φ 0*)]
unfolding *f1-def expand-singleton* **by** *auto*
have *sem-eq2*:*dterm-sem I* (*f1 fid1 vid1*) (*sol 0*, *b*) = *dterm-sem I* (*f1 fid1 vid1*) (*?φ 0*)
using *coincidence-dterm*[*OF dsafe2 agree2*] **by** *auto*
then have *sem-eq2'*:*sterm-sem I* (*f1 fid1 vid1*) (*sol 0*) = *sterm-sem I* (*f1 fid1 vid1*) (*?φs 0*)
using *dterm-sterm-dfree*[*OF free2*, *of I sol 0 b*] *dterm-sterm-dfree*[*OF free2*, *of I* (*?φs 0*) *snd* (*?φ 0*)]
unfolding *f1-def expand-singleton* **by** *auto*
have *good-interp'*:⋀*i x*. (*Functions I i has-derivative* (*THE f'*. ∀ *x*. (*Functions I i has-derivative f' x*) (*at x*)) *x*) (*at x*)

using *good-interp* **unfolding** *is-interp-def* **by** *auto*
have *chain* :
$\bigwedge f\ f'\ g\ g'\ x\ s.$
$(f$ *has-derivative* $f')$ $(at\ x\ within\ s) \Longrightarrow$
$(g$ *has-derivative* $g')$ $(at\ (f\ x)\ within\ f\ `\ s) \Longrightarrow (g \circ f$ *has-derivative* $g' \circ f')$
$(at\ x\ within\ s)$
by(*auto intro*: *derivative-intros*)
have *sol1*:$(sol$ *solves-ode* $(\lambda$-. *ODE-sem I* $(OVar\ vid1)))$ $\{0..t\}$ $\{x.$ *mk-v I* $(OVar$ *vid1*$)$ $(sol\ 0,\ b)\ x \in fml\text{-}sem\ I\ (Prop\ vid1\ empty)\}$
using *sol* **unfolding** *p1-def singleton-def empty-def* **by** *auto*
have *FVTsub1*:*vid1* $\in$ *ODE-vars I* $(OVar\ vid1) \Longrightarrow FVT\ (\$f\ fid2\ (\lambda i.\ if\ i =$
vid1 then trm.Var vid1 else Const 0$)) \subseteq semBV\ I\ ((OVar\ vid1))$
apply *auto*
subgoal for *x xa*
apply(*cases xa* = *vid1*)
by *auto*
done
have *FVTsub2*:*vid1* $\in$ *ODE-vars I* $(OVar\ vid1) \Longrightarrow FVT\ (\$f\ fid1\ (\lambda i.\ if\ i =$
vid1 then trm.Var vid1 else Const 0$)) \subseteq semBV\ I\ ((OVar\ vid1))$
apply *auto*
subgoal for *x xa*
apply(*cases xa* = *vid1*)
by *auto*
done
have *osafe*:*osafe* $(OVar\ vid1)$
by *auto*
have *deriv1*:$\bigwedge s.$ *vid1* $\in$ *ODE-vars I* $(OVar\ vid1) \Longrightarrow s \in ?int \Longrightarrow (?f1$ *has-derivative*
$(?f1'\ s))$ $(at\ s\ within\ \{0..t\})$
subgoal for *s*
using *rift-in-space-time*[*OF good-interp free1 osafe sol1 FVTsub1*, *of s*]
unfolding *f1-def expand-singleton directional-derivative-def*
by *blast*
done
have *deriv2*:$\bigwedge s.$ *vid1* $\in$ *ODE-vars I* $(OVar\ vid1) \Longrightarrow s \in ?int \Longrightarrow (?f2$ *has-derivative*
$(?f2'\ s))$ $(at\ s\ within\ \{0..t\})$
subgoal for *s*
using *rift-in-space-time*[*OF good-interp free2 osafe sol1 FVTsub2*, *of s*]
unfolding *f1-def expand-singleton directional-derivative-def*
by *blast*
done
have *leq*:$\bigwedge s\ .\ s \in ?int \Longrightarrow ?f1'\ s\ 1 \leq ?f2'\ s\ 1$
subgoal for *s* **using** *box'*[*of s*]
by (*simp add*: *directional-derivative-def*)
done
have *preserve-agree1*:*vid1* $\notin$ *ODE-vars I* $(OVar\ vid1) \Longrightarrow VSagree\ (sol\ 0)\ (fst$
$(mk\text{-}v\ I\ (OVar\ vid1)\ (sol\ 0,\ b)\ (sol\ t)))\ \{vid1\}$
using *mk-v-agree*[*of I OVar vid1* $(sol\ 0,\ b)$ *sol t*]
unfolding *Vagree-def VSagree-def*
by *auto*

```
  have preserve-coincide1:
    vid1 ∉ ODE-vars I (OVar vid1) ⟹
    sterm-sem I (f1 fid2 vid1) (fst (mk-v I (OVar vid1) (sol 0, b) (sol t)))
   = sterm-sem I (f1 fid2 vid1) (sol 0)
    using coincidence-sterm[of (sol 0, b) (mk-v I (OVar vid1) (sol 0, b) (sol t)) f1
fid2 vid1 I]
    preserve-agree1 unfolding VSagree-def Vagree-def f1-def by auto
  have preserve-coincide2:
    vid1 ∉ ODE-vars I (OVar vid1) ⟹
    sterm-sem I (f1 fid1 vid1) (fst (mk-v I (OVar vid1) (sol 0, b) (sol t)))
   = sterm-sem I (f1 fid1 vid1) (sol 0)
    using coincidence-sterm[of (sol 0, b) (mk-v I (OVar vid1) (sol 0, b) (sol t)) f1
fid1 vid1 I]
    preserve-agree1 unfolding VSagree-def Vagree-def f1-def by auto
  have ?f1 t < ?f2 t
    apply(cases t = 0)
     subgoal using geq0' sem-eq1' sem-eq2' by auto
    subgoal
      apply(cases vid1 ∈ ODE-vars I (OVar vid1))
      subgoal
        apply (rule MVT'-gr[OF deriv2 deriv1, of t])
              subgoal by auto
             subgoal by auto
            subgoal for s using deriv2[of s] using leq by auto
           using t leq geq0' sem-eq1' sem-eq2' by auto
      subgoal
        using geq0
        using dterm-sterm-dfree[OF free1, of I sol 0 b]
        using dterm-sterm-dfree[OF free2, of I sol 0 b]
        using preserve-coincide1 preserve-coincide2
        by(simp add: f1-def)
      done
    done
  then
  show dterm-sem I (f1 fid2 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
     < dterm-sem I (f1 fid1 vid1) (mk-v I (OVar vid1) (sol 0, b) (sol t))
    using t
    dterm-sterm-dfree[OF free2, of I ?φs t snd (?φ t)]
    dterm-sterm-dfree[OF free1, of I ?φs t snd (?φ t)]
    using geq0 f1-def
    by (simp add: f1-def)
qed

lemma DG-valid:valid DGaxiom
proof −
  have osafe:osafe (OSing vid1 (f1 fid1 vid1))
    by(auto simp add: osafe-Sing dfree-Fun dfree-Const f1-def expand-singleton)
  have fsafe:fsafe (p1 vid1 vid1)
    by(auto simp add: p1-def dfree-Const)
```

have *osafe2*:*osafe* (*OProd* (*OSing vid1* (*f1 fid1 vid1*)) (*OSing vid2* (*Plus* (*Times* (*f1 fid2 vid1*) (*trm.Var vid2*)) (*f1 fid3 vid1*))))
by(*auto simp add*: *f1-def expand-singleton osafe.intros dfree.intros vne12*)
note *sem* = *ode-alt-sem*[*OF osafe fsafe*]
note *sem2* = *ode-alt-sem*[*OF osafe2 fsafe*]
have *p2safe*:*fsafe* (*p1 vid2 vid1*) **by**(*auto simp add*: *p1-def dfree-Const*)
show *valid DGaxiom*
apply(*auto simp del*: *prog-sem.simps*(*8*) *simp add*: *DGaxiom-def valid-def sem sem2*)
apply(*rule exI*[**where** *x*=*0*], *auto simp add*: *f1-def p1-def expand-singleton*)
subgoal for *I a b aa ba sol t*
proof −
assume *good-interp*:*is-interp I*
assume
$\forall$ *aa ba*. ($\exists$ *sol t*. (*aa*, *ba*) = *mk-v I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol t*) $\wedge$
0 $\leq$ *t* $\wedge$
(*sol solves-ode* (λ*a b*. χ *i*. *if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) *b else 0*)) {*0*..*t*}
{*x*. *Predicates I vid1*
(χ *i*. *dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) *x*))} $\wedge$
VSagree (*sol 0*) *a* {*uu*. *uu* = *vid1* $\vee$
Inl uu $\in$ *Inl* ‘ {*x*. $\exists$ *xa*. *Inl x* $\in$ *FVT* (*if xa* = *vid1 then trm.Var vid1 else Const 0*)} $\vee$
($\exists$ *x*. *Inl uu* $\in$ *FVT* (*if x* = *vid1 then trm.Var vid1 else Const 0*))}) $\longrightarrow$
Predicates I vid2 (χ *i*. *dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*) (*aa*, *ba*))
then have
bigAll:
$\bigwedge$*aa ba*. ($\exists$ *sol t*. (*aa*, *ba*) = *mk-v I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol t*) $\wedge$
0 $\leq$ *t* $\wedge$
(*sol solves-ode* (λ*a b*. χ *i*. *if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) *b else 0*)) {*0*..*t*}
{*x*. *Predicates I vid1*
(χ *i*. *dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) *x*))} $\wedge$
VSagree (*sol 0*) *a* {*uu*. *uu* = *vid1* $\vee$ ($\exists$ *x*. *Inl uu* $\in$ *FVT* (*if x* = *vid1 then trm.Var vid1 else Const 0*))}) $\longrightarrow$
Predicates I vid2 (χ *i*. *dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*) (*aa*, *ba*))
by (*auto*)
assume *aaba*:(*aa*, *ba*) =

mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))
(OSing vid2
(Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0)) (trm.Var vid2))
($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
(χ y. if vid2 = y then 0 else fst (a, b) $ y, b) (sol t)
assume *t:0 ≤ t*
assume *sol:*
(sol solves-ode
(λa b. (χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) b else 0) +
(χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var vid2)) (f1 fid3 vid1)) b else 0)))
{0..t} {x. Predicates I vid1
(χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
(mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))
(OSing vid2
(Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0)) (trm.Var vid2))
($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
(χ y. if vid2 = y then 0 else fst (a, b) $ y, b) x))}
assume *VSag:VSagree (sol 0) (χ y. if vid2 = y then 0 else fst (a, b) $ y)*
{x. x = vid2 ∨ x = vid1 ∨ x = vid2 ∨ x = vid1 ∨ Inl x ∈ Inl ‘ {x. x = vid2 ∨ x = vid1} ∨ x = vid1}
let *?sol = (λt. χ i. if i = vid1 then sol t $ vid1 else 0)*
let *?aaba′ = mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))) (a, b) (?sol t)*
from *bigAll[of fst ?aaba′ snd ?aaba′]*
have *bigEx:(∃ sol t. ?aaba′ = mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))) (a, b) (sol t) ∧*
0 ≤ t ∧
(sol solves-ode (λa b. χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) b else 0)) {0..t}
{x. Predicates I vid1
(χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
(mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))) (a, b) x))} ∧
VSagree (sol 0) a {uu. uu = vid1 ∨ (∃ x. Inl uu ∈ FVT (if x = vid1 then trm.Var vid1 else Const 0))}) ⟶
Predicates I vid2 (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0) (?aaba′))
by *simp*
have *pre1:?aaba′ = mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))) (a, b) (?sol t)*
by *(rule refl)*
have *agreeL:⋀s. fst (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then*

trm.Var vid1 else Const 0)))
(*OSing vid2*
(*Plus* (*Times* ($*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
($*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2 = y then 0 else fst* (*a*, *b*) $ *y*, *b*) (*sol s*)) $ *vid1 = sol s* $ *vid1*
subgoal for *s*
using *mk-v-agree*[*of I* (*OProd* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* ($*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
($*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))) (χ *y*. *if vid2 = y then 0 else fst* (*a*, *b*) $ *y*, *b*) (*sol s*)]
unfolding *Vagree-def* **by** *auto* **done**
have *agreeR*:⋀*s*. *fst* (*mk-v I* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (χ *i*. *if i = vid1 then sol s* $ *vid1 else 0*)) $ *vid1 = sol s* $ *vid1*
subgoal for *s*
using *mk-v-agree*[*of I* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (χ *i*. *if i = vid1 then sol s* $ *vid1 else 0*)]
unfolding *Vagree-def* **by** *auto*
done
have *FV*:(*FVF* (*p1 vid1 vid1*)) = {*Inl vid1*} **unfolding** *p1-def expand-singleton*
apply *auto* **subgoal for** *x xa* **apply**(*cases xa = vid1*) **by** *auto* **done**
have *agree*:⋀*s*. *Vagree* (*mk-v I* (*OProd* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* ($*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
($*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2 = y then 0 else fst* (*a*, *b*) $ *y*, *b*) (*sol s*)) (*mk-v I* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (χ *i*. *if i = vid1 then sol s* $ *vid1 else 0*)) (*FVF* (*p1 vid1 vid1*))
using *agreeR agreeL* **unfolding** *Vagree-def FV* **by** *auto*
note *con-sem-eq* = *coincidence-formula*[*OF fsafe Iagree-refl agree*]
have *constraint*:⋀*s*. *0* ≤ *s* ∧ *s* ≤ *t* ⟹
Predicates I vid1
(χ *i*. *dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (χ *i*. *if i = vid1 then sol s* $ *vid1 else 0*)))
using *sol* **apply** *simp*
apply(*drule solves-odeD*(*2*))
apply *auto*[*1*]
subgoal for *s* **using** *con-sem-eq* **by** (*auto simp add*: *p1-def expand-singleton*)
done
have *eta*:*sol* = (λ*t*. χ *i*. *sol t* $ *i*) **by** (*rule ext*, *rule vec-extensionality*, *simp*)
have *yet-another-eq*:⋀*x*. (λ*xa*. *xa* ∗$_R$ ((χ *i*. *if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (*sol x*) *else 0*) +

(χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var vid2)) (f1 fid3 vid1)) (sol x) else 0)))
= (λxa. (χ i. (xa *R ((χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) (sol x) else 0) +
(χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var vid2)) (f1 fid3 vid1)) (sol x) else 0))) $ i))
subgoal for *x* **by** (*rule ext*, *rule vec-extensionality*, *simp*) **done**
have *sol-deriv*:⋀*x*. *x* ∈{*0..t*} ⟹
(*sol has-derivative*
(λxa. xa *R ((χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) (sol x) else 0) +
(χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var vid2)) (f1 fid3 vid1)) (sol x) else 0))))
(*at x within* {*0..t*})
using *sol* **apply** *simp*
apply(*drule solves-odeD*(*1*))
unfolding *has-vderiv-on-def has-vector-derivative-def* **by** *auto*
then have *sol-deriv*:⋀*x*. *x* ∈ {*0..t*} ⟹
((λt. χ i. sol t $ i) *has-derivative*
(λxa. (χ i. (xa *R ((χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) (sol x) else 0) +
(χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var vid2)) (f1 fid3 vid1)) (sol x) else 0))) $ i)))
(*at x within* {*0..t*}) **using** *yet-another-eq eta* **by** *auto*
have *sol-deriv1*: ⋀*x*. *x* ∈ {*0..t*} ⟹
((λt. sol t $ *vid1*) *has-derivative*
(λxa. (xa *R ((χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) (sol x) else 0) +
(χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var vid2)) (f1 fid3 vid1)) (sol x) else 0)) $ *vid1*)))
(*at x within* {*0..t*})
subgoal for *s*

apply(*rule has-derivative-proj*[*of* (λ i t. sol t $ i) (λj xa. (xa *R ((χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) (sol s) else 0) +
(χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var vid2)) (f1 fid3 vid1)) (sol s) else 0)) $ j)) *at s within* {*0..t*}*vid1*])
using *sol-deriv*[*of s*] **by** *auto* **done**
have *hmm*:⋀*s*. (χ i. sterm-sem I (if i = vid1 then trm.Var vid1 else Const 0) (sol s)) = (χ i. sterm-sem I (if i = vid1 then trm.Var vid1 else Const 0) (χ i. if i = vid1 then sol s $ vid1 else 0))
by(*rule vec-extensionality*, *auto*)
have *aha*:⋀*s*. (λxa. xa ∗ sterm-sem I (f1 fid1 vid1) (sol s)) = (λxa. xa ∗ sterm-sem I (f1 fid1 vid1) (χ i. if i = vid1 then sol s $ vid1 else 0))
subgoal for *s*
apply(*rule ext*)
subgoal for *xa* **using** *hmm* **by** (*auto simp add*: *f1-def*) **done done**
let *?sol′* = (λs. (λxa. χ i. if i = vid1 then xa ∗ sterm-sem I (f1 fid1 vid1) (χ i. if i = vid1 then sol s $ vid1 else 0) else 0))

let *?project-me-plz* = (λ*t*. (χ *i*. *if i* = *vid1 then ?sol t* $ *vid1 else 0*))
have *sol-deriv-eq*:⋀*s*. *s* ∈{*0..t*} ⟹
((λ*t*. (χ *i*. *if i* = *vid1 then ?sol t* $ *vid1 else 0*)) *has-derivative ?sol*′ *s*) (*at s within* {*0..t*})
subgoal for *s*
apply(*rule has-derivative-vec*)
subgoal for *i*
apply (*cases i* = *vid1*, *cases i* = *vid2*, *auto*)
using *vne12* **apply** *simp*
using *sol-deriv1*[*of s*] **using** *aha* **by** *auto*
done done
have *yup*:(λ*t*. (χ *i*. *if i* = *vid1 then ?sol t* $ *vid1 else 0*) $ *vid1*) = (λ*t*. *sol t* $ *vid1*)
by(*rule ext*, *auto*)
have *maybe*:⋀*s*. (λ*xa*. *xa* ∗ *sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1 then sol s* $ *vid1 else 0*)) = (λ*xa*. (χ *i*. *if i* = *vid1 then xa* ∗ *sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1 then sol s* $ *vid1 else 0*) *else 0*) $ *vid1*)
by(*rule ext*, *auto*)
have *almost*:(λ*x*. *if vid1* = *vid1 then* (χ *i*. *if i* = *vid1 then sol x* $ *vid1 else 0*) $ *vid1 else 0*) =
(λ*x*. (χ *i*. *if i* = *vid1 then sol x* $ *vid1 else 0*) $ *vid1*) **by**(*rule ext*, *auto*)
have *almost*′:⋀*s*. (λ*h*. *if vid1* = *vid1 then h* ∗ *sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1 then sol s* $ *vid1 else 0*) *else 0*) = (λ*h*. *h* ∗ *sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1 then sol s* $ *vid1 else 0*))
by(*rule ext*, *auto*)
have *deriv*′: ⋀*x*. *x* ∈ {*0..t*} ⟹
((λ*t*. χ *i*. *if i* = *vid1 then sol t* $ *vid1 else 0*) *has-derivative*
(λ*xa*. (χ *i*. *xa* ∗*R* (*if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1 then sol x* $ *vid1 else 0*) *else 0*))))
(*at x within* {*0..t*})
subgoal for *s*
apply(*rule has-derivative-vec*)
subgoal for *i*
apply(*cases i* = *vid1*)
prefer *2* **subgoal by** *auto*
apply *auto*
using *has-derivative-proj*[*OF sol-deriv-eq*[*of s*], *of vid1*] **using** *yup maybe*[*of s*] *almost almost*′[*of s*]
by *fastforce*
done
done
have *derEq*:⋀*s*. (λ*xa*. (χ *i*. *xa* ∗*R* (*if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1 then sol s* $ *vid1 else 0*) *else 0*)))
= (λ*xa*. *xa* ∗*R* (χ *i*. *if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1 then sol s* $ *vid1 else 0*) *else 0*))
subgoal for *s* **apply** (*rule ext*, *rule vec-extensionality*) **by** *auto* **done**
have ⋀*x*. *x* ∈ {*0..t*} ⟹
((λ*t*. χ *i*. *if i* = *vid1 then sol t* $ *vid1 else 0*) *has-derivative*
(λ*xa*. *xa* ∗*R* (χ *i*. *if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i* = *vid1*

then sol x $ vid1 else 0) else 0)))
(at x within {0..t}) **subgoal for** *s* **using** *deriv'[of s] derEq[of s]* **by** *auto* **done**
then have *deriv:((λt. χ i. if i = vid1 then sol t $ vid1 else 0) has-vderiv-on*
(λt. χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) (χ i. if i = vid1 then sol t $ vid1 else 0) else 0))
{0..t}
unfolding *has-vderiv-on-def has-vector-derivative-def*
by *auto*
have *pre2:(?sol solves-ode (λa b. χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) b else 0)) {0..t}*
{x. Predicates I vid1
(χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
(mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))) (a, b) x)}
apply(*rule solves-odeI*)
subgoal by (*rule deriv*)
subgoal for *s* **using** *constraint* **by** *auto*
done
have *pre3:VSagree (?sol 0) a {u. u = vid1 ∨ (∃ x. Inl u ∈ FVT (if x = vid1 then trm.Var vid1 else Const 0))}*
using *vne12 VSag* **unfolding** *VSagree-def* **by** *simp*
have *bigPre:(∃ sol t. ?aaba' = mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then Var vid1 else Const 0))) (a, b) (sol t) ∧*
0 ≤ t ∧
(sol solves-ode (λa b. χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) b else 0)) {0..t}
{x. Predicates I vid1
(χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
(mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then Var vid1 else Const 0))) (a, b) x)} ∧
VSagree (sol 0) a {u. u = vid1 ∨ (∃ x. Inl u ∈ FVT (if x = vid1 then Var vid1 else Const 0))})
apply(*rule exI*[**where** *x=?sol*])
apply(*rule exI*[**where** *x=t*])
apply(*rule conjI*)
apply(*rule pre1*)
apply(*rule conjI*)
apply(*rule t*)
apply(*rule conjI*)
apply(*rule pre2*)
by(*rule pre3*)
have *pred2:Predicates I vid2 (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0) ?aaba')*
using *bigEx bigPre* **by** *auto*
then have *pred2':?aaba' ∈ fml-sem I (p1 vid2 vid1)* **unfolding** *p1-def expand-singleton* **by** *auto*
let *?res-state = (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))*

(*OSing vid2*
(*Plus* (*Times* (*$f fid2* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(*$f fid3* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))))
(*χ y. if vid2 = y then 0 else fst* (*a*, *b*) *$ y*, *b*) (*sol t*))
have *aabaX*:(*fst ?aaba'*) *$ vid1 = sol t $ vid1*
using *aaba mk-v-agree*[*of I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))
(*a*, *b*) (*?sol t*)]
proof −
assume *Vagree* (*mk-v I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*χ i. if i = vid1 then sol t $ vid1 else 0*))
(*a*, *b*) (− *semBV I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))) ∧
Vagree (*mk-v I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*χ i. if i = vid1 then sol t $ vid1 else 0*))
(*mk-xode I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*χ i. if i = vid1 then sol t $ vid1 else 0*))
(*semBV I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))))
then have *ag*: *Vagree* (*mk-v I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*?sol t*))
(*mk-xode I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*?sol t*))
(*semBV I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))))
by *auto*
have *sembv*:(*semBV I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))) = {*Inl vid1*, *Inr vid1*}
by *auto*
have *sub*:{*Inl vid1*} ⊆ {*Inl vid1*, *Inr vid1*} **by** *auto*
have *ag'*:*Vagree* (*mk-v I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*?sol t*))
(*mk-xode I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*?sol t*)) {*Inl vid1*}
using *ag agree-sub*[*OF sub*] *sembv* **by** *auto*
then have *eq1*:*fst* (*mk-v I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*?sol t*)) *$ vid1*
= *fst* (*mk-xode I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*?sol t*)) *$ vid1* **unfolding** *Vagree-def* **by** *auto*
moreover have *... = sol t $ vid1* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
have *res-stateX*:(*fst ?res-state*) *$ vid1 = sol t $ vid1*
using *mk-v-agree*[*of I* (*OProd* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* (*$f fid2* (*λi. if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))

```
                    ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
          (χ y. if vid2 = y then 0 else fst (a, b) $ y, b) (sol t)]
    proof −
      assume Vagree (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then
trm.Var vid1 else Const 0)))
                  (OSing vid2
                    (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else
Const 0)) (trm.Var vid2))
                    ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
          (χ y. if vid2 = y then 0 else fst (a, b) $ y, b) (sol t))
    (χ y. if vid2 = y then 0 else fst (a, b) $ y, b)
    (− semBV I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1
else Const 0)))
                (OSing vid2
                 (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const
0)) (trm.Var vid2))
                  ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))) ∧
   Vagree (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1
else Const 0)))
                  (OSing vid2
                    (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else
Const 0)) (trm.Var vid2))
                    ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
          (χ y. if vid2 = y then 0 else fst (a, b) $ y, b) (sol t))
    (mk-xode I
      (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const
0)))
       (OSing vid2
         (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0))
(trm.Var vid2))
          ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
      (sol t))
    (semBV I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0)))
               (OSing vid2
                 (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const
0)) (trm.Var vid2))
                  ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0))))))
      then have ag: Vagree (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1
then trm.Var vid1 else Const 0)))
                  (OSing vid2
                    (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const
0)) (trm.Var vid2))
                    ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
        (χ y. if vid2 = y then 0 else fst (a, b) $ y, b) (sol t))
 (mk-xode I
   (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))
    (OSing vid2
      (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0))
```

```
(trm.Var vid2))
        ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
   (sol t))
 (semBV I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0)))
           (OSing vid2
            (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0))
(trm.Var vid2))
              ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))) by auto
       have sembv:(semBV I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then
trm.Var vid1 else Const 0)))
           (OSing vid2
            (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0))
(trm.Var vid2))
              ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))) = {Inl
vid1, Inr vid1, Inl vid2, Inr vid2} by auto
       have sub:{Inl vid1} ⊆ {Inl vid1, Inr vid1, Inl vid2, Inr vid2} by auto
       have ag':Vagree (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then
trm.Var vid1 else Const 0)))
              (OSing vid2
               (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const
0)) (trm.Var vid2))
                 ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
        (χ y. if vid2 = y then 0 else fst (a, b) $ y, b) (sol t))
   (mk-xode I
    (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))
      (OSing vid2
        (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0))
(trm.Var vid2))
          ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
    (sol t)) {Inl vid1} using ag sembv agree-sub[OF sub] by auto
      then have fst ?res-state $ vid1 = fst ((mk-xode I
    (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0)))
      (OSing vid2
        (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const 0))
(trm.Var vid2))
          ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
    (sol t))) $ vid1 unfolding Vagree-def by blast
      moreover have ... = sol t $ vid1 by auto
      ultimately show ?thesis by linarith
     qed
    have agree:Vagree ?aaba' (?res-state) (FVF (p1 vid2 vid1))
      unfolding p1-def Vagree-def using aabaX res-stateX by auto
    have fml-sem-eq:(?res-state ∈ fml-sem I (p1 vid2 vid1)) = (?aaba' ∈ fml-sem
I (p1 vid2 vid1))
      using coincidence-formula[OF p2safe Iagree-refl agree, of I] by auto
    then show Predicates I vid2
    (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
          (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1
```

else Const 0)))
(*OSing vid2*
(*Plus* (*Times* (\$*f fid2* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(\$*f fid3* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2* = *y then 0 else fst* (*a*, *b*) \$ *y*, *b*) (*sol t*)))
using *pred2* **unfolding** *p1-def expand-singleton* **by** *auto*
qed
subgoal for *I a b r aa ba sol t*
proof −
assume *good-interp*:*is-interp I*
assume *bigAll*: ∀ *aa ba*. (∃ *sol t*. (*aa*, *ba*) =
mk-v I (*OProd* (*OSing vid1* (\$*f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* (\$*f fid2* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(\$*f fid3* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2* = *y then r else fst* (*a*, *b*) \$ *y*, *b*) (*sol t*) ∧
0 ≤ *t* ∧
(*sol solves-ode*
(λ*a b*. (χ *i*. *if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) *b else 0*) +
(χ *i*. *if i* = *vid2 then sterm-sem I* (*Plus* (*Times* (*f1 fid2 vid1*) (*trm.Var vid2*)) (*f1 fid3 vid1*)) *b else 0*)))
{0..*t*} {*x*. *Predicates I vid1*
(χ *i*. *dterm-sem I* (*if i* = *vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OProd* (*OSing vid1* (\$*f fid1* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* (\$*f fid2* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(\$*f fid3* (λ*i*. *if i* = *vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2* = *y then r else fst* (*a*, *b*) \$ *y*, *b*) *x*))} ∧
VSagree (*sol 0*) (χ *y*. *if vid2* = *y then r else fst* (*a*, *b*) \$ *y*)
{*uu*. *uu* = *vid2* ∨
uu = *vid1* ∨
uu = *vid2* ∨
uu = *vid1* ∨
Inl uu
∈ *Inl* ‘ ({*x*. ∃ *xa*. *Inl x* ∈ *FVT* (*if xa* = *vid1 then trm.Var vid1 else Const 0*)} ∪
{*x*. *x* = *vid2* ∨ (∃ *xa*. *Inl x* ∈ *FVT* (*if xa* = *vid1 then trm.Var vid1 else Const 0*))}) ∨
(∃ *x*. *Inl uu* ∈ *FVT* (*if x* = *vid1 then trm.Var vid1 else Const 0*))}) ⟶

Predicates I vid2 (χ *i. dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*) (*aa*, *ba*))
assume *aaba*:(*aa*, *ba*) = *mk-v I* (*OSing vid1* (*$f fid1* (λ*i. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol t*)
assume *t*:*0* ≤ *t*
assume *sol*:(*sol solves-ode* (λ*a b.* χ *i. if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) *b else 0*)) {*0..t*}
{*x. Predicates I vid1*
(χ *i. dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OSing vid1* (*$f fid1* (λ*i. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) *x*)}
assume *VSA*:*VSagree* (*sol 0*) *a*
{*uu. uu = vid1* ∨
Inl uu ∈ *Inl* ' {*x.* ∃ *xa. Inl x* ∈ *FVT* (*if xa = vid1 then trm.Var vid1 else Const 0*)} ∨
(∃ *x. Inl uu* ∈ *FVT* (*if x = vid1 then trm.Var vid1 else Const 0*))}
let *?xode* = (λ*a b.* χ *i. if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) *b else 0*)
let *?xconstraint* = *UNIV*
let *?ivl* = *ll-on-open.existence-ivl* {*0 .. t*} *?xode ?xconstraint 0* (*sol 0*)
have *freef1*:*dfree* (*$f fid1* (λ*i. if i = vid1 then trm.Var vid1 else Const 0*))
by(*auto simp add*: *dfree-Fun dfree-Const*)
have *simple-term-inverse'*:⋀ϑ. *dfree* ϑ ⟹ *raw-term* (*simple-term* ϑ) = ϑ
using *simple-term-inverse* **by** *auto*
have *old-lipschitz*:*local-lipschitz* (*UNIV*::*real set*) *UNIV* (λ*a b.* χ *i. if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) *b else 0*)
apply(*rule c1-implies-local-lipschitz*[**where** *f'*=(λ (*t*,*b*). *blinfun-vec*(λ *i. if i = vid1 then blin-frechet* (*good-interp I*) (*simple-term* (*Function fid1* (λ *i. if i = vid1 then Var vid1 else Const 0*))) *b else Blinfun*(λ -. *0*)))])
apply *auto*
subgoal for *x*
apply(*rule has-derivative-vec*)
subgoal for *i*
apply(*auto simp add*: *bounded-linear-Blinfun-apply good-interp-inverse good-interp*)
apply(*auto simp add*: *simple-term-inverse'*[*OF freef1*])
apply(*cases i = vid1*)
apply(*auto simp add*: *f1-def expand-singleton*)
proof −
let *?h* = (λ*b. Functions I fid1* (χ *i. sterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*) *b*))
let *?h'* = (λ*b'. FunctionFrechet I fid1* (χ *i. sterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*) *x*) (χ *i. frechet I* (*if i = vid1 then trm.Var vid1 else Const 0*) *x b'*))
let *?f* = (λ *b.* (χ *i. sterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*) *b*))
let *?f'* = (λ *b'.* (χ *i. frechet I* (*if i = vid1 then trm.Var vid1 else Const 0*) *x b'*))
let *?g* = *Functions I fid1*
let *?g'*= *FunctionFrechet I fid1* (*?f x*)

```
          have heq:?h = ?g ∘ ?f by(rule ext, auto)
          have heq':?h' = ?g' ∘ ?f' by(rule ext, auto)
          have fderiv:(?f has-derivative ?f') (at x)
            apply(rule has-derivative-vec)
            by (auto simp add: svar-deriv axis-def)
          have gderiv:(?g has-derivative ?g') (at (?f x))
            using good-interp unfolding is-interp-def by blast
          have gfderiv: ((?g ∘ ?f) has-derivative(?g' ∘ ?f')) (at x)
            using fderiv gderiv diff-chain-at by blast
          have boring-eq:(λb. Functions I fid1 (χ i. sterm-sem I (if i = vid1 then
trm.Var vid1 else Const 0) b)) =
            sterm-sem I ($f fid1 (λi. if i = vid1 then trm.Var vid1 else Const 0))
            by(rule ext, auto)
          have (?h has-derivative ?h') (at x) using gfderiv heq heq' by auto
          then show (sterm-sem I ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0)) has-derivative
(λv'. (THE f'. ∀x. (Functions I fid1 has-derivative f' x) (at x)) (χ i. sterm-sem I
(if i = vid1 then trm.Var vid1 else Const 0) x)
       (χ i. frechet I (if i = vid1 then trm.Var vid1 else Const 0) x v')))
 (at x)
            using boring-eq by auto
        qed
        done
    proof −
      have the-thing:continuous-on (UNIV::('sz Rvec set))
       (λb.
         blinfun-vec
          (λi. if i = vid1 then blin-frechet (good-interp I) (simple-term ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0))) b
              else Blinfun (λ-. 0)))
        apply(rule continuous-blinfun-vec')
        subgoal for i
          apply(cases i = vid1)
           apply(auto)
             using frechet-continuous[OF good-interp freef1] by (auto simp add:
continuous-on-const)
        done
      have another-cont:continuous-on (UNIV)
       (λx.
         blinfun-vec
          (λi. if i = vid1 then blin-frechet (good-interp I) (simple-term ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0))) (snd x)
              else Blinfun (λ-. 0)))
        apply(rule continuous-on-compose2[of UNIV (λb. blinfun-vec
          (λi. if i = vid1 then blin-frechet (good-interp I) (simple-term ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0))) b
              else Blinfun (λ-. 0)))])
          apply(rule the-thing)
         by (auto intro!: continuous-intros)
```

```
    have ext:(λx. case x of
     (t, b) ⇒
       blinfun-vec
       (λi. if i = vid1 then blin-frechet (good-interp I) (simple-term ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0))) b
            else Blinfun (λ-. 0))) =(λx.
       blinfun-vec
       (λi. if i = vid1 then blin-frechet (good-interp I) (simple-term ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0))) (snd x)
        else Blinfun (λ-. 0))) apply(rule ext, auto)
     by (metis snd-conv)
    then show continuous-on (UNIV)
     (λx. case x of
     (t, b) ⇒
       blinfun-vec
       (λi. if i = vid1 then blin-frechet (good-interp I) (simple-term ($f fid1 (λi.
if i = vid1 then trm.Var vid1 else Const 0))) b
            else Blinfun (λ-. 0)))
      using another-cont
      by (simp add: another-cont local.ext)
  qed
  have old-continuous: ⋀x. x ∈ UNIV ⟹ continuous-on UNIV (λt. χ i. if i =
vid1 then sterm-sem I (f1 fid1 vid1) x else 0)
    by(rule continuous-on-const)
  interpret ll-old: ll-on-open-it UNIV ?xode ?xconstraint 0
    apply(standard)
      subgoal by auto
     prefer 3 subgoal by auto
    prefer 3 subgoal by auto
    apply(rule old-lipschitz)
    by (rule old-continuous)
  let ?ivl = (ll-old.existence-ivl 0 (sol 0))
  let ?flow = ll-old.flow 0 (sol 0)
  have tclosed:{0..t} = {0−−t} using t real-Icc-closed-segment by auto
  have (sol solves-ode (λa b. χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) b
else 0)) {0..t} UNIV
    apply(rule solves-ode-supset-range)
    apply(rule sol)
    by auto
  then have sol':(sol solves-ode (λa b. χ i. if i = vid1 then sterm-sem I (f1 fid1
vid1) b else 0)) {0−−t} UNIV
    using tclosed by auto
  have sub:{0−−t} ⊆ ll-old.existence-ivl 0 (sol 0)
    apply(rule ll-old.closed-segment-subset-existence-ivl)
    apply(rule ll-old.existence-ivl-maximal-segment)
     apply(rule sol')
    apply(rule refl)
    by auto
  have usol-old:(?flow usolves-ode ?xode from 0) ?ivl UNIV
```

by(*rule ll-old.flow-usolves-ode*, *auto*)
have *sol-old*:(*ll-old.flow 0* (*sol 0*) *solves-ode ?xode*) *?ivl UNIV*
by(*rule ll-old.flow-solves-ode*, *auto*)
have *another-sub*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow \{s--0\} \subseteq \{0..t\}$
unfolding *closed-segment-def*
apply *auto*
by (*metis diff-0-right diff-left-mono mult.commute mult-left-le order.trans*)
have *sol-eq-flow*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *sol s = ?flow s*
using *usol-old* **apply** *simp*
apply(*drule usolves-odeD*(*4*))
apply *auto*
subgoal for *s x*
proof −
assume *xs0*:$x \in \{s--0\}$
assume *s0*:$0 \le s$ **and** *st*: $s \le t$
have $\{s--0\} \subseteq \{0..t\}$ **using** *another-sub*[*of s*] *s0 st* **by** *auto*
then have $x \in \{0..t\}$ **using** *xs0* **by** *auto*
then have $x \in \{0--t\}$ **using** *tclosed* **by** *auto*
then show *x* ∈ *ll-old.existence-ivl 0* (*sol 0*)
using *sub* **by** *auto*
qed
apply(*rule solves-ode-subset*)
using *sol′* **apply** *auto*[*1*]
subgoal for *s*
proof −
assume *s0*:$0 \le s$ **and** *st*:$s \le t$
show $\{s--0\} \subseteq \{0--t\}$
using *tclosed* **unfolding** *closed-segment* **using** *s0 st*
using *another-sub intervalE* **by** *blast*
qed
done
have *sol-deriv-orig*:$\bigwedge s.\ s \in$ *?ivl* $\Longrightarrow$ (*?flow has-derivative* (λ*xa. xa* $*_R$ (χ *i. if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (*?flow s*) *else 0*))) (*at s within ?ivl*)
using *sol-old* **apply** *simp*
apply(*drule solves-odeD*(*1*))
by (*auto simp add*: *has-vderiv-on-def has-vector-derivative-def*)
have *sol-eta*:(λ*t.* χ *i. ?flow t* $ *i*) = *?flow* **by**(*rule ext*, *rule vec-extensionality*, *auto*)
have *sol-deriv-eq1*:$\bigwedge s\ i.$ (λ*xa. xa* $*_R$ (χ *i. if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (*?flow s*) *else 0*)) = (λ*xa.* χ *i. xa* ∗ (*if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (*?flow s*) *else 0*))
by(*rule ext*, *rule vec-extensionality*, *auto*)
have *sol-deriv-proj*:$\bigwedge s\ i.\ s \in$ *?ivl* $\Longrightarrow$ ((λ*t. ?flow t* $ *i*) *has-derivative* (λ*xa.* (*xa* $*_R$ (χ *i. if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (*?flow s*) *else 0*)) $ *i*)) (*at s within ?ivl*)
subgoal for *s i*
apply(*rule has-derivative-proj*[*of* (λ *i t. ?flow t* $ *i*) (λ *i t′.* (*t′* $*_R$ (χ *i. if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (*?flow s*) *else 0*)) $ *i*) (*at s within ?ivl*) *i*])
using *sol-deriv-orig*[*of s*] *sol-eta sol-deriv-eq1* **by** *auto*

done
have *sol-deriv-eq2*:$\bigwedge s\ i.\ (\lambda xa.\ xa * (if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ s)\ else\ 0)) = (\lambda xa.\ (xa *_R (\chi\ i.\ if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ s)\ else\ 0))\ \$\ i)$
by(*rule ext*, *auto*)
have *sol-deriv-proj'*:$\bigwedge s\ i.\ s \in ?ivl \Longrightarrow ((\lambda t.\ ?flow\ t\ \$\ i)\ has\text{-}derivative\ (\lambda xa.\ xa * (if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ s)\ else\ 0)))\ (at\ s\ within\ ?ivl)$
subgoal for *s i* **using** *sol-deriv-proj*[*of s i*] *sol-deriv-eq2*[*of i s*] **by** *metis* **done**

have *sol-deriv-proj-vid1*:$\bigwedge s.\ s \in ?ivl \Longrightarrow ((\lambda t.\ ?flow\ t\ \$\ vid1)\ has\text{-}derivative\ (\lambda xa.\ xa * (sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ s))))\ (at\ s\ within\ ?ivl)$
subgoal for *s*
using *sol-deriv-proj'*[*of s vid1*] **by** *auto* **done**
have *deriv1-args*:$\bigwedge s.\ s \in ?ivl \Longrightarrow ((\lambda\ t.\ (\chi\ i.\ sterm\text{-}sem\ I\ (if\ i = vid1\ then\ trm.Var\ vid1\ else\ Const\ 0)\ (?flow\ t)))\ has\text{-}derivative\ ((\lambda\ t'.\ \chi\ i\ .\ t' * (if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ s)\ else\ 0))))\ (at\ s\ within\ ?ivl)$
apply(*rule has-derivative-vec*)
by (*auto simp add*: *sol-deriv-proj-vid1*)
have *con-fid*:$\bigwedge fid.\ continuous\text{-}on\ ?ivl\ (\lambda x.\ sterm\text{-}sem\ I\ (f1\ fid\ vid1)\ (?flow\ x))$
subgoal for *fid*
apply(*rule has-derivative-continuous-on*[*of ?ivl* $(\lambda x.\ sterm\text{-}sem\ I\ (f1\ fid\ vid1)\ (?flow\ x))$
$(\lambda t\ t'.\ FunctionFrechet\ I\ fid\ (\chi\ i.\ sterm\text{-}sem\ I\ (if\ i = vid1\ then\ trm.Var\ vid1\ else\ Const\ 0)\ (?flow\ t))\ (\chi\ i\ .\ t' * (if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ t)\ else\ 0)))$])
proof −
fix *s*
assume *ivl*:$s \in ?ivl$
let $?h = (\lambda x.\ sterm\text{-}sem\ I\ (f1\ fid\ vid1)\ (?flow\ x))$
let $?g = Functions\ I\ fid$
let $?f = (\lambda x.\ (\chi\ i.\ sterm\text{-}sem\ I\ (if\ i = vid1\ then\ trm.Var\ vid1\ else\ Const\ 0)\ (?flow\ x)))$
let $?h' = (\lambda t'.\ FunctionFrechet\ I\ fid\ (\chi\ i.\ sterm\text{-}sem\ I\ (if\ i = vid1\ then\ trm.Var\ vid1\ else\ Const\ 0)\ (?flow\ s))$
$(\chi\ i.\ t' * (if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ s)\ else\ 0)))$
let $?g' = FunctionFrechet\ I\ fid\ (?f\ s)$
let $?f' = (\lambda\ t'.\ \chi\ i\ .\ t' * (if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (?flow\ s)\ else\ 0))$
have *heq*:$?h = ?g \circ ?f$ **unfolding** *comp-def f1-def expand-singleton* **by** *auto*
have *heq'*:$?h' = ?g' \circ ?f'$ **unfolding** *comp-def* **by** *auto*
have *fderiv*:$(?f\ has\text{-}derivative\ ?f')\ (at\ s\ within\ ?ivl)$
using *deriv1-args*[*OF ivl*] **by** *auto*
have *gderiv*:$(?g\ has\text{-}derivative\ ?g')\ (at\ (?f\ s)\ within\ (?f\ `\ ?ivl))$
using *good-interp* **unfolding** *is-interp-def*
using *has-derivative-subset* **by** *blast*
have *gfderiv*:$((?g \circ ?f)\ has\text{-}derivative\ (?g' \circ ?f'))\ (at\ s\ within\ ?ivl)$
using *fderiv gderiv diff-chain-within* **by** *blast*
show $((\lambda x.\ sterm\text{-}sem\ I\ (f1\ fid\ vid1)\ (?flow\ x))\ has\text{-}derivative$
$(\lambda t'.\ FunctionFrechet\ I\ fid\ (\chi\ i.\ sterm\text{-}sem\ I\ (if\ i = vid1\ then\ trm.Var\ vid1$

else Const 0) (*?flow s*))
(χ *i*. *t'* ∗ (*if i* = *vid1 then sterm-sem I* (*f1 fid1 vid1*) (*?flow s*) *else 0*))))
(*at s within ?ivl*)
using *heq heq' gfderiv* **by** *auto*
qed
done
have *con*:⋀*x*. *continuous-on* (*?ivl*) (λ*t*. *x* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*?flow t*) + *sterm-sem I* (*f1 fid3 vid1*) (*?flow t*))
apply(*rule continuous-on-add*)
apply(*rule continuous-on-mult-left*)
apply(*rule con-fid*[*of fid2*])
by(*rule con-fid*[*of fid3*])
let *?axis* = (λ *i*. *Blinfun*(*axis i*))
have *bounded-linear-deriv*:⋀*t*. *bounded-linear* (λ*y'* . *y'* ∗$_R$ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*))
using *bounded-linear-scaleR-left* **by** *blast*
have *ll*:*local-lipschitz* (*ll-old.existence-ivl 0* (*sol 0*)) *UNIV* (λ*t y*. *y* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*?flow t*) + *sterm-sem I* (*f1 fid3 vid1*) (*?flow t*))
apply(*rule c1-implies-local-lipschitz*[**where** *f'*=(λ (*t*,*y*). *Blinfun*(λ*y'* . *y'* ∗$_R$ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*)))])
apply *auto*
subgoal for *t x*
apply(*rule has-derivative-add-const*)
proof −
have *deriv*:((λ*x*. *x* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*)) *has-derivative* (λ*x*. *x* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*))) (*at x*)
by(*auto intro*: *derivative-eq-intros*)
have *eq*:(λ*x*. *x* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*)) = *blinfun-apply* (*Blinfun* (λ*y'*. *y'* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*)))
apply(*rule ext*)
using *bounded-linear-deriv*[*of t*] **by** (*auto simp add*: *bounded-linear-Blinfun-apply*)
show ((λ*x*. *x* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*)) *has-derivative*
blinfun-apply (*Blinfun* (λ*y'*. *y'* ∗ *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*))))
(*at x*) **using** *deriv eq* **by** *auto*
qed
apply(*auto intro*: *continuous-intros simp add*: *split-beta'*)
proof −
have *bounded-linear*:⋀*x*. *bounded-linear* (λ*y'*. *y'* ∗ *sterm-sem I* (*f1 fid2 vid1*) *x*)
by (*simp add*: *bounded-linear-mult-left*)
have *eq*:(λ*x*. *Blinfun* (λ*y'*. *y'* ∗ *sterm-sem I* (*f1 fid2 vid1*) *x*)) = (λ*x*. (*sterm-sem I* (*f1 fid2 vid1*) *x*) ∗$_R$ *id-blinfun*)
apply(*rule ext*, *rule blinfun-eqI*)
subgoal for *x i*
using *bounded-linear*[*of x*] **apply**(*auto simp add*: *bounded-linear-Blinfun-apply*)
by (*simp add*: *blinfun.scaleR-left*)

```
      done
    have conFlow:continuous-on (ll-old.existence-ivl 0 (sol 0)) (ll-old.flow 0 (sol
0))
      using ll-old.general.flow-continuous-on by blast
    have conF':continuous-on (ll-old.flow 0 (sol 0) ‘ ll-old.existence-ivl 0 (sol 0))

        (λx. (sterm-sem I (f1 fid2 vid1) x) ∗R id-blinfun)
      apply(rule continuous-on-scaleR)
       apply(auto intro: continuous-intros)
      apply(rule sterm-continuous')
       apply(rule good-interp)
      by(auto simp add: f1-def intro: dfree.intros)
    have conF:continuous-on (ll-old.flow 0 (sol 0) ‘ ll-old.existence-ivl 0 (sol 0))

        (λx. Blinfun (λy'. y' ∗ sterm-sem I (f1 fid2 vid1) x))
       apply(rule continuous-on-compose2[of UNIV (λx. Blinfun (λy'. y' ∗ x))
(ll-old.flow 0 (sol 0) ‘ ll-old.existence-ivl 0 (sol 0)) sterm-sem I (f1 fid2 vid1)])
        subgoal by (metis blinfun-mult-left.abs-eq bounded-linear-blinfun-mult-left
continuous-on-eq linear-continuous-on)
       apply(rule sterm-continuous')
        apply(rule good-interp)
       by(auto simp add: f1-def intro: dfree.intros)
    show continuous-on (ll-old.existence-ivl 0 (sol 0) × UNIV) (λx. Blinfun (λy'.
y' ∗ sterm-sem I (f1 fid2 vid1) (ll-old.flow 0 (sol 0) (fst x))))
     apply(rule continuous-on-compose2[of ll-old.existence-ivl 0 (sol 0) (λx. Blin-
fun (λy'. y' ∗ sterm-sem I (f1 fid2 vid1) (ll-old.flow 0 (sol 0) x))) (ll-old.existence-ivl
0 (sol 0) × UNIV) fst])
      apply(rule continuous-on-compose2[of (ll-old.flow 0 (sol 0) ‘ ll-old.existence-ivl
0 (sol 0)) (λx. Blinfun (λy'. y' ∗ sterm-sem I (f1 fid2 vid1) x))
            (ll-old.existence-ivl 0 (sol 0)) (ll-old.flow 0 (sol 0))])
         using conF conFlow by (auto intro!: continuous-intros)
    qed
  let ?ivl = ll-old.existence-ivl 0 (sol 0)
  — Construct solution to ODE for y' here:
  let ?yode = (λt y. y ∗ sterm-sem I (f1 fid2 vid1) (?flow t) + sterm-sem I (f1
fid3 vid1) (?flow t))
  let ?ysol0 = r
  interpret ll-new: ll-on-open-it ?ivl ?yode UNIV 0
    apply(standard)
      apply(auto)
     apply(rule ll)
    by(rule con)
  have sol-new:(ll-new.flow 0 r solves-ode ?yode) (ll-new.existence-ivl 0 r) UNIV
    by(rule ll-new.flow-solves-ode, auto)
  have more-lipschitz:⋀tm tM. tm ∈ ll-old.existence-ivl 0 (sol 0) ⟹
      tM ∈ ll-old.existence-ivl 0 (sol 0) ⟹
       ∃ M L. ∀ t∈{tm..tM}. ∀ x. |x ∗ sterm-sem I (f1 fid2 vid1) (?flow t) +
sterm-sem I (f1 fid3 vid1) (?flow t)| ≤ M + L ∗ |x|
  proof −
```

fix *tm tM*
assume *tm*:*tm* ∈ *ll-old.existence-ivl 0* (*sol 0*)
assume *tM*:*tM* ∈ *ll-old.existence-ivl 0* (*sol 0*)
let *?f2* = (λ*t*. *sterm-sem I* (*f1 fid2 vid1*) (*ll-old.flow 0* (*sol 0*) *t*))
let *?f3* = (λ*t*. *sterm-sem I* (*f1 fid3 vid1*) (*ll-old.flow 0* (*sol 0*) *t*))
let *?boundLP* = (λ*L t* . (*tm* ≤ *t* ∧ *t* ≤ *tM* ⟶ |*?f2 t*| ≤ *L*))
let *?boundL* = (*SOME L*. (∀ *t*. *?boundLP L t*))
have *compactT*:*compact* {*tm*..*tM*} **by** *auto*
have *sub*:{*tm*..*tM*} ⊆ *ll-old.existence-ivl 0* (*sol 0*)
by (*metis atLeastatMost-empty-iff empty-subsetI ll-old.general.segment-subset-existence-ivl real-Icc-closed-segment tM tm*)
let *?f2abs* = (λ*x*. *abs*(*?f2 x*))
have *neg-compact*:⋀*S*::*real set*. *compact S* ⟹ *compact* ((λ*x*. −*x*) ‘ *S*)
by(*rule compact-continuous-image*, *auto intro*: *continuous-intros*)
have *compactf2*:*compact* (*?f2* ‘ {*tm*..*tM*})
apply(*rule compact-continuous-image*)
apply(*rule continuous-on-compose2*[*of UNIV sterm-sem I* (*f1 fid2 vid1*) {*tm*..*tM*} *ll-old.flow 0* (*sol 0*)])
apply(*rule sterm-continuous*)
apply(*rule good-interp*)
subgoal by (*auto intro*: *dfree.intros simp add*: *f1-def*)
apply(*rule continuous-on-subset*)
prefer *2* **apply** (*rule sub*)
subgoal using *ll-old.general.flow-continuous-on* **by** *blast*
by *auto*
then have *boundedf2*:*bounded* (*?f2* ‘ {*tm*..*tM*}) **using** *compact-imp-bounded* **by** *auto*
then have *boundedf2neg*:*bounded* ((λ*x*. −*x*) ‘ *?f2* ‘ {*tm*..*tM*}) **using** *compact-imp-bounded neg-compact* **by** *auto*
then have *bdd-above-f2neg*:*bdd-above* ((λ*x*. −*x*) ‘ *?f2* ‘ {*tm*..*tM*}) **by** (*rule bounded-imp-bdd-above*)
then have *bdd-above-f2*:*bdd-above* (*?f2* ‘ {*tm*..*tM*}) **using** *bounded-imp-bdd-above boundedf2* **by** *auto*
have *bdd-above-f2-abs*:*bdd-above* (*abs* ‘ *?f2* ‘ {*tm*..*tM*})
using *bdd-above-f2neg bdd-above-f2* **unfolding** *bdd-above-def*
apply *auto*
subgoal for *M1 M2*
apply(*rule exI*[**where** *x*=*max M1 M2*])
by *fastforce*
done
then have *theBound*:∃ *L*. (∀ *t*. *?boundLP L t*)
unfolding *bdd-above-def norm-conv-dist*
by (*auto simp add*: *Ball-def Bex-def norm-conv-dist image-iff norm-bcontfun-def dist-blinfun-def*)
then have *boundLP*:∀ *t*. *?boundLP* (*?boundL*) *t* **using** *someI*[*of* (λ *L*. ∀ *t*. *?boundLP L t*)] **by** *blast*
let *?boundMP* = (λ*M t*. (*tm* ≤ *t* ∧ *t* ≤ *tM* ⟶ |*?f3 t*| ≤ *M*))
let *?boundM* = (*SOME M*. (∀ *t*. *?boundMP M t*))
have *compactf3*:*compact* (*?f3* ‘ {*tm*..*tM*})

apply(*rule compact-continuous-image*)
apply(*rule continuous-on-compose2*[*of UNIV sterm-sem I* (*f1 fid3 vid1*) {*tm..tM*} *ll-old.flow 0* (*sol 0*)])
apply(*rule sterm-continuous*)
apply(*rule good-interp*)
subgoal by (*auto intro*: *dfree.intros simp add*: *f1-def*)
apply(*rule continuous-on-subset*)
prefer *2* **apply** (*rule sub*)
subgoal using *ll-old.general.flow-continuous-on* **by** *blast*
by *auto*
then have *boundedf3*:*bounded* (*?f3* ‘ {*tm..tM*}) **using** *compact-imp-bounded* **by** *auto*
then have *boundedf3neg*:*bounded* ((λ*x*. −*x*) ‘ *?f3* ‘ {*tm..tM*}) **using** *compact-imp-bounded neg-compact* **by** *auto*
then have *bdd-above-f3neg*:*bdd-above* ((λ*x*. −*x*) ‘ *?f3* ‘ {*tm..tM*}) **by** (*rule bounded-imp-bdd-above*)
then have *bdd-above-f3*:*bdd-above* (*?f3* ‘{*tm..tM*}) **using** *bounded-imp-bdd-above boundedf3* **by** *auto*
have *bdd-above-f3-abs*:*bdd-above* (*abs* ‘ *?f3* ‘ {*tm..tM*})
using *bdd-above-f3neg bdd-above-f3* **unfolding** *bdd-above-def*
apply *auto*
subgoal for *M1 M2*
apply(*rule exI*[**where** *x*=*max M1 M2*])
by *fastforce*
done
then have *theBound*:∃ *L*. (∀ *t*. *?boundMP L t*)
unfolding *bdd-above-def norm-conv-dist*
by (*auto simp add*: *Ball-def Bex-def norm-conv-dist image-iff norm-bcontfun-def dist-blinfun-def*)
then have *boundMP*:∀ *t*. *?boundMP* (*?boundM*) *t* **using** *someI*[*of* (λ *M*. ∀ *t*. *?boundMP M t*)] **by** *blast*
show ∃ *M L*. ∀ *t*∈{*tm..tM*}. ∀ *x*. |*x* ∗ *?f2 t* + *?f3 t*| ≤ *M* + *L* ∗ |*x*|
apply(*rule exI*[**where** *x*=*?boundM*])
apply(*rule exI*[**where** *x*=*?boundL*])
apply *auto*
proof −
fix *t* **and** *x* :: *real*
assume *ttm*:*tm* ≤ *t*
assume *ttM*:*t* ≤ *tM*
from *ttm ttM* **have** *ttmM*:*tm* ≤ *t* ∧ *t* ≤ *tM* **by** *auto*
have *leqf3*:|*?f3 t*| ≤ *?boundM* **using** *boundMP ttmM* **by** *auto*
have *leqf2*:|*?f2 t*| ≤ *?boundL* **using** *boundLP ttmM* **by** *auto*
have *gr0*: |*x*| ≥ *0* **by** *auto*
have *leqf2x*:|*?f2 t*| ∗ |*x*| ≤ *?boundL* ∗ |*x*| **using** *gr0 leqf2*
by (*metis* (*no-types*, *lifting*) *real-scaleR-def scaleR-right-mono*)
have |*x* ∗ *?f2 t* + *?f3 t*| ≤ |*x*| ∗ |*?f2 t*| + |*?f3 t*|
proof −
have *f1*: ⋀*r ra*. |*r*::*real*| ∗ |*ra*| = |*r* ∗ *ra*|
by (*metis norm-scaleR real-norm-def real-scaleR-def*)

```
        have ⋀r ra. |(r::real) + ra| ≤ |r| + |ra|
          by (metis norm-triangle-ineq real-norm-def)
          then show ?thesis
          using f1 by presburger
        qed
      moreover have ... = |?f3 t| + |?f2 t| * |x|
        by auto
      moreover have ... ≤ ?boundM + |?f2 t| * |x|
        using leqf3 by linarith
      moreover have ... ≤ ?boundM + ?boundL * |x|
        using leqf2x  by linarith
      ultimately show |x * ?f2 t + ?f3 t| ≤ ?boundM + ?boundL * |x|
        by linarith
    qed
  qed
  have ivls-eq:(ll-new.existence-ivl 0 r) = (ll-old.existence-ivl 0 (sol 0))
    apply(rule ll-new.existence-ivl-eq-domain)
      apply auto
    apply (rule more-lipschitz)
    by auto
  have sub':{0--t} ⊆ ll-new.existence-ivl 0 r
    using sub ivls-eq by auto
  have sol-new':(ll-new.flow 0 r solves-ode ?yode) {0--t} UNIV
    by(rule solves-ode-subset, rule sol-new, rule sub')
  let ?soly = ll-new.flow 0 r
  let ?sol' = (λt. χ i. if i = vid2 then ?soly t else sol t $ i)
  let ?aaba' = mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0)))
                        (OSing vid2
                          (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1
else Const 0)) (trm.Var vid2))
                                ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const
0)))))
                      (χ y. if vid2 = y then r else fst (a, b) $ y, b)
                      (?sol' t)
  have duh:(fst ?aaba', snd ?aaba') = ?aaba' by auto
   note bigEx = spec[OF spec[OF bigAll, where x=fst ?aaba'], where x=snd
?aaba']
  have sol-deriv:⋀s. s ∈ {0..t} ⟹ (sol has-derivative (λxa. xa *R (χ i. if i =
vid1 then sterm-sem I (f1 fid1 vid1) (sol s) else 0))) (at s within {0..t})
    using sol apply simp
   by(drule solves-odeD(1), auto simp add: has-vderiv-on-def has-vector-derivative-def)
   have silly-eq1:(λt. χ i. sol t $ i) = sol
    by(rule ext, rule vec-extensionality, auto)
   have silly-eq2:⋀s. (λxa. χ i. (xa *R (χ i. if i = vid1 then sterm-sem I (f1 fid1
vid1) (sol s) else 0)) $ i) = (λxa. xa *R (χ i. if i = vid1 then sterm-sem I (f1 fid1
vid1) (sol s) else 0))
    by(rule ext, rule vec-extensionality, auto)
   have sol-proj-deriv:⋀s i. s ∈ {0..t} ⟹ ((λ t. sol t $ i) has-derivative (λxa.
```

$(xa *_R (\chi\ i.\ if\ i = vid1\ then\ sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (sol\ s)\ else\ 0))\ \$\ i))\ (at\ s\ within\ \{0..t\})$

subgoal for *s i*

apply(*rule has-derivative-proj*)

using *sol-deriv*[*of s*] *silly-eq1 silly-eq2*[*of s*] **by** *auto*

done

have *sol-proj-deriv-vid1*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow ((\lambda\ t.\ sol\ t\ \$\ vid1)\ has\text{-}derivative\ (\lambda xa.\ xa * sterm\text{-}sem\ I\ (f1\ fid1\ vid1)\ (sol\ s)))\ (at\ s\ within\ \{0..t\})$

subgoal for *s* **using** *sol-proj-deriv*[*of s vid1*] **by** *auto* **done**

have *sol-proj-deriv-other*:$\bigwedge s\ i.\ s \in \{0..t\} \Longrightarrow i \neq vid1 \Longrightarrow ((\lambda\ t.\ sol\ t\ \$\ i)\ has\text{-}derivative\ (\lambda xa.\ 0))\ (at\ s\ within\ \{0..t\})$

subgoal for *s i* **using** *sol-proj-deriv*[*of s i*] **by** *auto* **done**

have *fact*:$\bigwedge x.\ x \in \{0..t\} \Longrightarrow$
$(ll\text{-}new.flow\ 0\ r\ has\text{-}derivative$
$(\lambda xa.\ xa *_R (ll\text{-}new.flow\ 0\ r\ x * sterm\text{-}sem\ I\ (f1\ fid2\ vid1)\ (ll\text{-}old.flow\ 0\ (sol\ 0)\ x) +$
$sterm\text{-}sem\ I\ (f1\ fid3\ vid1)\ (ll\text{-}old.flow\ 0\ (sol\ 0)\ x))))$
$(at\ x\ within\ \{0\ ..\ t\})$

using *sol-new′* **apply** *simp*

apply(*drule solves-odeD*(*1*))

using *tclosed* **unfolding** *has-vderiv-on-def has-vector-derivative-def* **by** *auto*

have *new-sol-deriv*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow (ll\text{-}new.flow\ 0\ r\ has\text{-}derivative$
$(\lambda xa.\ xa *_R (ll\text{-}new.flow\ 0\ r\ s * sterm\text{-}sem\ I\ (f1\ fid2\ vid1)\ (sol\ s) + sterm\text{-}sem\ I\ (f1\ fid3\ vid1)\ (sol\ s))))$
$(at\ s\ within\ \{0..\ t\})$

subgoal for *s*

using *fact*[*of s*] *tclosed sol-eq-flow*[*of s*] **by** *auto*

done

have *sterm-agree*:$\bigwedge s.\ Vagree\ (\chi\ i.\ if\ i = vid2\ then\ ll\text{-}new.flow\ 0\ r\ s\ else\ sol\ s\ \$\ i,\ undefined)\ (sol\ s,\ undefined)\ \{Inl\ vid1\}$

subgoal for *s* **unfolding** *Vagree-def* **using** *vne12* **by** *auto* **done**

have *FVF*:$(FVT\ (f1\ fid2\ vid1)) = \{Inl\ vid1\}$ **unfolding** *f1-def expand-singleton* **apply** *auto* **subgoal for** *x xa* **by** (*cases xa = vid1*, *auto*) **done**

have *FVF2*:$(FVT\ (f1\ fid3\ vid1)) = \{Inl\ vid1\}$ **unfolding** *f1-def expand-singleton* **apply** *auto* **subgoal for** *x xa* **by** (*cases xa = vid1*, *auto*) **done**

have *sterm-agree-FVF*:$\bigwedge s.\ Vagree\ (\chi\ i.\ if\ i = vid2\ then\ ll\text{-}new.flow\ 0\ r\ s\ else\ sol\ s\ \$\ i,\ undefined)\ (sol\ s,\ undefined)\ (FVT\ (f1\ fid2\ vid1))$

using *sterm-agree FVF* **by** *auto*

have *sterm-agree-FVF2*:$\bigwedge s.\ Vagree\ (\chi\ i.\ if\ i = vid2\ then\ ll\text{-}new.flow\ 0\ r\ s\ else\ sol\ s\ \$\ i,\ undefined)\ (sol\ s,\ undefined)\ (FVT\ (f1\ fid3\ vid1))$

using *sterm-agree FVF2* **by** *auto*

have *y-component-sem-eq2*:$\bigwedge s.\ sterm\text{-}sem\ I\ (f1\ fid2\ vid1)\ (\chi\ i.\ if\ i = vid2\ then\ ll\text{-}new.flow\ 0\ r\ s\ else\ sol\ s\ \$\ i)$
$= sterm\text{-}sem\ I\ (f1\ fid2\ vid1)\ (sol\ s)$

using *coincidence-sterm*[*OF sterm-agree-FVF*, *of I*] **by** *auto*

have *y-component-sem-eq3*:$\bigwedge s.\ sterm\text{-}sem\ I\ (f1\ fid3\ vid1)\ (\chi\ i.\ if\ i = vid2\ then\ ll\text{-}new.flow\ 0\ r\ s\ else\ sol\ s\ \$\ i)$
$= sterm\text{-}sem\ I\ (f1\ fid3\ vid1)\ (sol\ s)$

using *coincidence-sterm*[*OF sterm-agree-FVF2*, *of I*] **by** *auto*

have *y-component-ode-eq*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
(*λxa. xa ∗ (ll-new.flow 0 r s ∗ sterm-sem I (f1 fid2 vid1) (sol s) + sterm-sem I (f1 fid3 vid1) (sol s)*))
= (*λxa. xa ∗ (sterm-sem I (f1 fid2 vid1) (χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i) ∗ ll-new.flow 0 r s +*
sterm-sem I (f1 fid3 vid1) (χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i)))
subgoal for *s*
apply(*rule ext*)
subgoal for *xa*
using *y-component-sem-eq2 y-component-sem-eq3* **by** *auto*
done
done
have *agree-vid1*:$\bigwedge s.$ *Vagree* (*sol s*, *undefined*) (*χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i*, *undefined*) {*Inl vid1*}
unfolding *Vagree-def* **using** *vne12* **by** *auto*
have *FVT-vid1*:*FVT*(*f1 fid1 vid1*) = {*Inl vid1*} **apply**(*auto simp add*: *f1-def*) **subgoal for** *x xa* **apply**(*cases xa = vid1*) **by** *auto* **done**
have *agree-vid1-FVT*:$\bigwedge s.$ *Vagree* (*sol s*, *undefined*) (*χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i*, *undefined*) (*FVT* (*f1 fid1 vid1*))
using *FVT-vid1 agree-vid1* **by** *auto*
have *sterm-eq-vid1*:$\bigwedge s.$ *sterm-sem I* (*f1 fid1 vid1*) (*sol s*) = *sterm-sem I* (*f1 fid1 vid1*) (*χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i*)
subgoal for *s*
using *coincidence-sterm*[*OF agree-vid1-FVT*[*of s*], *of I*] **by** *auto*
done
have *vid1-deriv-eq*:$\bigwedge s.$ (*λxa. xa ∗ sterm-sem I* (*f1 fid1 vid1*) (*sol s*)) =
(*λxa. xa ∗ sterm-sem I* (*f1 fid1 vid1*) (*χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i*))
subgoal for *s*
apply(*rule ext*)
subgoal for *x′*
using *sterm-eq-vid1*[*of s*] **by** *auto*
done done
have *inner-deriv*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
((*λt. χ i. if i = vid2 then ll-new.flow 0 r t else sol t $ i*) *has-derivative* (*λxa.* (*χ i. xa ∗ (if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (*χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i*) *else*
if i = vid2 then sterm-sem I (*Plus* (*Times* (*f1 fid2 vid1*) (*trm.Var vid2*)) (*f1 fid3 vid1*)) (*χ i. if i = vid2 then ll-new.flow 0 r s else sol s $ i*) *else 0*))))
(*at s within* {*0..t*}))
subgoal for *s*
apply(*rule has-derivative-vec*)
subgoal for *i*
apply(*cases i = vid2*)
subgoal
using *vne12*
using *new-sol-deriv*[*of s*]

using *y-component-ode-eq* **by** *auto*
subgoal
apply(*cases i = vid1*)
using *sol-proj-deriv-vid1*[*of s*] *vid1-deriv-eq*[*of s*] *sol-proj-deriv-other*[*of s i*] **by** *auto*
done
done
done
have *deriv-eta*:⋀*s*. (λ*xa*. *xa* ∗$_R$ ((χ *i*. *if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* $ *i*) *else 0*) +
(χ *i*. *if i = vid2*
then sterm-sem I (*Plus* (*Times* (*f1 fid2 vid1*) (*trm.Var vid2*)) (*f1 fid3 vid1*))
(χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* $ *i*)
else 0)))
= (λ*xa*. (χ *i*. *xa* ∗ (*if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* $ *i*) *else*
if i = vid2 then sterm-sem I (*Plus* (*Times* (*f1 fid2 vid1*) (*trm.Var vid2*)) (*f1 fid3 vid1*)) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* $ *i*) *else 0*)))
subgoal for *s*
apply(*rule ext*)
apply(*rule vec-extensionality*)
using *vne12* **by** *auto*
done
have *sol′-deriv*:⋀*s*. *s* ∈ {*0..t*} ⟹
((λ*t*. χ *i*. *if i = vid2 then ll-new.flow 0 r t else sol t* $ *i*) *has-derivative*
(λ*xa*. *xa* ∗$_R$ ((χ *i*. *if i = vid1 then sterm-sem I* (*f1 fid1 vid1*) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* $ *i*) *else 0*) +
(χ *i*. *if i = vid2*
then sterm-sem I (*Plus* (*Times* (*f1 fid2 vid1*) (*trm.Var vid2*)) (*f1 fid3 vid1*))
(χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* $ *i*)
else 0))))
(*at s within* {*0..t*})
subgoal for *s*
using *inner-deriv*[*of s*] *deriv-eta*[*of s*] **by** *auto* **done**
have *FVT*:⋀*i*. *FVT* (*if i = vid1 then trm.Var vid1 else Const 0*) ⊆ {*Inl vid1*} **by** *auto*
have *agree*:⋀*s*. *Vagree* (*mk-v I* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol s*)) (*mk-v I* (*OProd* (*OSing vid1* ($*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* ($*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
($*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2 = y then r else fst* (*a*, *b*) $ *y*, *b*) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* $ *i*)) {*Inl vid1*}
subgoal for *s*

using *mk-v-agree* [*of I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol s*)]

using *mk-v-agree* [*of I* (*OProd* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* (\$*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(\$*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))))) (χ *y*. *if vid2 = y then r else fst* (*a*, *b*) \$ *y*, *b*) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* \$ *i*)]

unfolding *Vagree-def* **using** *vne12* **by** *simp*

done

have *agree'*:⋀*s i*. *Vagree* (*mk-v I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol s*)) (*mk-v I* (*OProd* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* (\$*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(\$*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2 = y then r else fst* (*a*, *b*) \$ *y*, *b*) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* \$ *i*)) (*FVT* (*if i = vid1 then trm.Var vid1 else Const 0*))

subgoal for *s i* **using** *agree-sub*[*OF FVT*[*of i*] *agree*[*of s*]] **by** *auto* **done**

have *safe*:⋀*i*. *dsafe* (*if i = vid1 then trm.Var vid1 else Const 0*) **subgoal for** *i* **apply**(*cases i = vid1*, *auto*) **done done**

have *dterm-sem-eq*:⋀*s i*. *dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*) (*mk-v I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol s*))
= *dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OProd* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* (\$*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(\$*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))))
(χ *y*. *if vid2 = y then r else fst* (*a*, *b*) \$ *y*, *b*) (χ *i*. *if i = vid2 then ll-new.flow 0 r s else sol s* \$ *i*))

subgoal for *s i* **using** *coincidence-dterm*[*OF safe*[*of i*] *agree'*[*of s i*], *of I*] **by** *auto* **done**

have *dterm-vec-eq*:⋀*s*. (χ *i*. *dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*) (*mk-v I* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol s*)))
= (χ *i*. *dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OProd* (*OSing vid1* (\$*f fid1* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))
(*OSing vid2*
(*Plus* (*Times* (\$*f fid2* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)) (*trm.Var vid2*))
(\$*f fid3* (λ*i*. *if i = vid1 then trm.Var vid1 else Const 0*)))))

```
        (χ y. if vid2 = y then r else fst (a, b) $ y, b) (χ i. if i = vid2 then
ll-new.flow 0 r s else sol s $ i)))
    subgoal for s
      apply(rule vec-extensionality)
      subgoal for i using dterm-sem-eq[of i s] by auto
      done done
  have pred-same:⋀s. s ∈ {0..t} ⟹ Predicates I vid1
    (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
          (mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0))) (a, b) (sol s))) ⟹
Predicates I vid1
 (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
     (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1 else
Const 0)))
              (OSing vid2
               (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const
0)) (trm.Var vid2))
                 ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
        (χ y. if vid2 = y then r else fst (a, b) $ y, b) (χ i. if i = vid2 then
ll-new.flow 0 r s else sol s $ i)))
    subgoal for s using dterm-vec-eq[of s] by auto done
  have sol'-domain:⋀s. 0 ≤ s ⟹
 s ≤ t ⟹
 Predicates I vid1
  (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
       (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var vid1
else Const 0)))
                (OSing vid2
                 (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else
Const 0)) (trm.Var vid2))
                   ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
                   (χ y. if vid2 = y then r else fst (a, b) $ y, b) (χ i. if i = vid2
then ll-new.flow 0 r s else sol s $ i)))
    subgoal for s
      using sol apply simp
      apply(drule solves-odeD(2))
      using pred-same[of s] by auto
    done
  have sol':(?sol' solves-ode
(λa b. (χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) b else 0) +
       (χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2 vid1) (trm.Var
vid2)) (f1 fid3 vid1)) b else 0)))
{0..t} {x. Predicates I vid1
          (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
               (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0)))
                           (OSing vid2
                            (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1
else Const 0)) (trm.Var vid2))
```

```
                                    ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const
0)))))
                          (χ y. if vid2 = y then r else fst (a, b) $ y, b) x))}
      apply(rule solves-odeI)
      subgoal
        unfolding has-vderiv-on-def has-vector-derivative-def
        using sol'-deriv by auto
      by(auto, rule sol'-domain, auto)
    have set-eq:{y. y = vid2 ∨ y = vid1 ∨ y = vid2 ∨ y = vid1 ∨ (∃ x. Inl y ∈
FVT (if x = vid1 then trm.Var vid1 else Const 0))} = {vid1, vid2}
      by auto
   have VSagree (?sol' 0) (χ y. if vid2 = y then r else fst (a, b) $ y) {vid1, vid2}
     using VSA unfolding VSagree-def by simp
    then have VSA': VSagree (?sol' 0) (χ y. if vid2 = y then r else fst (a, b) $
y)

 {y. y = vid2 ∨ y = vid1 ∨ y = vid2 ∨ y = vid1 ∨ (∃ x. Inl y ∈ FVT (if x =
vid1 then trm.Var vid1 else Const 0))}
      by (auto simp add: set-eq)
    have bigPre:(∃ sol t. (fst ?aaba', snd ?aaba') =
                  mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0)))
                                (OSing vid2
                                  (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1
else Const 0)) (trm.Var vid2))
                                        ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const
0)))))
                   ((χ y. if vid2 = y then r else fst (a,b) $ y), b) (sol t) ∧
                  0 ≤ t ∧
                  (sol solves-ode
                  (λa b. (χ i. if i = vid1 then sterm-sem I (f1 fid1 vid1) b else 0) +
                                (χ i. if i = vid2 then sterm-sem I (Plus (Times (f1 fid2
vid1) (trm.Var vid2)) (f1 fid3 vid1)) b else 0)))
                   {0..t} {x. Predicates I vid1
                                (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else
Const 0)
                                      (mk-v I (OProd (OSing vid1 ($f fid1 (λi. if i =
vid1 then trm.Var vid1 else Const 0)))
                                                     (OSing vid2
                                                       (Plus (Times ($f fid2 (λi. if i = vid1
then trm.Var vid1 else Const 0)) (trm.Var vid2))
                                                             ($f fid3 (λi. if i = vid1 then trm.Var
vid1 else Const 0)))))
                                      ((χ y. if vid2 = y then r else (fst (a,b)) $ y), b)
x))} ∧
                  VSagree (sol 0) (χ y. if vid2 = y then r else fst (a,b) $ y)
                   {uu. uu = vid2 ∨
                  uu = vid1 ∨
                  uu = vid2 ∨
```

```
                uu = vid1 ∨
                Inl uu ∈ Inl ‘ ({x. ∃ xa. Inl x ∈ FVT (if xa = vid1 then trm.Var
vid1 else Const 0)} ∪
                                {x. x = vid2 ∨ (∃ xa. Inl x ∈ FVT (if xa = vid1
then trm.Var vid1 else Const 0))}) ∨
              (∃ x. Inl uu ∈ FVT (if x = vid1 then trm.Var vid1 else Const 0))})
    apply(rule exI[where x=?sol'])
    apply(rule exI[where x=t])
    apply(rule conjI)
     subgoal by simp
    apply(rule conjI)
     subgoal by (rule t)
    apply(rule conjI)
     apply(rule sol')
     using VSA' unfolding VSagree-def by auto
   have pred-sem:Predicates I vid2 (χ i. dterm-sem I (if i = vid1 then trm.Var
vid1 else Const 0) ?aaba')
    using mp[OF bigEx bigPre] by auto
   let ?other-state = (mk-v I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0))) (a, b) (sol t))
   have agree:Vagree (?aaba') (?other-state) {Inl vid1}
     using mk-v-agree [of I (OProd (OSing vid1 ($f fid1 (λi. if i = vid1 then
trm.Var vid1 else Const 0)))
           (OSing vid2
            (Plus (Times ($f fid2 (λi. if i = vid1 then trm.Var vid1 else Const
0)) (trm.Var vid2))
             ($f fid3 (λi. if i = vid1 then trm.Var vid1 else Const 0)))))
      (χ y. if vid2 = y then r else fst (a, b) $ y, b) (?sol' t)]
     using mk-v-agree [of I (OSing vid1 ($f fid1 (λi. if i = vid1 then trm.Var
vid1 else Const 0))) (a, b) (sol t)]
    unfolding Vagree-def using vne12 by simp
   have sub:⋀i. FVT (if i = vid1 then trm.Var vid1 else Const 0) ⊆ {Inl vid1}
    by auto
   have agree':⋀i. Vagree (?aaba') (?other-state) (FVT (if i = vid1 then trm.Var
vid1 else Const 0))
    subgoal for i using agree-sub[OF sub[of i] agree] by auto done
   have silly-safe:⋀i. dsafe (if i = vid1 then trm.Var vid1 else Const 0)
    subgoal for i
     apply(cases i = vid1)
      by (auto simp add: dsafe-Var dsafe-Const)
    done
   have dsem-eq:(χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0)
?aaba') =
     (χ i. dterm-sem I (if i = vid1 then trm.Var vid1 else Const 0) ?other-state)
    apply(rule vec-extensionality)
    subgoal for i
     using coincidence-dterm[OF silly-safe[of i] agree'[of i], of I] by auto
    done
   show
```

Predicates I vid2
(*χ i. dterm-sem I* (*if i = vid1 then trm.Var vid1 else Const 0*)
(*mk-v I* (*OSing vid1* (*$f fid1* (*λi. if i = vid1 then trm.Var vid1 else Const 0*))) (*a*, *b*) (*sol t*)))
using *pred-sem dsem-eq* **by** *auto*
qed

done
qed
end end
theory *USubst*
imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
Static-Semantics
begin

10 Uniform Substitution Definitions

This section defines substitutions and implements the substitution operation. Every part of substitution comes in two flavors. The "Nsubst" variant of each function returns a term/formula/ode/program which (as encoded in the type system) has less symbols that the input. We use this operation when substitution into functions and function-like constructs to make it easy to distinguish identifiers that stand for arguments to functions from other identifiers. In order to expose a simpler interface, we also have a "subst" variant which does not delete variables.

Naive substitution without side conditions would not always be sound. The various admissibility predicates *admit describe conditions under which the various substitution operations are sound.

Explicit data structure for substitutions.

The RHS of a function or predicate substitution is a term or formula with extra variables, which are used to refer to arguments.

record (*'a*, *'b*, *'c*) *subst* =
SFunctions :: *'a* ⇀ (*'a* + *'c*, *'c*) *trm*
SPredicates :: *'c* ⇀ (*'a* + *'c*, *'b*, *'c*) *formula*
SContexts :: *'b* ⇀ (*'a*, *'b* + *unit*, *'c*) *formula*
SPrograms :: *'c* ⇀ (*'a*, *'b*, *'c*) *hp*
SODEs :: *'c* ⇀ (*'a*, *'c*) *ODE*

context *ids* **begin**

definition *NTUadmit* :: $('d \Rightarrow ('a, 'c)\ trm) \Rightarrow ('a + 'd, 'c)\ trm \Rightarrow ('c + 'c)\ set \Rightarrow bool$
where $NTUadmit\ \sigma\ \vartheta\ U \longleftrightarrow ((\bigcup\ i \in \{i.\ Inr\ i \in SIGT\ \vartheta\}.\ FVT\ (\sigma\ i)) \cap U) = \{\}$

inductive *TadmitFFO* :: $('d \Rightarrow ('a, 'c)\ trm) \Rightarrow ('a + 'd, 'c)\ trm \Rightarrow bool$
where
TadmitFFO-Diff:$TadmitFFO\ \sigma\ \vartheta \Longrightarrow NTUadmit\ \sigma\ \vartheta\ UNIV \Longrightarrow TadmitFFO\ \sigma\ (Differential\ \vartheta)$
| *TadmitFFO-Fun1*:$(\bigwedge i.\ TadmitFFO\ \sigma\ (args\ i)) \Longrightarrow TadmitFFO\ \sigma\ (Function\ (Inl\ f)\ args)$
| *TadmitFFO-Fun2*:$(\bigwedge i.\ TadmitFFO\ \sigma\ (args\ i)) \Longrightarrow dfree\ (\sigma\ f) \Longrightarrow TadmitFFO\ \sigma\ (Function\ (Inr\ f)\ args)$
| *TadmitFFO-Plus*:$TadmitFFO\ \sigma\ \vartheta 1 \Longrightarrow TadmitFFO\ \sigma\ \vartheta 2 \Longrightarrow TadmitFFO\ \sigma\ (Plus\ \vartheta 1\ \vartheta 2)$
| *TadmitFFO-Times*:$TadmitFFO\ \sigma\ \vartheta 1 \Longrightarrow TadmitFFO\ \sigma\ \vartheta 2 \Longrightarrow TadmitFFO\ \sigma\ (Times\ \vartheta 1\ \vartheta 2)$
| *TadmitFFO-Var*:$TadmitFFO\ \sigma\ (Var\ x)$
| *TadmitFFO-Const*:$TadmitFFO\ \sigma\ (Const\ r)$

inductive-simps
TadmitFFO-Diff-simps[*simp*]: $TadmitFFO\ \sigma\ (Differential\ \vartheta)$
and *TadmitFFO-Fun-simps*[*simp*]: $TadmitFFO\ \sigma\ (Function\ f\ args)$
and *TadmitFFO-Plus-simps*[*simp*]: $TadmitFFO\ \sigma\ (Plus\ t1\ t2)$
and *TadmitFFO-Times-simps*[*simp*]: $TadmitFFO\ \sigma\ (Times\ t1\ t2)$
and *TadmitFFO-Var-simps*[*simp*]: $TadmitFFO\ \sigma\ (Var\ x)$
and *TadmitFFO-Const-simps*[*simp*]: $TadmitFFO\ \sigma\ (Const\ r)$

primrec *TsubstFO*::$('a + 'b, 'c)\ trm \Rightarrow ('b \Rightarrow ('a, 'c)\ trm) \Rightarrow ('a, 'c)\ trm$
where
$TsubstFO\ (Var\ v)\ \sigma = Var\ v$
| $TsubstFO\ (DiffVar\ v)\ \sigma = DiffVar\ v$
| $TsubstFO\ (Const\ r)\ \sigma = Const\ r$
| $TsubstFO\ (Function\ f\ args)\ \sigma =$
  $(case\ f\ of$
    $Inl\ f' \Rightarrow Function\ f'\ (\lambda\ i.\ TsubstFO\ (args\ i)\ \sigma)$
  | $Inr\ f' \Rightarrow \sigma\ f')$
| $TsubstFO\ (Plus\ \vartheta 1\ \vartheta 2)\ \sigma = Plus\ (TsubstFO\ \vartheta 1\ \sigma)\ (TsubstFO\ \vartheta 2\ \sigma)$
| $TsubstFO\ (Times\ \vartheta 1\ \vartheta 2)\ \sigma = Times\ (TsubstFO\ \vartheta 1\ \sigma)\ (TsubstFO\ \vartheta 2\ \sigma)$
| $TsubstFO\ (Differential\ \vartheta)\ \sigma = Differential\ (TsubstFO\ \vartheta\ \sigma)$

inductive *TadmitFO* :: $('d \Rightarrow ('a, 'c)\ trm) \Rightarrow ('a + 'd, 'c)\ trm \Rightarrow bool$
where
TadmitFO-Diff:$TadmitFFO\ \sigma\ \vartheta \Longrightarrow NTUadmit\ \sigma\ \vartheta\ UNIV \Longrightarrow dfree\ (TsubstFO\ \vartheta\ \sigma) \Longrightarrow TadmitFO\ \sigma\ (Differential\ \vartheta)$
| *TadmitFO-Fun*:$(\bigwedge i.\ TadmitFO\ \sigma\ (args\ i)) \Longrightarrow TadmitFO\ \sigma\ (Function\ f\ args)$
| *TadmitFO-Plus*:$TadmitFO\ \sigma\ \vartheta 1 \Longrightarrow TadmitFO\ \sigma\ \vartheta 2 \Longrightarrow TadmitFO\ \sigma\ (Plus\ \vartheta 1\ \vartheta 2)$
| *TadmitFO-Times*:$TadmitFO\ \sigma\ \vartheta 1 \Longrightarrow TadmitFO\ \sigma\ \vartheta 2 \Longrightarrow TadmitFO\ \sigma\ (Times$

$\vartheta 1$ $\vartheta 2$)
| *TadmitFO-DiffVar*:*TadmitFO* σ (*DiffVar* x)
| *TadmitFO-Var*:*TadmitFO* σ (*Var* x)
| *TadmitFO-Const*:*TadmitFO* σ (*Const* r)

inductive-simps
TadmitFO-Plus-simps[*simp*]: *TadmitFO* σ (*Plus* a b)
and *TadmitFO-Times-simps*[*simp*]: *TadmitFO* σ (*Times* a b)
and *TadmitFO-Var-simps*[*simp*]: *TadmitFO* σ (*Var* x)
and *TadmitFO-DiffVar-simps*[*simp*]: *TadmitFO* σ (*DiffVar* x)
and *TadmitFO-Differential-simps*[*simp*]: *TadmitFO* σ (*Differential* ϑ)
and *TadmitFO-Const-simps*[*simp*]: *TadmitFO* σ (*Const* r)
and *TadmitFO-Fun-simps*[*simp*]: *TadmitFO* σ (*Function* i *args*)

primrec *Tsubst*::('a, 'c) *trm* $\Rightarrow$ ('a, 'b, 'c) *subst* $\Rightarrow$ ('a, 'c) *trm*
where
Tsubst (*Var* x) σ = *Var* x
| *Tsubst* (*DiffVar* x) σ = *DiffVar* x
| *Tsubst* (*Const* r) σ = *Const* r
| *Tsubst* (*Function* f *args*) σ = (*case SFunctions* σ f *of Some* f' $\Rightarrow$ *TsubstFO* f' | *None* $\Rightarrow$ *Function* f) (λ i. *Tsubst* (*args* i) σ)
| *Tsubst* (*Plus* $\vartheta 1$ $\vartheta 2$) σ = *Plus* (*Tsubst* $\vartheta 1$ σ) (*Tsubst* $\vartheta 2$ σ)
| *Tsubst* (*Times* $\vartheta 1$ $\vartheta 2$) σ = *Times* (*Tsubst* $\vartheta 1$ σ) (*Tsubst* $\vartheta 2$ σ)
| *Tsubst* (*Differential* ϑ) σ = *Differential* (*Tsubst* ϑ σ)

primrec *OsubstFO*::('a + 'b, 'c) *ODE* $\Rightarrow$ ('b $\Rightarrow$ ('a, 'c) *trm*) $\Rightarrow$ ('a, 'c) *ODE*
where
OsubstFO (*OVar* c) σ = *OVar* c
| *OsubstFO* (*OSing* x ϑ) σ = *OSing* x (*TsubstFO* ϑ σ)
| *OsubstFO* (*OProd ODE1 ODE2*) σ = *OProd* (*OsubstFO ODE1* σ) (*OsubstFO ODE2* σ)

primrec *Osubst*::('a, 'c) *ODE* $\Rightarrow$ ('a, 'b, 'c) *subst* $\Rightarrow$ ('a, 'c) *ODE*
where
Osubst (*OVar* c) σ = (*case SODEs* σ c *of Some* c' $\Rightarrow$ c' | *None* $\Rightarrow$ *OVar* c)
| *Osubst* (*OSing* x ϑ) σ = *OSing* x (*Tsubst* ϑ σ)
| *Osubst* (*OProd ODE1 ODE2*) σ = *OProd* (*Osubst ODE1* σ) (*Osubst ODE2* σ)

fun *PsubstFO*::('a + 'd, 'b, 'c) *hp* $\Rightarrow$ ('d $\Rightarrow$ ('a, 'c) *trm*) $\Rightarrow$ ('a, 'b, 'c) *hp*
and *FsubstFO*::('a + 'd, 'b, 'c) *formula* $\Rightarrow$ ('d $\Rightarrow$ ('a, 'c) *trm*) $\Rightarrow$ ('a, 'b, 'c) *formula*
where
PsubstFO (*Pvar* a) σ = *Pvar* a
| *PsubstFO* (*Assign* x ϑ) σ = *Assign* x (*TsubstFO* ϑ σ)
| *PsubstFO* (*DiffAssign* x ϑ) σ = *DiffAssign* x (*TsubstFO* ϑ σ)
| *PsubstFO* (*Test* φ) σ = *Test* (*FsubstFO* φ σ)
| *PsubstFO* (*EvolveODE ODE* φ) σ = *EvolveODE* (*OsubstFO ODE* σ) (*FsubstFO* φ σ)
| *PsubstFO* (*Choice* α β) σ = *Choice* (*PsubstFO* α σ) (*PsubstFO* β σ)
| *PsubstFO* (*Sequence* α β) σ = *Sequence* (*PsubstFO* α σ) (*PsubstFO* β σ)

| *PsubstFO* (*Loop* α) σ = *Loop* (*PsubstFO* α σ)

| *FsubstFO* (*Geq* $\vartheta 1$ $\vartheta 2$) σ = *Geq* (*TsubstFO* $\vartheta 1$ σ) (*TsubstFO* $\vartheta 2$ σ)
| *FsubstFO* (*Prop p args*) σ = *Prop p* (λi. *TsubstFO* (*args i*) σ)
| *FsubstFO* (*Not* φ) σ = *Not* (*FsubstFO* φ σ)
| *FsubstFO* (*And* φ ψ) σ = *And* (*FsubstFO* φ σ) (*FsubstFO* ψ σ)
| *FsubstFO* (*Exists x* φ) σ = *Exists x* (*FsubstFO* φ σ)
| *FsubstFO* (*Diamond* α φ) σ = *Diamond* (*PsubstFO* α σ) (*FsubstFO* φ σ)
| *FsubstFO* (*InContext C* φ) σ = *InContext C* (*FsubstFO* φ σ)

fun *PPsubst*::('*a*, '*b* + '*d*, '*c*) *hp* ⇒ ('*d* ⇒ ('*a*, '*b*, '*c*) *formula*) ⇒ ('*a*, '*b*, '*c*) *hp*
and *PFsubst*::('*a*, '*b* + '*d*, '*c*) *formula* ⇒ ('*d* ⇒ ('*a*, '*b*, '*c*) *formula*) ⇒ ('*a*, '*b*, '*c*) *formula*
where
  PPsubst (*Pvar a*) σ = *Pvar a*
| *PPsubst* (*Assign x* ϑ) σ = *Assign x* ϑ
| *PPsubst* (*DiffAssign x* ϑ) σ = *DiffAssign x* ϑ
| *PPsubst* (*Test* φ) σ = *Test* (*PFsubst* φ σ)
| *PPsubst* (*EvolveODE ODE* φ) σ = *EvolveODE ODE* (*PFsubst* φ σ)
| *PPsubst* (*Choice* α β) σ = *Choice* (*PPsubst* α σ) (*PPsubst* β σ)
| *PPsubst* (*Sequence* α β) σ = *Sequence* (*PPsubst* α σ) (*PPsubst* β σ)
| *PPsubst* (*Loop* α) σ = *Loop* (*PPsubst* α σ)

| *PFsubst* (*Geq* $\vartheta 1$ $\vartheta 2$) σ = (*Geq* $\vartheta 1$ $\vartheta 2$)
| *PFsubst* (*Prop p args*) σ = *Prop p args*
| *PFsubst* (*Not* φ) σ = *Not* (*PFsubst* φ σ)
| *PFsubst* (*And* φ ψ) σ = *And* (*PFsubst* φ σ) (*PFsubst* ψ σ)
| *PFsubst* (*Exists x* φ) σ = *Exists x* (*PFsubst* φ σ)
| *PFsubst* (*Diamond* α φ) σ = *Diamond* (*PPsubst* α σ) (*PFsubst* φ σ)
| *PFsubst* (*InContext C* φ) σ = (*case C of Inl C'* ⇒ *InContext C'* (*PFsubst* φ σ)
| *Inr p'* ⇒ σ *p'*)

fun *Psubst*::('*a*, '*b*, '*c*) *hp* ⇒ ('*a*, '*b*, '*c*) *subst* ⇒ ('*a*, '*b*, '*c*) *hp*
and *Fsubst*::('*a*, '*b*, '*c*) *formula* ⇒ ('*a*, '*b*, '*c*) *subst* ⇒ ('*a*, '*b*, '*c*) *formula*
where
  Psubst (*Pvar a*) σ = (*case SPrograms* σ *a of Some a'* ⇒ *a'* | *None* ⇒ *Pvar a*)
| *Psubst* (*Assign x* ϑ) σ = *Assign x* (*Tsubst* ϑ σ)
| *Psubst* (*DiffAssign x* ϑ) σ = *DiffAssign x* (*Tsubst* ϑ σ)
| *Psubst* (*Test* φ) σ = *Test* (*Fsubst* φ σ)
| *Psubst* (*EvolveODE ODE* φ) σ = *EvolveODE* (*Osubst ODE* σ) (*Fsubst* φ σ)
| *Psubst* (*Choice* α β) σ = *Choice* (*Psubst* α σ) (*Psubst* β σ)
| *Psubst* (*Sequence* α β) σ = *Sequence* (*Psubst* α σ) (*Psubst* β σ)
| *Psubst* (*Loop* α) σ = *Loop* (*Psubst* α σ)

| *Fsubst* (*Geq* $\vartheta 1$ $\vartheta 2$) σ = *Geq* (*Tsubst* $\vartheta 1$ σ) (*Tsubst* $\vartheta 2$ σ)
| *Fsubst* (*Prop p args*) σ = (*case SPredicates* σ *p of Some p'* ⇒ *FsubstFO p'* (λi. *Tsubst* (*args i*) σ) | *None* ⇒ *Prop p* (λi. *Tsubst* (*args i*) σ))
| *Fsubst* (*Not* φ) σ = *Not* (*Fsubst* φ σ)

| *Fsubst* (*And* φ ψ) σ = *And* (*Fsubst* φ σ) (*Fsubst* ψ σ)
| *Fsubst* (*Exists* x φ) σ = *Exists* x (*Fsubst* φ σ)
| *Fsubst* (*Diamond* α φ) σ = *Diamond* (*Psubst* α σ) (*Fsubst* φ σ)
| *Fsubst* (*InContext* C φ) σ = (*case SContexts* σ C *of Some* C' ⇒ *PFsubst* C' (λ -. (*Fsubst* φ σ)) | *None* ⇒ *InContext* C (*Fsubst* φ σ))

definition *FVA* :: ('*a* ⇒ ('*a*, '*c*) *trm*) ⇒ ('*c* + '*c*) *set*
where *FVA args* = ($\bigcup$ *i*. *FVT* (*args i*))

fun *SFV* :: ('*a*, '*b*, '*c*) *subst* ⇒ ('*a* + '*b* + '*c*) ⇒ ('*c* + '*c*) *set*
where *SFV* σ (*Inl i*) = (*case SFunctions* σ *i of Some* f' ⇒ *FVT* f' | *None* ⇒ {})
| *SFV* σ (*Inr* (*Inl i*)) = {}
| *SFV* σ (*Inr* (*Inr i*)) = (*case SPredicates* σ *i of Some* p' ⇒ *FVF* p' | *None* ⇒ {})

definition *FVS* :: ('*a*, '*b*, '*c*) *subst* ⇒ ('*c* + '*c*) *set*
where *FVS* σ = ($\bigcup$*i*. *SFV* σ *i*)

definition *SDom* :: ('*a*, '*b*, '*c*) *subst* ⇒ ('*a* + '*b* + '*c*) *set*
where *SDom* σ =
{*Inl x* | *x*. *x* ∈ *dom* (*SFunctions* σ)}
∪ {*Inr* (*Inl x*) | *x*. *x* ∈ *dom* (*SContexts* σ)}
∪ {*Inr* (*Inr x*) | *x*. *x* ∈ *dom* (*SPredicates* σ)}
∪ {*Inr* (*Inr x*) | *x*. *x* ∈ *dom* (*SPrograms* σ)}

definition *TUadmit* :: ('*a*, '*b*, '*c*) *subst* ⇒ ('*a*, '*c*) *trm* ⇒ ('*c* + '*c*) *set* ⇒ *bool*
where *TUadmit* σ ϑ *U* ⟷ (($\bigcup$ *i* ∈ *SIGT* ϑ. (*case SFunctions* σ *i of Some* f' ⇒ *FVT* f' | *None* ⇒ {})) ∩ *U*) = {}

inductive *Tadmit* :: ('*a*, '*b*, '*c*) *subst* ⇒ ('*a*, '*c*) *trm* ⇒ *bool*
where
Tadmit-Diff: *Tadmit* σ ϑ ⟹ *TUadmit* σ ϑ *UNIV* ⟹ *Tadmit* σ (*Differential* ϑ)
| *Tadmit-Fun1*: ($\bigwedge$*i*. *Tadmit* σ (*args i*)) ⟹ *SFunctions* σ f = *Some* f' ⟹ *TadmitFO* (λ *i*. *Tsubst* (*args i*) σ) f' ⟹ *Tadmit* σ (*Function f args*)
| *Tadmit-Fun2*: ($\bigwedge$*i*. *Tadmit* σ (*args i*)) ⟹ *SFunctions* σ f = *None* ⟹ *Tadmit* σ (*Function f args*)
| *Tadmit-Plus*: *Tadmit* σ ϑ*1* ⟹ *Tadmit* σ ϑ*2* ⟹ *Tadmit* σ (*Plus* ϑ*1* ϑ*2*)
| *Tadmit-Times*: *Tadmit* σ ϑ*1* ⟹ *Tadmit* σ ϑ*2* ⟹ *Tadmit* σ (*Times* ϑ*1* ϑ*2*)
| *Tadmit-DiffVar*: *Tadmit* σ (*DiffVar x*)
| *Tadmit-Var*: *Tadmit* σ (*Var x*)
| *Tadmit-Const*: *Tadmit* σ (*Const r*)

inductive-simps
Tadmit-Plus-simps[*simp*]: *Tadmit* σ (*Plus a b*)
and *Tadmit-Times-simps*[*simp*]: *Tadmit* σ (*Times a b*)
and *Tadmit-Var-simps*[*simp*]: *Tadmit* σ (*Var x*)
and *Tadmit-DiffVar-simps*[*simp*]: *Tadmit* σ (*DiffVar x*)
and *Tadmit-Differential-simps*[*simp*]: *Tadmit* σ (*Differential* ϑ)
and *Tadmit-Const-simps*[*simp*]: *Tadmit* σ (*Const r*)
and *Tadmit-Fun-simps*[*simp*]: *Tadmit* σ (*Function i args*)

inductive *TadmitF* :: (*'a*, *'b*, *'c*) *subst* ⇒ (*'a*, *'c*) *trm* ⇒ *bool*
where
TadmitF-Diff:*TadmitF σ ϑ* ⟹ *TUadmit σ ϑ UNIV* ⟹ *TadmitF σ* (*Differential ϑ*)
| *TadmitF-Fun1*:(⋀*i*. *TadmitF σ* (*args i*)) ⟹ *SFunctions σ f* = *Some f'* ⟹ (⋀*i*. *dfree* (*Tsubst* (*args i*) *σ*)) ⟹ *TadmitFFO* (λ *i*. *Tsubst* (*args i*) *σ*) *f'* ⟹ *TadmitF σ* (*Function f args*)
| *TadmitF-Fun2*:(⋀*i*. *TadmitF σ* (*args i*)) ⟹ *SFunctions σ f* = *None* ⟹ *TadmitF σ* (*Function f args*)
| *TadmitF-Plus*:*TadmitF σ ϑ1* ⟹ *TadmitF σ ϑ2* ⟹ *TadmitF σ* (*Plus ϑ1 ϑ2*)
| *TadmitF-Times*:*TadmitF σ ϑ1* ⟹ *TadmitF σ ϑ2* ⟹ *TadmitF σ* (*Times ϑ1 ϑ2*)
| *TadmitF-DiffVar*:*TadmitF σ* (*DiffVar x*)
| *TadmitF-Var*:*TadmitF σ* (*Var x*)
| *TadmitF-Const*:*TadmitF σ* (*Const r*)

inductive-simps
TadmitF-Plus-simps[*simp*]: *TadmitF σ* (*Plus a b*)
and *TadmitF-Times-simps*[*simp*]: *TadmitF σ* (*Times a b*)
and *TadmitF-Var-simps*[*simp*]: *TadmitF σ* (*Var x*)
and *TadmitF-DiffVar-simps*[*simp*]: *TadmitF σ* (*DiffVar x*)
and *TadmitF-Differential-simps*[*simp*]: *TadmitF σ* (*Differential ϑ*)
and *TadmitF-Const-simps*[*simp*]: *TadmitF σ* (*Const r*)
and *TadmitF-Fun-simps*[*simp*]: *TadmitF σ* (*Function i args*)

inductive *Oadmit*:: (*'a*, *'b*, *'c*) *subst* ⇒ (*'a*, *'c*) *ODE* ⇒ (*'c* + *'c*) *set* ⇒ *bool*
where
Oadmit-Var:*Oadmit σ* (*OVar c*) *U*
| *Oadmit-Sing*:*TUadmit σ ϑ U* ⟹ *TadmitF σ ϑ* ⟹ *Oadmit σ* (*OSing x ϑ*) *U*
| *Oadmit-Prod*:*Oadmit σ ODE1 U* ⟹ *Oadmit σ ODE2 U* ⟹ *ODE-dom* (*Osubst ODE1 σ*) ∩ *ODE-dom* (*Osubst ODE2 σ*) = {} ⟹ *Oadmit σ* (*OProd ODE1 ODE2*) *U*

inductive-simps
Oadmit-Var-simps[*simp*]: *Oadmit σ* (*OVar c*) *U*
and *Oadmit-Sing-simps*[*simp*]: *Oadmit σ* (*OSing x e*) *U*
and *Oadmit-Prod-simps*[*simp*]: *Oadmit σ* (*OProd ODE1 ODE2*) *U*

definition *PUadmit* :: (*'a*, *'b*, *'c*) *subst* ⇒ (*'a*, *'b*, *'c*) *hp* ⇒ (*'c* + *'c*) *set* ⇒ *bool*
where *PUadmit σ ϑ U* ⟷ ((⋃ *i* ∈ (*SDom σ* ∩ *SIGP ϑ*). *SFV σ i*) ∩ *U*) = {}

definition *FUadmit* :: (*'a*, *'b*, *'c*) *subst* ⇒ (*'a*, *'b*, *'c*) *formula* ⇒ (*'c* + *'c*) *set* ⇒ *bool*
where *FUadmit σ ϑ U* ⟷ ((⋃ *i* ∈ (*SDom σ* ∩ *SIGF ϑ*). *SFV σ i*) ∩ *U*) = {}

definition *OUadmitFO* :: (*'d* ⇒ (*'a*, *'c*) *trm*) ⇒ (*'a* + *'d*, *'c*) *ODE* ⇒ (*'c* + *'c*) *set* ⇒ *bool*
where *OUadmitFO σ ϑ U* ⟷ ((⋃ *i* ∈ {*i*. *Inl* (*Inr i*) ∈ *SIGO ϑ*}. *FVT* (*σ i*))

$\cap\ U) = \{\}$

inductive *OadmitFO* :: $('d \Rightarrow ('a,\ 'c)\ trm) \Rightarrow ('a + 'd,\ 'c)\ ODE \Rightarrow ('c + 'c)\ set \Rightarrow bool$
where
OadmitFO-OVar:$OUadmitFO\ \sigma\ (OVar\ c)\ U \Longrightarrow OadmitFO\ \sigma\ (OVar\ c)\ U$
| *OadmitFO-OSing*:$OUadmitFO\ \sigma\ (OSing\ x\ \vartheta)\ U \Longrightarrow TadmitFFO\ \sigma\ \vartheta \Longrightarrow OadmitFO\ \sigma\ (OSing\ x\ \vartheta)\ U$
| *OadmitFO-OProd*:$OadmitFO\ \sigma\ ODE1\ U \Longrightarrow OadmitFO\ \sigma\ ODE2\ U \Longrightarrow OadmitFO\ \sigma\ (OProd\ ODE1\ ODE2)\ U$

inductive-simps
OadmitFO-OVar-simps[*simp*]: $OadmitFO\ \sigma\ (OVar\ a)\ U$
and *OadmitFO-OProd-simps*[*simp*]: $OadmitFO\ \sigma\ (OProd\ ODE1\ ODE2)\ U$
and *OadmitFO-OSing-simps*[*simp*]: $OadmitFO\ \sigma\ (OSing\ x\ e)\ U$

definition *FUadmitFO* :: $('d \Rightarrow ('a,\ 'c)\ trm) \Rightarrow ('a + 'd,\ 'b,\ 'c)\ formula \Rightarrow ('c + 'c)\ set \Rightarrow bool$
where $FUadmitFO\ \sigma\ \vartheta\ U \longleftrightarrow ((\bigcup\ i \in \{i.\ Inl\ (Inr\ i) \in SIGF\ \vartheta\}.\ FVT\ (\sigma\ i)) \cap U) = \{\}$

definition *PUadmitFO* :: $('d \Rightarrow ('a,\ 'c)\ trm) \Rightarrow ('a + 'd,\ 'b,\ 'c)\ hp \Rightarrow ('c + 'c)\ set \Rightarrow bool$
where $PUadmitFO\ \sigma\ \vartheta\ U \longleftrightarrow ((\bigcup\ i \in \{i.\ Inl\ (Inr\ i) \in SIGP\ \vartheta\}.\ FVT\ (\sigma\ i)) \cap U) = \{\}$

inductive *NPadmit* :: $('d \Rightarrow ('a,\ 'c)\ trm) \Rightarrow ('a + 'd,\ 'b,\ 'c)\ hp \Rightarrow bool$
and *NFadmit* :: $('d \Rightarrow ('a,\ 'c)\ trm) \Rightarrow ('a + 'd,\ 'b,\ 'c)\ formula \Rightarrow bool$
where
NPadmit-Pvar:$NPadmit\ \sigma\ (Pvar\ a)$
| *NPadmit-Sequence*:$NPadmit\ \sigma\ a \Longrightarrow NPadmit\ \sigma\ b \Longrightarrow PUadmitFO\ \sigma\ b\ (BVP\ (PsubstFO\ a\ \sigma)) \Longrightarrow hpsafe\ (PsubstFO\ a\ \sigma) \Longrightarrow NPadmit\ \sigma\ (Sequence\ a\ b)$
| *NPadmit-Loop*:$NPadmit\ \sigma\ a \Longrightarrow PUadmitFO\ \sigma\ a\ (BVP\ (PsubstFO\ a\ \sigma)) \Longrightarrow hpsafe\ (PsubstFO\ a\ \sigma) \Longrightarrow NPadmit\ \sigma\ (Loop\ a)$
| *NPadmit-ODE*:$OadmitFO\ \sigma\ ODE\ (BVO\ ODE) \Longrightarrow NFadmit\ \sigma\ \varphi \Longrightarrow FUadmitFO\ \sigma\ \varphi\ (BVO\ ODE) \Longrightarrow fsafe\ (FsubstFO\ \varphi\ \sigma) \Longrightarrow osafe\ (OsubstFO\ ODE\ \sigma) \Longrightarrow NPadmit\ \sigma\ (EvolveODE\ ODE\ \varphi)$
| *NPadmit-Choice*:$NPadmit\ \sigma\ a \Longrightarrow NPadmit\ \sigma\ b \Longrightarrow NPadmit\ \sigma\ (Choice\ a\ b)$

| *NPadmit-Assign*:$TadmitFO\ \sigma\ \vartheta \Longrightarrow NPadmit\ \sigma\ (Assign\ x\ \vartheta)$
| *NPadmit-DiffAssign*:$TadmitFO\ \sigma\ \vartheta \Longrightarrow NPadmit\ \sigma\ (DiffAssign\ x\ \vartheta)$
| *NPadmit-Test*:$NFadmit\ \sigma\ \varphi \Longrightarrow NPadmit\ \sigma\ (Test\ \varphi)$

| *NFadmit-Geq*:$TadmitFO\ \sigma\ \vartheta 1 \Longrightarrow TadmitFO\ \sigma\ \vartheta 2 \Longrightarrow NFadmit\ \sigma\ (Geq\ \vartheta 1\ \vartheta 2)$
| *NFadmit-Prop*:$(\bigwedge i.\ TadmitFO\ \sigma\ (args\ i)) \Longrightarrow NFadmit\ \sigma\ (Prop\ f\ args)$
| *NFadmit-Not*:$NFadmit\ \sigma\ \varphi \Longrightarrow NFadmit\ \sigma\ (Not\ \varphi)$
| *NFadmit-And*:$NFadmit\ \sigma\ \varphi \Longrightarrow NFadmit\ \sigma\ \psi \Longrightarrow NFadmit\ \sigma\ (And\ \varphi\ \psi)$
| *NFadmit-Exists*:$NFadmit\ \sigma\ \varphi \Longrightarrow FUadmitFO\ \sigma\ \varphi\ \{Inl\ x\} \Longrightarrow NFadmit\ \sigma\ (Exists\ x\ \varphi)$

| *NFadmit-Diamond*:*NFadmit* σ φ $\Longrightarrow$ *NPadmit* σ a $\Longrightarrow$ *FUadmitFO* σ φ (*BVP* (*PsubstFO* a σ)) $\Longrightarrow$ *hpsafe* (*PsubstFO* a σ) $\Longrightarrow$ *NFadmit* σ (*Diamond* a φ)
| *NFadmit-Context*:*NFadmit* σ φ $\Longrightarrow$ *FUadmitFO* σ φ *UNIV* $\Longrightarrow$ *NFadmit* σ (*InContext* C φ)

inductive-simps
NPadmit-Pvar-simps[*simp*]: *NPadmit* σ (*Pvar* a)
and *NPadmit-Sequence-simps*[*simp*]: *NPadmit* σ (a ;; b)
and *NPadmit-Loop-simps*[*simp*]: *NPadmit* σ (a**)
and *NPadmit-ODE-simps*[*simp*]: *NPadmit* σ (*EvolveODE ODE p*)
and *NPadmit-Choice-simps*[*simp*]: *NPadmit* σ (a ∪∪ b)
and *NPadmit-Assign-simps*[*simp*]: *NPadmit* σ (*Assign x e*)
and *NPadmit-DiffAssign-simps*[*simp*]: *NPadmit* σ (*DiffAssign x e*)
and *NPadmit-Test-simps*[*simp*]: *NPadmit* σ (*? p*)

and *NFadmit-Geq-simps*[*simp*]: *NFadmit* σ (*Geq t1 t2*)
and *NFadmit-Prop-simps*[*simp*]: *NFadmit* σ (*Prop p args*)
and *NFadmit-Not-simps*[*simp*]: *NFadmit* σ (*Not p*)
and *NFadmit-And-simps*[*simp*]: *NFadmit* σ (*And p q*)
and *NFadmit-Exists-simps*[*simp*]: *NFadmit* σ (*Exists x p*)
and *NFadmit-Diamond-simps*[*simp*]: *NFadmit* σ (*Diamond a p*)
and *NFadmit-Context-simps*[*simp*]: *NFadmit* σ (*InContext C p*)

definition *PFUadmit* :: ('*d* ⇒ ('*a*, '*b*, '*c*) *formula*) ⇒ ('*a*, '*b* + '*d*, '*c*) *formula* ⇒ ('*c* + '*c*) *set* ⇒ *bool*
where *PFUadmit* σ ϑ U ⟷ *True*

definition *PPUadmit* :: ('*d* ⇒ ('*a*, '*b*, '*c*) *formula*) ⇒ ('*a*, '*b* + '*d*, '*c*) *hp* ⇒ ('*c* + '*c*) *set* ⇒ *bool*
where *PPUadmit* σ ϑ U ⟷ ((⋃ i. *FVF* (σ i)) ∩ U) = {}

inductive *PPadmit*:: ('*d* ⇒ ('*a*, '*b*, '*c*) *formula*) ⇒ ('*a*, '*b* + '*d*, '*c*) *hp* ⇒ *bool*
and *PFadmit*:: ('*d* ⇒ ('*a*, '*b*, '*c*) *formula*) ⇒ ('*a*, '*b* + '*d*, '*c*) *formula* ⇒ *bool*
where
PPadmit-Pvar:*PPadmit* σ (*Pvar* a)
| *PPadmit-Sequence*:*PPadmit* σ a $\Longrightarrow$ *PPadmit* σ b $\Longrightarrow$ *PPUadmit* σ b (*BVP* (*PPsubst* a σ))$\Longrightarrow$ *hpsafe* (*PPsubst* a σ) $\Longrightarrow$ *PPadmit* σ (*Sequence* a b)
| *PPadmit-Loop*:*PPadmit* σ a $\Longrightarrow$ *PPUadmit* σ a (*BVP* (*PPsubst* a σ)) $\Longrightarrow$ *hpsafe* (*PPsubst* a σ) $\Longrightarrow$ *PPadmit* σ (*Loop* a)
| *PPadmit-ODE*:*PFadmit* σ φ $\Longrightarrow$ *PFUadmit* σ φ (*BVO ODE*) $\Longrightarrow$ *PPadmit* σ (*EvolveODE ODE* φ)
| *PPadmit-Choice*:*PPadmit* σ a $\Longrightarrow$ *PPadmit* σ b $\Longrightarrow$ *PPadmit* σ (*Choice* a b)

| *PPadmit-Assign*:*PPadmit* σ (*Assign* x ϑ)
| *PPadmit-DiffAssign*:*PPadmit* σ (*DiffAssign* x ϑ)
| *PPadmit-Test*:*PFadmit* σ φ $\Longrightarrow$ *PPadmit* σ (*Test* φ)

| *PFadmit-Geq*:*PFadmit* σ (*Geq* $\vartheta 1$ $\vartheta 2$)
| *PFadmit-Prop*:*PFadmit* σ (*Prop* f *args*)

| *PFadmit-Not*:*PFadmit* σ φ $\Longrightarrow$ *PFadmit* σ (*Not* φ)
| *PFadmit-And*:*PFadmit* σ φ $\Longrightarrow$ *PFadmit* σ ψ $\Longrightarrow$ *PFadmit* σ (*And* φ ψ)
| *PFadmit-Exists*:*PFadmit* σ φ $\Longrightarrow$ *PFUadmit* σ φ {*Inl x*} $\Longrightarrow$ *PFadmit* σ (*Exists* x φ)
| *PFadmit-Diamond*:*PFadmit* σ φ $\Longrightarrow$ *PPadmit* σ a $\Longrightarrow$ *PFUadmit* σ φ (*BVP* (*PPsubst a* σ)) $\Longrightarrow$ *PFadmit* σ (*Diamond a* φ)
| *PFadmit-Context*:*PFadmit* σ φ $\Longrightarrow$ *PFUadmit* σ φ *UNIV* $\Longrightarrow$ *PFadmit* σ (*InContext* C φ)

inductive-simps
PPadmit-Pvar-simps[*simp*]: *PPadmit* σ (*Pvar a*)
and *PPadmit-Sequence-simps*[*simp*]: *PPadmit* σ (a ;; b)
and *PPadmit-Loop-simps*[*simp*]: *PPadmit* σ (a**)
and *PPadmit-ODE-simps*[*simp*]: *PPadmit* σ (*EvolveODE ODE p*)
and *PPadmit-Choice-simps*[*simp*]: *PPadmit* σ (a ∪∪ b)
and *PPadmit-Assign-simps*[*simp*]: *PPadmit* σ (*Assign x e*)
and *PPadmit-DiffAssign-simps*[*simp*]: *PPadmit* σ (*DiffAssign x e*)
and *PPadmit-Test-simps*[*simp*]: *PPadmit* σ (? p)

and *PFadmit-Geq-simps*[*simp*]: *PFadmit* σ (*Geq t1 t2*)
and *PFadmit-Prop-simps*[*simp*]: *PFadmit* σ (*Prop p args*)
and *PFadmit-Not-simps*[*simp*]: *PFadmit* σ (*Not p*)
and *PFadmit-And-simps*[*simp*]: *PFadmit* σ (*And p q*)
and *PFadmit-Exists-simps*[*simp*]: *PFadmit* σ (*Exists x p*)
and *PFadmit-Diamond-simps*[*simp*]: *PFadmit* σ (*Diamond a p*)
and *PFadmit-Context-simps*[*simp*]: *PFadmit* σ (*InContext C p*)

inductive *Padmit*:: ('*a*, '*b*, '*c*) *subst* ⇒ ('*a*, '*b*, '*c*) *hp* ⇒ *bool*
and *Fadmit*:: ('*a*, '*b*, '*c*) *subst* ⇒ ('*a*, '*b*, '*c*) *formula* ⇒ *bool*
where
Padmit-Pvar:*Padmit* σ (*Pvar a*)
| *Padmit-Sequence*:*Padmit* σ a $\Longrightarrow$ *Padmit* σ b (*BVP* (*Psubst a* σ))$\Longrightarrow$ *hpsafe* (*Psubst a* σ) $\Longrightarrow$ *Padmit* σ (*Sequence a b*)
| *Padmit-Loop*:*Padmit* σ a $\Longrightarrow$ *PUadmit* σ a (*BVP* (*Psubst a* σ)) $\Longrightarrow$ *hpsafe* (*Psubst a* σ) $\Longrightarrow$ *Padmit* σ (*Loop a*)
| *Padmit-ODE*:*Oadmit* σ *ODE* (*BVO ODE*) $\Longrightarrow$ *Fadmit* σ φ $\Longrightarrow$ *FUadmit* σ φ (*BVO ODE*) $\Longrightarrow$ *Padmit* σ (*EvolveODE ODE* φ)
| *Padmit-Choice*:*Padmit* σ a $\Longrightarrow$ *Padmit* σ b $\Longrightarrow$ *Padmit* σ (*Choice a b*)
| *Padmit-Assign*:*Tadmit* σ ϑ $\Longrightarrow$ *Padmit* σ (*Assign x* ϑ)
| *Padmit-DiffAssign*:*Tadmit* σ ϑ $\Longrightarrow$ *Padmit* σ (*DiffAssign x* ϑ)
| *Padmit-Test*:*Fadmit* σ φ $\Longrightarrow$ *Padmit* σ (*Test* φ)

| *Fadmit-Geq*:*Tadmit* σ $\vartheta 1$ $\Longrightarrow$ *Tadmit* σ $\vartheta 2$ $\Longrightarrow$ *Fadmit* σ (*Geq* $\vartheta 1$ $\vartheta 2$)
| *Fadmit-Prop1*:($\bigwedge i$. *Tadmit* σ (*args i*)) $\Longrightarrow$ *SPredicates* σ p = *Some* p' $\Longrightarrow$ *NFadmit* (λ i. *Tsubst* (*args i*) σ) p' $\Longrightarrow$ ($\bigwedge i$. *dsafe* (*Tsubst* (*args i*) σ))$\Longrightarrow$ *Fadmit* σ (*Prop p args*)
| *Fadmit-Prop2*:($\bigwedge i$. *Tadmit* σ (*args i*)) $\Longrightarrow$ *SPredicates* σ p = *None* $\Longrightarrow$ *Fadmit* σ (*Prop p args*)
| *Fadmit-Not*:*Fadmit* σ φ $\Longrightarrow$ *Fadmit* σ (*Not* φ)

| *Fadmit-And*:*Fadmit* σ $\varphi \Longrightarrow$ *Fadmit* σ $\psi \Longrightarrow$ *Fadmit* σ (*And* φ ψ)
| *Fadmit-Exists*:*Fadmit* σ $\varphi \Longrightarrow$ *FUadmit* σ φ {*Inl* x} $\Longrightarrow$ *Fadmit* σ (*Exists* x φ)
| *Fadmit-Diamond*:*Fadmit* σ $\varphi \Longrightarrow$ *Padmit* σ $a \Longrightarrow$ *FUadmit* σ φ (*BVP* (*Psubst* a σ)) $\Longrightarrow$ *hpsafe* (*Psubst* a σ) $\Longrightarrow$ *Fadmit* σ (*Diamond* a φ)
| *Fadmit-Context1*:*Fadmit* σ $\varphi \Longrightarrow$ *FUadmit* σ φ *UNIV* $\Longrightarrow$ *SContexts* σ C = *Some* $C' \Longrightarrow$ *PFadmit* (λ -. *Fsubst* φ σ) $C' \Longrightarrow$ *fsafe*(*Fsubst* φ σ) $\Longrightarrow$ *Fadmit* σ (*InContext* C φ)
| *Fadmit-Context2*:*Fadmit* σ $\varphi \Longrightarrow$ *FUadmit* σ φ *UNIV* $\Longrightarrow$ *SContexts* σ C = *None* $\Longrightarrow$ *Fadmit* σ (*InContext* C φ)

inductive-simps
Padmit-Pvar-simps[*simp*]: *Padmit* σ (*Pvar* a)
and *Padmit-Sequence-simps*[*simp*]: *Padmit* σ (a ;; b)
and *Padmit-Loop-simps*[*simp*]: *Padmit* σ (a**)
and *Padmit-ODE-simps*[*simp*]: *Padmit* σ (*EvolveODE ODE* p)
and *Padmit-Choice-simps*[*simp*]: *Padmit* σ (a ∪∪ b)
and *Padmit-Assign-simps*[*simp*]: *Padmit* σ (*Assign* x e)
and *Padmit-DiffAssign-simps*[*simp*]: *Padmit* σ (*DiffAssign* x e)
and *Padmit-Test-simps*[*simp*]: *Padmit* σ (*?* p)

and *Fadmit-Geq-simps*[*simp*]: *Fadmit* σ (*Geq t1 t2*)
and *Fadmit-Prop-simps*[*simp*]: *Fadmit* σ (*Prop* p *args*)
and *Fadmit-Not-simps*[*simp*]: *Fadmit* σ (*Not* p)
and *Fadmit-And-simps*[*simp*]: *Fadmit* σ (*And* p q)
and *Fadmit-Exists-simps*[*simp*]: *Fadmit* σ (*Exists* x p)
and *Fadmit-Diamond-simps*[*simp*]: *Fadmit* σ (*Diamond* a p)
and *Fadmit-Context-simps*[*simp*]: *Fadmit* σ (*InContext* C p)

fun *extendf* :: ('*sf*, '*sc*, '*sz*) *interp* ⇒ '*sz Rvec* ⇒ ('*sf* + '*sz*, '*sc*, '*sz*) *interp*
where *extendf I R* =
(|*Functions* = (λ*f*. *case f of Inl f'* ⇒ *Functions I f'* | *Inr f'* ⇒ (λ-. *R* $ *f'*)),
Predicates = *Predicates I*,
Contexts = *Contexts I*,
Programs = *Programs I*,
ODEs = *ODEs I*,
ODEBV = *ODEBV I*
|)

fun *extendc* :: ('*sf*, '*sc*, '*sz*) *interp* ⇒ '*sz state set* ⇒ ('*sf*, '*sc* + *unit*, '*sz*) *interp*
where *extendc I R* =
(|*Functions* = *Functions I*,
Predicates = *Predicates I*,
Contexts = (λ*C*. *case C of Inl C'* ⇒ *Contexts I C'* | *Inr* () ⇒ (λ-. *R*)),
Programs = *Programs I*,
ODEs = *ODEs I*,
ODEBV = *ODEBV I*|)

definition *adjoint* :: ('*sf*, '*sc*, '*sz*) *interp* ⇒ ('*sf*, '*sc*, '*sz*) *subst* ⇒ '*sz state* ⇒ ('*sf*, '*sc*, '*sz*) *interp*

where *adjoint I σ ν* =
(|*Functions* = (λ*f*. *case SFunctions σ f of Some f′* ⇒ (λ*R*. *dterm-sem* (*extendf I R*) *f′ ν*) | *None* ⇒ *Functions I f*),
Predicates = (λ*p*. *case SPredicates σ p of Some p′* ⇒ (λ*R*. *ν* ∈ *fml-sem* (*extendf I R*) *p′*) | *None* ⇒ *Predicates I p*),
Contexts = (λ*c*. *case SContexts σ c of Some c′* ⇒ (λ*R*. *fml-sem* (*extendc I R*) *c′*) | *None* ⇒ *Contexts I c*),
Programs = (λ*a*. *case SPrograms σ a of Some a′* ⇒ *prog-sem I a′* | *None* ⇒ *Programs I a*),
ODEs = (λ*ode*. *case SODEs σ ode of Some ode′* ⇒ *ODE-sem I ode′* | *None* ⇒ *ODEs I ode*),
ODEBV = (λ*ode*. *case SODEs σ ode of Some ode′* ⇒ *ODE-vars I ode′* | *None* ⇒ *ODEBV I ode*)
|)

lemma *dsem-to-ssem*:*dfree ϑ* ⟹ *dterm-sem I ϑ ν* = *sterm-sem I ϑ* (*fst ν*)
by (*induct rule*: *dfree.induct*) (*auto*)

definition *adjointFO*::('*sf*, '*sc*, '*sz*) *interp* ⇒ ('*d*::*finite* ⇒ ('*sf*, '*sz*) *trm*) ⇒ '*sz state* ⇒ ('*sf* + '*d*, '*sc*, '*sz*) *interp*
where *adjointFO I σ ν* =
(|*Functions* = (λ*f*. *case f of Inl f′* ⇒ *Functions I f′* | *Inr f′* ⇒ (λ-. *dterm-sem I* (*σ f′*) *ν*)),
Predicates = *Predicates I*,
Contexts = *Contexts I*,
Programs = *Programs I*,
ODEs = *ODEs I*,
ODEBV = *ODEBV I*
|)

lemma *adjoint-free*:
assumes *sfree*:(⋀*i f′*. *SFunctions σ i* = *Some f′* ⟹ *dfree f′*)
shows *adjoint I σ ν* =
(|*Functions* = (λ*f*. *case SFunctions σ f of Some f′* ⇒ (λ*R*. *sterm-sem* (*extendf I R*) *f′* (*fst ν*)) | *None* ⇒ *Functions I f*),
Predicates = (λ*p*. *case SPredicates σ p of Some p′* ⇒ (λ*R*. *ν* ∈ *fml-sem* (*extendf I R*) *p′*) | *None* ⇒ *Predicates I p*),
Contexts = (λ*c*. *case SContexts σ c of Some c′* ⇒ (λ*R*. *fml-sem* (*extendc I R*) *c′*) | *None* ⇒ *Contexts I c*),
Programs = (λ*a*. *case SPrograms σ a of Some a′* ⇒ *prog-sem I a′* | *None* ⇒ *Programs I a*),
ODEs = (λ*ode*. *case SODEs σ ode of Some ode′* ⇒ *ODE-sem I ode′* | *None* ⇒ *ODEs I ode*),
ODEBV = (λ*ode*. *case SODEs σ ode of Some ode′* ⇒ *ODE-vars I ode′* | *None* ⇒ *ODEBV I ode*)|)
using *dsem-to-ssem*[*OF sfree*]
by (*cases ν*) (*auto simp add*: *adjoint-def fun-eq-iff split*: *option.split*)

lemma *adjointFO-free*:(⋀*i*. *dfree* (*σ i*)) ⟹ (*adjointFO I σ ν* =

(|*Functions* = (λ*f*. *case f of Inl f′* ⇒ *Functions I f′* | *Inr f′* ⇒ (λ-. *sterm-sem I* (σ *f′*) (*fst* ν))),
Predicates = *Predicates I*,
Contexts = *Contexts I*,
Programs = *Programs I*,
ODEs = *ODEs I*,
ODEBV = *ODEBV I*|)
by (*auto simp add*: *dsem-to-ssem adjointFO-def*)

definition *PFadjoint*::('*sf*, '*sc*, '*sz*) *interp* ⇒ ('*d*::*finite* ⇒ ('*sf*, '*sc*, '*sz*) *formula*) ⇒ ('*sf*, '*sc* + '*d*, '*sz*) *interp*
where *PFadjoint I* σ =
(|*Functions* = *Functions I*,
Predicates = *Predicates I*,
Contexts = (λ*f*. *case f of Inl f′* ⇒ *Contexts I f′* | *Inr f′* ⇒ (λ-. *fml-sem I* (σ *f′*))),
Programs = *Programs I*,
ODEs = *ODEs I*,
ODEBV = *ODEBV I*|)

fun *Ssubst*::('*sf*, '*sc*, '*sz*) *sequent* ⇒ ('*sf*,'*sc*,'*sz*) *subst* ⇒ ('*sf*,'*sc*,'*sz*) *sequent*
where *Ssubst* (Γ,Δ) σ = (*map* (λ φ. *Fsubst* φ σ) Γ, *map* (λ φ. *Fsubst* φ σ) Δ)

fun *Rsubst*::('*sf*, '*sc*, '*sz*) *rule* ⇒ ('*sf*,'*sc*,'*sz*) *subst* ⇒ ('*sf*,'*sc*,'*sz*) *rule*
where *Rsubst* (*SG*,*C*) σ = (*map* (λ φ. *Ssubst* φ σ) *SG*, *Ssubst C* σ)

definition *Sadmit*::('*sf*,'*sc*,'*sz*) *subst* ⇒ ('*sf*,'*sc*,'*sz*) *sequent* ⇒ *bool*
where *Sadmit* σ *S* ⟷ ((∀ *i*. *i* ≥ *0* ⟶ *i* < *length* (*fst S*) ⟶ *Fadmit* σ (*nth* (*fst S*) *i*))
∧(∀ *i*. *i* ≥ *0* ⟶ *i* < *length* (*snd S*) ⟶ *Fadmit* σ (*nth* (*snd S*) *i*)))

definition *Radmit*::('*sf*,'*sc*,'*sz*) *subst* ⇒ ('*sf*,'*sc*,'*sz*) *rule* ⇒ *bool*
where *Radmit* σ *R* ⟷ (((∀ *i*. *i* ≥ *0* ⟶ *i* < *length* (*fst R*) ⟶ *Sadmit* σ (*nth* (*fst R*) *i*))
∧ *Sadmit* σ (*snd R*)))

end end
theory *USubst-Lemma*
imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
Frechet-Correctness
Static-Semantics
Coincidence
Bound-Effect

USubst
begin context *ids* **begin**

11 Soundness proof for uniform substitution rule

lemma *interp-eq*:
f = f′ ⟹ p = p′ ⟹ c = c′ ⟹ PP = PP′ ⟹ ode = ode′ ⟹ odebv = odebv′
⟹
(|Functions = f, Predicates = p, Contexts = c, Programs = PP, ODEs = ode, ODEBV = odebv|) =
(|Functions = f′, Predicates = p′, Contexts = c′, Programs = PP′, ODEs = ode′, ODEBV = odebv′|)
by *auto*

11.1 Lemmas about well-formedness of (adjoint) interpretations.

When adding a function to an interpretation with `extendf`, we need to show it's C1 continuous. We do this by explicitly constructing the derivative `extendf_deriv` and showing it's continuous.

primrec *extendf-deriv* :: *('sf,'sc,'sz) interp ⇒ 'sf ⇒ ('sf + 'sz,'sz) trm ⇒ 'sz state ⇒ 'sz Rvec ⇒ ('sz Rvec ⇒ real)*
where
extendf-deriv I - (Var i) ν x = (λ-. 0)
| extendf-deriv I - (Const r) ν x = (λ-. 0)
| extendf-deriv I g (Function f args) ν x =
(case f of
Inl ff ⇒ (THE f′. ∀ y. (Functions I ff has-derivative f′ y) (at y))
(χ i. dterm-sem
(|Functions = case-sum (Functions I) (λf′ -. x $ f′), Predicates = Predicates I, Contexts = Contexts I, Programs = Programs I,
ODEs = ODEs I, ODEBV = ODEBV I|)
(args i) ν) ∘
(λν′. χ ia. extendf-deriv I g (args ia) ν x ν′)
| Inr ff ⇒ (λ ν′. ν′ $ ff))
| extendf-deriv I g (Plus t1 t2) ν x = (λν′. (extendf-deriv I g t1 ν x ν′) + (extendf-deriv I g t2 ν x ν′))
| extendf-deriv I g (Times t1 t2) ν x =
(λν′. ((dterm-sem (extendf I x) t1 ν ∗ (extendf-deriv I g t2 ν x ν′)))
+ (extendf-deriv I g t1 ν x ν′) ∗ (dterm-sem (extendf I x) t2 ν))
| extendf-deriv I g ($′ -) ν = undefined
| extendf-deriv I g (Differential -) ν = undefined

lemma *extendf-dterm-sem-continuous*:
fixes *f′*::*('sf + 'sz,'sz) trm* **and** *I*::*('sf,'sc,'sz) interp*
assumes *free*:*dfree f′*
assumes *good-interp*:*is-interp I*
shows *continuous-on UNIV (λx. dterm-sem (extendf I x) f′ ν)*

proof(*induction rule*: *dfree.induct*[*OF free*])
case (*3 args f*)
then show *?case*
apply(*cases f*)
apply (*auto simp add*: *continuous-intros*)
subgoal for *a*
apply(*rule continuous-on-compose2*[*of UNIV Functions I a UNIV* (λ *x*. (χ *i*. *dterm-sem*
(⦇*Functions* = *case-sum* (*Functions I*) (λ*f'* -. *x* $ *f'*), *Predicates* = *Predicates I*, *Contexts* = *Contexts I*,
Programs = *Programs I*, *ODEs* = *ODEs I*, *ODEBV* = *ODEBV I*⦈)
(*args i*) ν))])
subgoal
using *is-interpD*[*OF good-interp*]
using *has-derivative-continuous-on*[*of UNIV* (*Functions I a*) (*THE f'*. ∀ *x*. (*Functions I a has-derivative f' x*) (*at x*))]
by *auto*
apply(*rule continuous-on-vec-lambda*) **by** *auto*
done
qed (*auto simp add*: *continuous-intros*)

lemma *extendf-deriv-bounded*:
fixes *f'*::('*sf* + '*sz*,'*sz*) *trm* **and** *I*::('*sf*,'*sc*,'*sz*) *interp*
assumes *free*:*dfree f'*
assumes *good-interp*:*is-interp I*
shows *bounded-linear* (*extendf-deriv I i f' ν x*)
proof(*induction rule*: *dfree.induct*[*OF free*])
case (*1 i*)
then show *?case* **by** *auto*
next
case (*2 r*)
then show *?case* **by** *auto*
next
case (*3 args f*)
then show *?case* **apply** *auto*
apply(*cases f*)
apply *auto*
subgoal for *a*
apply(*rule bounded-linear-compose*[*of* (*THE f'*. ∀ *y*. (*Functions I a has-derivative f' y*) (*at y*))
(χ *i*. *dterm-sem*
(⦇*Functions* = *case-sum* (*Functions I*) (λ*f'* -. *x* $ *f'*), *Predicates* = *Predicates I*, *Contexts* = *Contexts I*, *Programs* = *Programs I*,
ODEs = *ODEs I*, *ODEBV* = *ODEBV I*⦈)
(*args i*) ν)])
subgoal using *good-interp* **unfolding** *is-interp-def* **using** *has-derivative-bounded-linear* **by** *fastforce*
apply(*rule bounded-linear-vec*)

```
    by auto
  done
next
 case (4 ϑ1 ϑ2)
 then show ?case apply auto
  using bounded-linear-add by blast
next
 case (5 ϑ1 ϑ2)
 then show ?case apply auto
  apply(rule bounded-linear-add)
   apply(rule bounded-linear-const-mult)
   subgoal by auto
  apply(rule bounded-linear-mult-const)
  subgoal by auto
  done
qed

lemma extendf-deriv-continuous:
 fixes f'::('sf + 'sz,'sz) trm and I::('sf,'sc,'sz) interp
 assumes free:dfree f'
 assumes good-interp:is-interp I
 shows continuous-on UNIV (λx. Blinfun (extendf-deriv I i f' ν x))
proof (induction rule: dfree.induct[OF free])
 case (3 args f)
 assume dfrees:⋀i. dfree (args i)
 assume const:⋀j. continuous-on UNIV (λx. Blinfun (extendf-deriv I i (args j)
ν x))
 then show ?case
  unfolding extendf-deriv.simps
  apply(cases f)
  subgoal for a
   apply simp
   proof −
   have boundedF:⋀x. bounded-linear (((THE f'. ∀ y. (Functions I a has-derivative
f' y) (at y))
                    (χ i. dterm-sem (extendf I x) (args i) ν) ))
    using blinfun.bounded-linear-right using good-interp unfolding is-interp-def

     by auto
    have boundedG:⋀x. bounded-linear (λ b. (χ ia. extendf-deriv I i (args ia) ν
x b))
     by (simp add: bounded-linear-vec dfrees extendf-deriv-bounded good-interp)
      have boundedH:⋀x. bounded-linear (λb. (THE f'. ∀ y. (Functions I a
has-derivative f' y) (at y))
                    (χ i. dterm-sem
                    (extendf I x)

                          (args i) ν)
                    (χ ia. extendf-deriv I i (args ia) ν x b))
```

using *bounded-linear-compose boundedG boundedF* **by** *blast*
have *eq*:(λ*x*. *Blinfun* (λ*b*. (*THE f'*. ∀ *y*. (*Functions I a has-derivative f' y*) (*at y*))
(χ *i*. *dterm-sem*
(*extendf I x*)
(*args i*) ν)
(χ *ia*. *extendf-deriv I i* (*args ia*) ν *x b*)))
=
(λ*x*. *blinfun-compose*(*Blinfun*((*THE f'*. ∀ *y*. (*Functions I a has-derivative f' y*) (*at y*))
(χ *i*. *dterm-sem*
(*extendf I x*)
(*args i*) ν))) (*Blinfun*(λ *b*. (χ *ia*. *extendf-deriv I i* (*args ia*) ν *x b*))))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
subgoal for *x ia*
using *boundedG*[*of x*] *blinfun-apply-blinfun-compose bounded-linear-Blinfun-apply*
proof −
have *f1*: *bounded-linear* (λ*v*. *FunctionFrechet I a* (χ *s*. *dterm-sem* (*extendf I x*) (*args s*) ν) (χ *s*. *extendf-deriv I i* (*args s*) ν *x v*))
using *FunctionFrechet.simps* ‹*bounded-linear* (λ*b*. (*THE f'*. ∀ *y*. (*Functions I a has-derivative f' y*) (*at y*)) (χ *i*. *dterm-sem* (*extendf I x*) (*args i*) ν) (χ *ia*. *extendf-deriv I i* (*args ia*) ν *x b*))›
by *fastforce*
have *bounded-linear* (*FunctionFrechet I a* (χ *s*. *dterm-sem* (*extendf I x*) (*args s*) ν))
using *good-interp is-interp-def* **by** *blast*
then have *blinfun-apply* (*Blinfun* (*FunctionFrechet I a* (χ *s*. *dterm-sem* (*extendf I x*) (*args s*) ν))) (χ *s*. *extendf-deriv I i* (*args s*) ν *x ia*) = *blinfun-apply* (*Blinfun* (λ*v*. *FunctionFrechet I a* (χ *s*. *dterm-sem* (*extendf I x*) (*args s*) ν) (χ *s*. *extendf-deriv I i* (*args s*) ν *x v*))) *ia*
using *f1* **by** (*simp add*: *bounded-linear-Blinfun-apply*)
then have *blinfun-apply* (*Blinfun* (*FunctionFrechet I a* (χ *s*. *dterm-sem* (*extendf I x*) (*args s*) ν))) (χ *s*. *extendf-deriv I i* (*args s*) ν *x ia*) = *blinfun-apply* (*Blinfun* (λ*v*. *FunctionFrechet I a* (χ *s*. *dterm-sem* (*extendf I x*) (*args s*) ν) (χ *s*. *extendf-deriv I i* (*args s*) ν *x v*))) *ia* ∧ *bounded-linear* (λ*v*. χ *s*. *extendf-deriv I i* (*args s*) ν *x v*)
by (*metis* ‹*bounded-linear* (λ*b*. χ *ia*. *extendf-deriv I i* (*args ia*) ν *x b*)›)
then show *?thesis*
by (*simp add*: *bounded-linear-Blinfun-apply*)
qed
done
have *bounds*:⋀*ia x*. *bounded-linear* (*extendf-deriv I i* (*args ia*) ν *x*)
by (*simp add*: *dfrees extendf-deriv-bounded good-interp*)
have *vec-bound*:⋀*x*. *bounded-linear* (λ*b*. χ *ia*. *extendf-deriv I i* (*args ia*) ν *x b*)
by (*simp add*: *boundedG*)
have *blinfun-vec*:(λ*x*. *Blinfun* (λ*b*. χ *ia*. *extendf-deriv I i* (*args ia*) ν *x b*)) =

(λx. *blinfun-vec* (λ *ia*. *Blinfun*(λb. *extendf-deriv I i* (*args ia*) ν *x b*)))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
apply(*rule vec-extensionality*)
subgoal for *x y ia*
proof −
have (χ *s. extendf-deriv I i* (*args s*) ν *x y*) $ *ia* = *blinfun-apply* (*blinfun-vec* (λs. *Blinfun* (*extendf-deriv I i* (*args s*) ν *x*))) *y* $ *ia*
by (*simp add*: *bounded-linear-Blinfun-apply bounds*)
then have (χ *s. extendf-deriv I i* (*args s*) ν *x y*) $ *ia* = *blinfun-apply* (*blinfun-vec* (λs. *Blinfun* (*extendf-deriv I i* (*args s*) ν *x*))) *y* $ *ia* ∧ *bounded-linear* (λv. χ *s. extendf-deriv I i* (*args s*) ν *x v*)
by (*metis* ‹*bounded-linear* (λb. χ *ia. extendf-deriv I i* (*args ia*) ν *x b*)›)
then show *?thesis*
by (*simp add*: *bounded-linear-Blinfun-apply*)
qed
done
have *vec-cont*:*continuous-on UNIV* (λx. *blinfun-vec* (λ *ia*. *Blinfun*(λb. *extendf-deriv I i* (*args ia*) ν *x b*)))
apply(*rule continuous-blinfun-vec'*)
using *3.IH* **by** *blast*
have *cont-intro*:⋀ *f g s. continuous-on s f* ⟹ *continuous-on s g* ⟹ *continuous-on s* (λx. *f x* $_L$ *g x*)
by(*auto intro*: *continuous-intros*)
have *cont*:*continuous-on UNIV* (λx. *blinfun-compose*(*Blinfun*((*THE f'*. ∀ *y*. (*Functions I a has-derivative f' y*) (*at y*))
(χ *i. dterm-sem*
(⦇*Functions* = *case-sum* (*Functions I*) (λ*f'* -. *x* $ *f'*), *Predicates* = *Predicates I*, *Contexts* = *Contexts I*,
Programs = *Programs I*, *ODEs* = *ODEs I*, *ODEBV* = *ODEBV I*⦈)
(*args i*) ν))) (*Blinfun*(λ *b*. (χ *ia. extendf-deriv I i* (*args ia*) ν *x b*))))
apply(*rule cont-intro*)
defer
subgoal using *blinfun-vec vec-cont* **by** *presburger*
apply(*rule continuous-on-compose2*[*of UNIV* (λx. *Blinfun* ((*THE f'*. ∀ *y*. (*Functions I a has-derivative f' y*) (*at y*)) *x*))])
subgoal using *good-interp* **unfolding** *is-interp-def* **by** *simp*
apply(*rule continuous-on-vec-lambda*)
subgoal for *i* **using** *extendf-dterm-sem-continuous*[*OF dfrees*[*of i*] *good-interp*] **by** *auto*
by *auto*
then show *continuous-on UNIV*
(λx. *Blinfun* (λb. (*THE f'*. ∀ *y*. (*Functions I a has-derivative f' y*) (*at y*))
(χ *i. dterm-sem*
(⦇*Functions* = *case-sum* (*Functions I*) (λ*f'* -. *x* $ *f'*), *Predicates* = *Predicates I*, *Contexts* = *Contexts I*,
Programs = *Programs I*, *ODEs* = *ODEs I*, *ODEBV*

$= ODEBV\ I\rbrbrack$
$(args\ i)\ \nu)$
$(\chi\ ia.\ extendf\text{-}deriv\ I\ i\ (args\ ia)\ \nu\ x\ b)))$
using *eq* **apply** *simp* **by** *presburger*
qed
by *simp*
next
case $(4\ \vartheta_1\ \vartheta_2)$
assume *free1*:*dfree* ϑ_1
assume *free2*:*dfree* ϑ_2
assume *IH1*:*continuous-on UNIV* $(\lambda x.\ Blinfun\ (extendf\text{-}deriv\ I\ i\ \vartheta_1\ \nu\ x))$
assume *IH2*:*continuous-on UNIV* $(\lambda x.\ Blinfun\ (extendf\text{-}deriv\ I\ i\ \vartheta_2\ \nu\ x))$
have *bound*:$\bigwedge x.\ bounded\text{-}linear\ (\lambda a.\ extendf\text{-}deriv\ I\ i\ \vartheta_1\ \nu\ x\ a + extendf\text{-}deriv\ I\ i\ \vartheta_2\ \nu\ x\ a)$
using *extendf-deriv-bounded*[*OF free1 good-interp*] *extendf-deriv-bounded*[*OF free2 good-interp*]
by (*simp add*: *bounded-linear-add*)
have *eq*:$(\lambda x.\ Blinfun\ (\lambda a.\ extendf\text{-}deriv\ I\ i\ \vartheta_1\ \nu\ x\ a + extendf\text{-}deriv\ I\ i\ \vartheta_2\ \nu\ x\ a)) = (\lambda x.\ Blinfun\ (\lambda a.\ extendf\text{-}deriv\ I\ i\ \vartheta_1\ \nu\ x\ a) + Blinfun\ (\lambda a.\ extendf\text{-}deriv\ I\ i\ \vartheta_2\ \nu\ x\ a))$
apply(*rule ext*)
apply(*rule blinfun-eqI*)
subgoal for *x j*
using *bound*[*of x*] *extendf-deriv-bounded*[*OF free1 good-interp*]
extendf-deriv-bounded[*OF free2 good-interp*]
blinfun.add-left[*of Blinfun* (*extendf-deriv I i* ϑ_1 ν *x*) *Blinfun* (*extendf-deriv I i* ϑ_2 ν *x*)]
bounded-linear-Blinfun-apply[*of* (*extendf-deriv I i* ϑ_1 ν *x*)]
bounded-linear-Blinfun-apply[*of* (*extendf-deriv I i* ϑ_2 ν *x*)]
by (*simp add*: *bounded-linear-Blinfun-apply*)
done
have *continuous-on UNIV* $(\lambda x.\ Blinfun\ (\lambda a.\ extendf\text{-}deriv\ I\ i\ \vartheta_1\ \nu\ x\ a) + Blinfun\ (\lambda a.\ extendf\text{-}deriv\ I\ i\ \vartheta_2\ \nu\ x\ a))$
apply(*rule continuous-intros*)
using *IH1 IH2* **by** *auto*
then show *?case*
apply *simp*
using *eq* **by** *presburger*
next
case $(5\ \vartheta_1\ \vartheta_2)$
assume *free1*:*dfree* ϑ_1
assume *free2*:*dfree* ϑ_2
assume *IH1*:*continuous-on UNIV* $(\lambda x.\ Blinfun\ (extendf\text{-}deriv\ I\ i\ \vartheta_1\ \nu\ x))$
assume *IH2*:*continuous-on UNIV* $(\lambda x.\ Blinfun\ (extendf\text{-}deriv\ I\ i\ \vartheta_2\ \nu\ x))$
have *bounded*:$\bigwedge x.\ bounded\text{-}linear\ (\lambda a.\ dterm\text{-}sem\ (extendf\ I\ x)\ \vartheta_1\ \nu * extendf\text{-}deriv\ I\ i\ \vartheta_2\ \nu\ x\ a +$
$extendf\text{-}deriv\ I\ i\ \vartheta_1\ \nu\ x\ a * dterm\text{-}sem\ (extendf\ I\ x)\ \vartheta_2\ \nu)$
using *extendf-deriv-bounded*[*OF free1 good-interp*] *extendf-deriv-bounded*[*OF free2 good-interp*]

by (*simp add*: *bounded-linear-add bounded-linear-const-mult bounded-linear-mult-const*)
have *eq*:(λ*x*. *Blinfun* (λ*a*. *dterm-sem* (*extendf I x*) ϑ$_1$ ν ∗ *extendf-deriv I i* ϑ$_2$ ν *x a* +
extendf-deriv I i ϑ$_1$ ν *x a* ∗ *dterm-sem* (*extendf I x*) ϑ$_2$ ν)) =
(λ*x*. *dterm-sem* (*extendf I x*) ϑ$_1$ ν ∗$_R$ *Blinfun* (λ*a*. *extendf-deriv I i* ϑ$_2$ ν *x a*) +
dterm-sem (*extendf I x*) ϑ$_2$ ν ∗$_R$ *Blinfun* (λ*a*. *extendf-deriv I i* ϑ$_1$ ν *x a*))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
subgoal for *x j*
using *extendf-deriv-bounded*[*OF free1 good-interp*] *extendf-deriv-bounded*[*OF free2 good-interp*] *bounded*[*of x*]
blinfun.scaleR-left
bounded-linear-Blinfun-apply[*of Blinfun* (*extendf-deriv I i* ϑ$_2$ ν *x*)]
bounded-linear-Blinfun-apply[*of Blinfun* (*extendf-deriv I i* ϑ$_1$ ν *x*)]
mult.commute
plus-blinfun.rep-eq[*of dterm-sem* (*extendf I x*) ϑ$_1$ ν ∗$_R$ *Blinfun* (*extendf-deriv I i* ϑ$_2$ ν *x*) *dterm-sem* (*extendf I x*) ϑ$_2$ ν ∗$_R$ *Blinfun* (*extendf-deriv I i* ϑ$_1$ ν *x*)]
real-scaleR-def
by (*simp add*: *blinfun.scaleR-left bounded-linear-Blinfun-apply*)
done
have *continuous-on UNIV* (λ*x*. *dterm-sem* (*extendf I x*) ϑ$_1$ ν ∗$_R$ *Blinfun* (λ*a*. *extendf-deriv I i* ϑ$_2$ ν *x a*) +
dterm-sem (*extendf I x*) ϑ$_2$ ν ∗$_R$ *Blinfun* (λ*a*. *extendf-deriv I i* ϑ$_1$ ν *x a*))
apply(*rule continuous-intros*)+
apply(*rule extendf-dterm-sem-continuous*[*OF free1 good-interp*])
apply(*rule IH2*)
apply(*rule continuous-intros*)+
apply(*rule extendf-dterm-sem-continuous*[*OF free2 good-interp*])
by(*rule IH1*)
then show *?case*
unfolding *extendf-deriv.simps*
using *eq* **by** *presburger*
qed (*auto intro*: *continuous-intros*)

lemma *extendf-deriv*:
fixes *f'*::('*sf* + '*sz*,'*sz*) *trm* **and** *I*::('*sf*,'*sc*,'*sz*) *interp*
assumes *free*:*dfree f'*
assumes *good-interp*:*is-interp I*
shows ∃*f''*. ∀*x*. ((λ*R*. *dterm-sem* (*extendf I R*) *f' ν*) *has-derivative* (*extendf-deriv I i-f f' ν x*)) (*at x*)
using *free* **apply** (*induction rule*: *dfree.induct*)
apply(*auto*)+
defer
subgoal for ϑ$_1$ ϑ$_2$ *x*
apply(*rule has-derivative-mult*)
by *auto*
subgoal for *args i x*
apply(*cases i*)

```
    defer
    apply auto
    subgoal for b using has-derivative-proj' by blast
   subgoal for a
  proof −
   assume dfrees:(⋀i. dfree (args i))
   assume IH1:(⋀ia. ∀ x. ((λR. dterm-sem
                (⦇Functions = case-sum (Functions I) (λf' -. R $ f'), Predicates
= Predicates I, Contexts = Contexts I, Programs = Programs I,
                  ODEs = ODEs I, ODEBV = ODEBV I⦈)
                (args ia) ν) has-derivative
            extendf-deriv I i-f (args ia) ν x)
            (at x))
   then have IH1':(⋀ia. ⋀x. ((λR. dterm-sem
                (⦇Functions = case-sum (Functions I) (λf' -. R $ f'), Predicates
= Predicates I, Contexts = Contexts I, Programs = Programs I,
                  ODEs = ODEs I, ODEBV = ODEBV I⦈)
                (args ia) ν) has-derivative
            extendf-deriv I i-f (args ia) ν x)
            (at x))
     by auto
   assume a:i = Inl a
   have chain:⋀f f' x s g g'. (f has-derivative f') (at x within s) ⟹
    (g has-derivative g') (at (f x) within f ‘ s) ⟹ (g ∘ f has-derivative g' ∘ f')
(at x within s)
     by (auto intro: derivative-intros)
   let ?f = (λx. Functions I a x)
   let ?g = (λ R. (χ i. dterm-sem
                (⦇Functions = case-sum (Functions I) (λf' -. R $ f'), Predicates
= Predicates I, Contexts = Contexts I,
                    Programs = Programs I, ODEs = ODEs I, ODEBV =
ODEBV I⦈)
                 (args i) ν))
   let ?myf' = (λx. (THE f'. ∀ y. (Functions I a has-derivative f' y) (at y)) (?g
x))
   let ?myg' = (λx. (λν'. χ ia. extendf-deriv I i-f (args ia) ν x ν'))
   have fg-eq:(λR. Functions I a
       (χ i. dterm-sem
            (⦇Functions = case-sum (Functions I) (λf' -. R $ f'), Predicates =
Predicates I, Contexts = Contexts I, Programs = Programs I,
              ODEs = ODEs I, ODEBV = ODEBV I⦈)
            (args i) ν)) = (?f ∘ ?g)
     by auto
   have ∀ x. ((?f o ?g) has-derivative (?myf' x ∘ ?myg' x)) (at x)
     apply (rule allI)
     apply (rule diff-chain-at)
     subgoal for xa
      apply (rule has-derivative-vec)
      subgoal for i using IH1'[of i xa] by auto
```

```
      done
     subgoal for xa
     proof −
      have deriv:⋀x. (Functions I a has-derivative FunctionFrechet I a x) (at x)
      and cont:continuous-on UNIV (λx. Blinfun (FunctionFrechet I a x))
       using good-interp[unfolded is-interp-def] by auto
      show ?thesis
       apply(rule has-derivative-at-withinI)
       using deriv by auto
     qed
    done
  then have ((?f o ?g) has-derivative (?myf' x ∘ ?myg' x)) (at x) by auto
  then show ((λR. Functions I a
       (χ i. dterm-sem
            (|Functions = case-sum (Functions I) (λf' -. R $ f'), Predicates =
Predicates I, Contexts = Contexts I, Programs = Programs I,
              ODEs = ODEs I, ODEBV = ODEBV I|)
            (args i) ν)) has-derivative
        (THE f'. ∀ y. (Functions I a has-derivative f' y) (at y))
    (χ i. dterm-sem
          (|Functions = case-sum (Functions I) (λf' -. x $ f'), Predicates =
Predicates I, Contexts = Contexts I, Programs = Programs I,
            ODEs = ODEs I, ODEBV = ODEBV I|)
          (args i) ν) ∘
    (λν'. χ ia. extendf-deriv I i-f (args ia) ν x ν'))
    (at x)
    using fg-eq by auto
 qed
 done
done

lemma adjoint-safe:
 assumes good-interp:is-interp I
 assumes good-subst:(⋀i f'. SFunctions σ i = Some f' ⟹ dfree f')
 shows is-interp (adjoint I σ ν)
 apply(unfold adjoint-def)
 apply(unfold is-interp-def)
 apply(auto simp del: extendf.simps extendc.simps FunctionFrechet.simps)
  subgoal for x i
   apply(cases SFunctions σ i = None)
    subgoal
     apply(auto simp del: extendf.simps extendc.simps)
     using good-interp unfolding is-interp-def by simp
    apply(auto  simp del: extendf.simps extendc.simps)
    subgoal for f'
     using good-subst[of i f'] apply (auto  simp del: extendf.simps extendc.simps)
    proof −
     assume some:SFunctions σ i = Some f'
     assume free:dfree f'
```

```
        let ?f = (λR. dterm-sem (extendf I R) f' ν)
        let ?Pred = (λfd. (∀x. (?f has-derivative (fd x)) (at x)))
        let ?f''=extendf-deriv I i f' ν
        have Pf:?Pred ?f''
          using extendf-deriv[OF good-subst[of i f'] good-interp, of ν i, OF some]
          by auto
        have (THE G. (?f has-derivative G) (at x)) = ?f'' x
          apply(rule the-deriv)
          using Pf by auto
        then have the-eq:(THE G. ∀ x. (?f has-derivative G x) (at x)) = ?f''
          using Pf the-all-deriv by auto
         show ((λR. dterm-sem (extendf I R) f' ν) has-derivative (THE f'a. ∀x.
((λR. dterm-sem (extendf I R) f' ν) has-derivative f'a x) (at x)) x) (at x)
          using the-eq Pf by simp
      qed
      done
    subgoal for i
      apply(cases SFunctions σ i = None)
       subgoal
         apply(auto  simp del: extendf.simps extendc.simps)
         using good-interp unfolding is-interp-def by simp
      apply(auto  simp del: extendf.simps extendc.simps)
      subgoal for f'
       using good-subst[of i f'] apply (auto  simp del: extendf.simps extendc.simps)
      proof −
        assume some:SFunctions σ i = Some f'
        assume free:dfree f'
        let ?f = (λR. dterm-sem (extendf I R) f' ν)
        let ?Pred = (λfd. (∀x. (?f has-derivative (fd x)) (at x)))
        let ?f''=extendf-deriv I i f' ν
        have Pf:?Pred ?f''
          using extendf-deriv[OF good-subst[of i f'] good-interp, of ν i, OF some]
          by auto
        have ⋀x. (THE G. (?f has-derivative G) (at x)) = ?f'' x
          apply(rule the-deriv)
          using Pf by auto
        then have the-eq:(THE G. ∀ x. (?f has-derivative G x) (at x)) = ?f''
          using Pf the-all-deriv by auto
        have continuous-on UNIV (λx. Blinfun (?f'' x))
          by(rule extendf-deriv-continuous[OF free good-interp])
         show continuous-on UNIV (λx. Blinfun ((THE f'a. ∀x. ((λR. dterm-sem
(extendf I R) f' ν) has-derivative f'a x) (at x)) x))
          using the-eq Pf
           by (simp add: ‹continuous-on UNIV (λx. Blinfun (extendf-deriv I i f' ν
x))›)
      qed
    done
  done
```

lemma *adjointFO-safe*:
assumes *good-interp*:*is-interp I*
assumes *good-subst*:($\bigwedge i$. *dsafe* (σ *i*))
shows *is-interp* (*adjointFO I* σ ν)
apply(*unfold adjointFO-def*)
apply(*unfold is-interp-def*)
apply(*auto simp del*: *extendf.simps extendc.simps FunctionFrechet.simps*)
subgoal for *x i*
apply(*cases i*)
subgoal
apply(*auto simp del*: *extendf.simps extendc.simps*)
using *good-interp* **unfolding** *is-interp-def* **by** *simp*
apply(*auto simp del*: *extendf.simps extendc.simps*)
subgoal for f'
proof −
assume *some*:$i = Inr\ f'$
have *free*:*dsafe* (σ f') **using** *good-subst* **by** *auto*
let *?f* = (λ-. *dterm-sem I* (σ f') ν)
let *?Pred* = (λfd. ($\forall x$. (*?f has-derivative* (*fd x*)) (*at x*)))
let *?f''*=(λ- -. *0*)
have *Pf*:*?Pred ?f''*
proof (*induction* σ f')
qed (*auto*)
have (*THE G*. (*?f has-derivative G*) (*at x*)) = *?f'' x*
apply(*rule the-deriv*)
using *Pf* **by** *auto*
then have *the-eq*:(*THE G*. $\forall$ *x*. (*?f has-derivative G x*) (*at x*)) = *?f''*
using *Pf the-all-deriv*[*of ?f ?f''*] **by** *auto*
have *another-eq*:(*THE f'a*. $\forall x$. ((λ-. *dterm-sem I* (σ f') ν) *has-derivative* *f'a x*) (*at x*)) *x* = (λ -. *0*)
using *Pf* **by** (*simp add*: *the-eq*)
then show ((λ-. *dterm-sem I* (σ f') ν) *has-derivative* (*THE f'a*. $\forall x$. ((λ-. *dterm-sem I* (σ f') ν) *has-derivative f'a x*) (*at x*)) *x*) (*at x*)
using *the-eq Pf* **by** *simp*
qed
done
subgoal for *i*
apply(*cases i*)
subgoal
apply(*auto simp del*: *extendf.simps extendc.simps*)
using *good-interp* **unfolding** *is-interp-def* **by** *simp*
apply(*auto simp del*: *extendf.simps extendc.simps*)
subgoal for f'
using *good-subst*[*of f'*]
proof −
assume *some*:$i = Inr\ f'$
have *free*:*dsafe* (σ f') **using** *good-subst* **by** *auto*
let *?f* = (λR. *dterm-sem I* (σ f') ν)
let *?Pred* = (λfd. ($\forall x$. (*?f has-derivative* (*fd x*)) (*at x*)))

let *?f''*=(λ- -. *0*)
have *Pf*:*?Pred ?f''* **by** *simp*
have ⋀*x*. (*THE G*. (*?f has-derivative G*) (*at x*)) = *?f'' x*
apply(*rule the-deriv*)
using *Pf* **by** *auto*
then have *the-eq*:(*THE G*. ∀ *x*. (*?f has-derivative G x*) (*at x*)) = *?f''*
using *Pf the-all-deriv*[*of* (λ*R*. *dterm-sem I* (σ *f'*) ν) (λ- -. *0*)]
by *blast*
then have *blin-cont*:*continuous-on UNIV* (λ*x*. *Blinfun* (*?f'' x*))
by (*simp add*: *continuous-on-const*)
have *truth*:(λ*x*. *Blinfun* ((*THE f'a*. ∀ *x*. ((λ-. *dterm-sem I* (σ *f'*) ν) *has-derivative f'a x*) (*at x*)) *x*))
= (λ*x*. *Blinfun* (λ -. *0*))
apply(*rule ext*)
apply(*rule blinfun-eqI*)
by (*simp add*: *local.the-eq*)
then show *continuous-on UNIV* (λ*x*. *Blinfun* ((*THE f'a*. ∀ *x*. ((λ-. *dterm-sem I* (σ *f'*) ν) *has-derivative f'a x*) (*at x*)) *x*))
using *truth*
by (*metis* (*mono-tags*, *lifting*) *blin-cont continuous-on-eq*)
qed
done
done

11.2 Lemmas about adjoint interpretations

lemma *adjoint-consequence*:(⋀*f f'*. *SFunctions* σ *f* = *Some f'* ⟹ *dsafe f'*) ⟹ (⋀*f f'*. *SPredicates* σ *f* = *Some f'* ⟹ *fsafe f'*) ⟹ *Vagree* ν ω (*FVS* σ) ⟹ *adjoint I* σ ν = *adjoint I* σ ω
apply(*unfold FVS-def*)
apply(*auto*)
apply(*unfold adjoint-def*)
apply(*rule interp-eq*)
apply(*auto simp add*: *fun-eq-iff*)
subgoal for *xa xaa*
apply(*cases SFunctions* σ *xa*)
apply(*auto*)
subgoal for *a*
proof −
assume *safes*:(⋀*f f'*. *SFunctions* σ *f* = *Some f'* ⟹ *dsafe f'*)
assume *agrees*:*Vagree* ν ω (⋃*x*. *SFV* σ *x*)
assume *some*:*SFunctions* σ *xa* = *Some a*
from *safes some* **have** *safe*:*dsafe a* **by** *auto*
have *sub*:*SFV* σ (*Inl xa*) ⊆ (⋃*x*. *SFV* σ *x*)
by *blast*
from *agrees*
have *Vagree* ν ω (*SFV* σ (*Inl xa*))
using *agree-sub*[*OF sub agrees*] **by** *auto*
then have *agree*:*Vagree* ν ω (*FVT a*)

```
      using some by auto
    show ?thesis
      using coincidence-dterm[of a, OF safes[of xa a, OF some] agree] by auto
  qed
 done
subgoal for xa xaa
 apply(cases SPredicates σ xa)
  apply(auto)
 subgoal for a
 proof −
  assume safes:(⋀f f'. SPredicates σ f = Some f' ⟹ fsafe f')
  assume agrees:Vagree ν ω (⋃x. SFV σ x)
  assume some:SPredicates σ xa = Some a
  assume sem:ν ∈ fml-sem (|Functions = case-sum (Functions I) (λf' -. xaa $
f'), Predicates = Predicates I, Contexts = Contexts I, Programs = Programs I,
                ODEs = ODEs I, ODEBV = ODEBV I|)
    a
  from safes some have safe:fsafe a by auto
  have sub:SFV σ (Inr (Inr xa)) ⊆ (⋃x. SFV σ x)
    by blast
  from agrees
  have Vagree ν ω (SFV σ (Inr (Inr xa)))
    using agree-sub[OF sub agrees] by auto
  then have agree:Vagree ν ω (FVF a)
    using some by auto
  let ?I' = (|Functions = case-sum (Functions I) (λf' -. xaa $ f'), Predicates
= Predicates I, Contexts = Contexts I, Programs = Programs I,
                ODEs = ODEs I, ODEBV = ODEBV I|)
  have IA:⋀S. Iagree ?I' ?I' (SIGF a) using Iagree-refl by auto
  show ?thesis
    using coincidence-formula[of a, OF safes[of xa a, OF some] IA agree] sem
by auto
 qed
 done
subgoal for xa xaa
 apply(cases SPredicates σ xa)
  apply(auto)
 subgoal for a
 proof −
  assume safes:(⋀f f'. SPredicates σ f = Some f' ⟹ fsafe f')
  assume agrees:Vagree ν ω (⋃x. SFV σ x)
  assume some:SPredicates σ xa = Some a
  assume sem:ω ∈ fml-sem (|Functions = case-sum (Functions I) (λf' -. xaa $
f'), Predicates = Predicates I, Contexts = Contexts I, Programs = Programs I,
                ODEs = ODEs I, ODEBV = ODEBV I|)
    a
  from safes some have safe:fsafe a by auto
  have sub:SFV σ (Inr (Inr xa)) ⊆ (⋃x. SFV σ x)
    by blast
```

from *agrees*
have *Vagree ν ω (SFV σ (Inr (Inr xa)))*
using *agree-sub*[*OF sub agrees*] **by** *auto*
then have *agree*: *Vagree ν ω (FVF a)*
using *some* **by** *auto*
let *?I′* = (|*Functions* = *case-sum* (*Functions I*) (λ*f′* -. *xaa* $ *f′*), *Predicates* = *Predicates I*, *Contexts* = *Contexts I*, *Programs* = *Programs I*,
ODEs = *ODEs I*, *ODEBV* = *ODEBV I*|)
have *IA*:⋀*S*. *Iagree ?I′ ?I′* (*SIGF a*) **using** *Iagree-refl* **by** *auto*
show *?thesis*
using *coincidence-formula*[*of a*, *OF safes*[*of xa a*, *OF some*] *IA agree*] *sem* **by** *auto*
qed
done
done

lemma *SIGT-plus1*: *Vagree ν ω* (⋃*i*∈*SIGT* (*Plus t1 t2*). *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
⟹ *Vagree ν ω* (⋃*i*∈*SIGT t1*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
unfolding *Vagree-def* **by** *auto*

lemma *SIGT-plus2*: *Vagree ν ω* (⋃*i*∈*SIGT* (*Plus t1 t2*). *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
⟹ *Vagree ν ω* (⋃*i*∈*SIGT t2*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
unfolding *Vagree-def* **by** *auto*

lemma *SIGT-times1*: *Vagree ν ω* (⋃*i*∈*SIGT* (*Times t1 t2*). *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
⟹ *Vagree ν ω* (⋃*i*∈*SIGT t1*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
unfolding *Vagree-def* **by** *auto*

lemma *SIGT-times2*: *Vagree ν ω* (⋃*i*∈*SIGT* (*Times t1 t2*). *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
⟹ *Vagree ν ω* (⋃*i*∈*SIGT t2*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
unfolding *Vagree-def* **by** *auto*

lemma *uadmit-sterm-adjoint′*:
assumes *dsafe*:⋀*f f′*. *SFunctions σ f* = *Some f′* ⟹ *dsafe f′*
assumes *fsafe*:⋀*f f′*. *SPredicates σ f* = *Some f′* ⟹ *fsafe f′*
shows *Vagree ν ω* (⋃*i*∈*SIGT ϑ*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {}) ⟹ *sterm-sem* (*adjoint I σ ν*) *ϑ* = *sterm-sem* (*adjoint I σ ω*) *ϑ*
proof (*induct ϑ*)
case (*Plus ϑ1 ϑ2*)
assume *IH1*: *Vagree ν ω* (⋃*i*∈*SIGT ϑ1*. *case SFunctions σ i of Some a* ⇒ *FVT a* | *None* ⇒ {}) ⟹ *sterm-sem* (*local.adjoint I σ ν*) *ϑ1* = *sterm-sem* (*local.adjoint*

I σ ω) ϑ1
assume *IH2*:*Vagree ν ω* (⋃*i*∈*SIGT ϑ2. case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*) ⟹ *sterm-sem* (*local.adjoint I σ ν*) *ϑ2* = *sterm-sem* (*local.adjoint I σ ω*) *ϑ2*
assume *VA*:*Vagree ν ω* (⋃*i*∈*SIGT* (*Plus ϑ1 ϑ2*). *case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*)
then show *?case*
using *IH1*[*OF SIGT-plus1*[*OF VA*]] *IH2*[*OF SIGT-plus2*[*OF VA*]] **by** *auto*
next
case (*Times ϑ1 ϑ2*)
assume *IH1*:*Vagree ν ω* (⋃*i*∈*SIGT ϑ1. case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*) ⟹ *sterm-sem* (*local.adjoint I σ ν*) *ϑ1* = *sterm-sem* (*local.adjoint I σ ω*) *ϑ1*
assume *IH2*:*Vagree ν ω* (⋃*i*∈*SIGT ϑ2. case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*) ⟹ *sterm-sem* (*local.adjoint I σ ν*) *ϑ2* = *sterm-sem* (*local.adjoint I σ ω*) *ϑ2*
assume *VA*:*Vagree ν ω* (⋃*i*∈*SIGT* (*Times ϑ1 ϑ2*). *case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*)
then show *?case*
using *IH1*[*OF SIGT-times1*[*OF VA*]] *IH2*[*OF SIGT-times2*[*OF VA*]] **by** *auto*
next
case (*Function x1a x2a*)
assume *IH*:⋀*x. x* ∈ *range x2a* ⟹ *Vagree ν ω* (⋃*i*∈*SIGT x. case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*) ⟹
sterm-sem (*local.adjoint I σ ν*) *x* = *sterm-sem* (*local.adjoint I σ ω*) *x*
from *IH* **have** *IH′*:⋀*j. Vagree ν ω* (⋃*i*∈*SIGT* (*x2a j*). *case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*) ⟹
sterm-sem (*local.adjoint I σ ν*) (*x2a j*) = *sterm-sem* (*local.adjoint I σ ω*) (*x2a j*)
using *rangeI* **by** *auto*
assume *VA*:*Vagree ν ω* (⋃*i*∈*SIGT* (*$f x1a x2a*). *case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*)
from *VA* **have** *VAs*:⋀*j. Vagree ν ω* (⋃*i*∈*SIGT* (*x2a j*). *case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*)
unfolding *Vagree-def SIGT.simps* **using** *rangeI* **by** *blast*
have *SIGT*:*x1a* ∈ *SIGT* (*$f x1a x2a*) **by** *auto*
have *VAsub*:⋀*a. SFunctions σ x1a* = *Some a* ⟹ (*FVT a*) ⊆ (⋃*i*∈*SIGT* (*$f x1a x2a*). *case SFunctions σ i of Some a ⇒ FVT a | None ⇒ {}*)
using *SIGT* **by** *auto*
have *VAf*:⋀*a. SFunctions σ x1a* = *Some a* ⟹ *Vagree ν ω* (*FVT a*)
using *agree-sub*[*OF VAsub VA*] **by** *auto*
then show *?case*
using *IH′*[*OF VAs*] **apply** (*auto simp add*: *fun-eq-iff*)
apply(*cases SFunctions σ x1a*)
defer
subgoal for *x a*
proof −
assume *VA*:(⋀*a. SFunctions σ x1a* = *Some a* ⟹ *Vagree ν ω* (*FVT a*))
assume *sems*:(⋀*j.* ∀ *x. sterm-sem* (*local.adjoint I σ ν*) (*x2a j*) *x* = *sterm-sem*

(*local.adjoint I σ ω*) (*x2a j*) *x*)
assume *some*:*SFunctions σ x1a* = *Some a*
note *FVT* = *VAf*[*OF some*]
have *dsem*:⋀*R* . *dterm-sem* (*extendf I R*) *a ν* = *dterm-sem* (*extendf I R*) *a ω*
using *coincidence-dterm*[*OF dsafe*[*OF some*] *FVT*] **by** *auto*
have ⋀*R*. *Functions* (*local.adjoint I σ ν*) *x1a R* = *Functions* (*local.adjoint I σ ω*) *x1a R*
using *dsem some* **unfolding** *adjoint-def* **by** *auto*
then show *Functions* (*local.adjoint I σ ν*) *x1a* (*χ i. sterm-sem* (*local.adjoint I σ ω*) (*x2a i*) *x*) =
Functions (*local.adjoint I σ ω*) *x1a* (*χ i. sterm-sem* (*local.adjoint I σ ω*) (*x2a i*) *x*)
by *auto*
qed
unfolding *adjoint-def* **apply** *auto*
done
qed (*auto*)

— Not used, but good practice for *dterm* adjoint
lemma *uadmit-sterm-adjoint*:
assumes *TUA*:*TUadmit σ ϑ U*
assumes *VA*:*Vagree ν ω* (−*U*)
assumes *dsafe*:⋀*f f'*. *SFunctions σ f* = *Some f'* ⟹ *dsafe f'*
assumes *fsafe*:⋀*f f'*. *SPredicates σ f* = *Some f'* ⟹ *fsafe f'*
shows *sterm-sem* (*adjoint I σ ν*) *ϑ* = *sterm-sem* (*adjoint I σ ω*) *ϑ*
proof −
have *duh*:⋀*A B*. *A* ∩ *B* = {} ⟹ *A* ⊆ −*B*
by *auto*
have ⋀*x*. *x* ∈ (⋃*i*∈*SIGT ϑ*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {}) ⟹ *x* ∈ (−*U*)
using *TUA* **unfolding** *TUadmit-def* **by** *auto*
then have *sub1*:(⋃*i*∈*SIGT ϑ*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {}) ⊆ −*U*
by *auto*
then have *VA'*:*Vagree ν ω* (⋃*i*∈*SIGT ϑ*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {})
using *agree-sub*[*OF sub1 VA*] **by** *auto*
then show *?thesis* **using** *uadmit-sterm-adjoint'*[*OF dsafe fsafe VA'*] **by** *auto*
qed

lemma *uadmit-sterm-ntadjoint'*:
assumes *dsafe*:⋀*i*. *dsafe* (*σ i*)
shows *Vagree ν ω* ((⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ*}. *FVT* (*σ i*))) ⟹ *sterm-sem* (*adjointFO I σ ν*) *ϑ* = *sterm-sem* (*adjointFO I σ ω*) *ϑ*
proof (*induct ϑ*)
case (*Plus ϑ1 ϑ2*)
assume *IH1*:*Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ1*}. *FVT* (*σ i*)) ⟹ *sterm-sem* (*adjointFO I σ ν*) *ϑ1* = *sterm-sem* (*adjointFO I σ ω*) *ϑ1*

assume *IH2*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* $\vartheta 2$}. *FVT* (σ *i*)) ⟹ *sterm-sem* (*adjointFO I* σ ν) $\vartheta 2$ = *sterm-sem* (*adjointFO I* σ ω) $\vartheta 2$
assume *VA*:*Vagree* ν ω (($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* (*Plus* $\vartheta 1$ $\vartheta 2$)}. *FVT* (σ *i*)))
from *VA*
have *VA1*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* $\vartheta 1$}. *FVT* (σ *i*))
and *VA2*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* $\vartheta 2$}. *FVT* (σ *i*)) **unfolding** *Vagree-def* **by** *auto*
then show *?case*
using *IH1*[*OF VA1*] *IH2*[*OF VA2*] **by** *auto*
next
case (*Times* $\vartheta 1$ $\vartheta 2$)
assume *IH1*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* $\vartheta 1$}. *FVT* (σ *i*)) ⟹ *sterm-sem* (*adjointFO I* σ ν) $\vartheta 1$ = *sterm-sem* (*adjointFO I* σ ω) $\vartheta 1$
assume *IH2*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* $\vartheta 2$}. *FVT* (σ *i*)) ⟹ *sterm-sem* (*adjointFO I* σ ν) $\vartheta 2$ = *sterm-sem* (*adjointFO I* σ ω) $\vartheta 2$
assume *VA*:*Vagree* ν ω (($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* (*Times* $\vartheta 1$ $\vartheta 2$)}. *FVT* (σ *i*)))
from *VA*
have *VA1*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* $\vartheta 1$}. *FVT* (σ *i*))
and *VA2*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* $\vartheta 2$}. *FVT* (σ *i*)) **unfolding** *Vagree-def* **by** *auto*
then show *?case*
using *IH1*[*OF VA1*] *IH2*[*OF VA2*] **by** *auto*
next
case (*Function x1a x2a*)
assume *IH*:$\bigwedge$*x*. *x* ∈ *range x2a* ⟹ *Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT x*}. *FVT* (σ *i*)) ⟹
sterm-sem (*adjointFO I* σ ν) *x* = *sterm-sem* (*adjointFO I* σ ω) *x*
from *IH* **have** *IH'*:$\bigwedge$*j*. *Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* (*x2a j*)}. *FVT* (σ *i*)) ⟹
sterm-sem (*adjointFO I* σ ν) (*x2a j*) = *sterm-sem* (*adjointFO I* σ ω) (*x2a j*)
using *rangeI* **by** *auto*
assume *VA*:*Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* ($f *x1a x2a*)}. *FVT* ($\sigma$ *i*))
from *VA* **have** *VAs*:$\bigwedge$*j*. *Vagree* ν ω ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* (*x2a j*)}. *FVT* (σ *i*))
unfolding *Vagree-def SIGT.simps* **using** *rangeI* **by** *blast*
have *SIGT*:*x1a* ∈ *SIGT* ($f *x1a x2a*) **by** *auto*
have *VAsub*:$\bigwedge$*a*. *x1a* = *Inr a* ⟹ (*FVT* (σ *a*)) ⊆ ($\bigcup$ i∈{*i*. *Inr i* ∈ *SIGT* ($f *x1a x2a*)}. *FVT* ($\sigma$ *i*))
using *SIGT* **by** *auto*
have *VAf*:$\bigwedge$*a*. *x1a* = *Inr a* ⟹*Vagree* ν ω (*FVT* (σ *a*))
using *agree-sub*[*OF VAsub VA*] **by** *auto*
then show *?case*
using *IH'*[*OF VAs*] **apply** (*auto simp add*: *fun-eq-iff*)
apply(*cases x1a*)
defer
subgoal for *x a*
proof −
assume *VA*:($\bigwedge$*a*. *x1a* = *Inr a* ⟹ *Vagree* ν ω (*FVT* (σ *a*)))
assume *sems*:($\bigwedge$*j*. ∀ *x*. *sterm-sem* (*adjointFO I* σ ν) (*x2a j*) *x* = *sterm-sem*

(*adjointFO I σ ω*) (*x2a j*) *x*)
assume *some*:*x1a* = *Inr a*
note *FVT* = *VAf*[*OF some*]
from *dsafe* **have** *dsafer*:⋀*i*. *dsafe* (*σ i*) **using** *dfree-is-dsafe* **by** *auto*
have *dsem*:*dterm-sem I* (*σ a*) *ν* = *dterm-sem I* (*σ a*) *ω*
using *coincidence-dterm*[*OF dsafer FVT*] *some* **by** *auto*
then have ⋀*R*. *Functions* (*adjointFO I σ ν*) *x1a R* = *Functions* (*adjointFO I σ ω*) *x1a R*
using *some* **unfolding** *adjoint-def* **unfolding** *adjointFO-def* **by** *auto*
then show *Functions* (*adjointFO I σ ν*) *x1a* (*χ i*. *sterm-sem* (*adjointFO I σ ω*) (*x2a i*) *x*) =
Functions (*adjointFO I σ ω*) *x1a* (*χ i*. *sterm-sem* (*adjointFO I σ ω*) (*x2a i*) *x*)
by *auto*
qed
unfolding *adjointFO-def* **by** *auto*
qed (*auto*)

lemma *uadmit-sterm-ntadjoint*:
assumes *TUA*:*NTUadmit σ ϑ U*
assumes *VA*:*Vagree ν ω* (−*U*)
assumes *dsafe*:⋀*i* . *dsafe* (*σ i*)
assumes *good-interp*:*is-interp I*
shows *sterm-sem* (*adjointFO I σ ν*) *ϑ* = *sterm-sem* (*adjointFO I σ ω*) *ϑ*
proof −
have *duh*:⋀*A B*. *A* ∩ *B* = {} ⟹ *A* ⊆ −*B*
by *auto*
have ⋀*x*. *x* ∈ ((⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ*}. *FVT* (*σ i*))) ⟹ *x* ∈ (−*U*)
using *TUA* **unfolding** *NTUadmit-def* **by** *auto*
then have *sub1*:(⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ*}. *FVT* (*σ i*)) ⊆ −*U*
by *auto*
then have *VA'*:*Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ*}. *FVT* (*σ i*))
using *agree-sub*[*OF sub1 VA*] **by** *auto*
then show *?thesis* **using** *uadmit-sterm-ntadjoint'*[*OF dsafe VA'*] **by** *auto*
qed

lemma *uadmit-dterm-adjoint'*:
assumes *dfree*:⋀*f f'*. *SFunctions σ f* = *Some f'* ⟹ *dfree f'*
assumes *fsafe*:⋀*f f'*. *SPredicates σ f* = *Some f'* ⟹ *fsafe f'*
assumes *good-interp*:*is-interp I*
shows ⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈*SIGT ϑ*. *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {}) ⟹ *dsafe ϑ* ⟹ *dterm-sem* (*adjoint I σ ν*) *ϑ* = *dterm-sem* (*adjoint I σ ω*) *ϑ*
proof (*induct ϑ*)
case (*Plus ϑ1 ϑ2*)
assume *IH1*:⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈*SIGT ϑ1*. *case SFunctions σ i of Some a* ⇒ *FVT a* | *None* ⇒ {}) ⟹ *dsafe ϑ1* ⟹ *dterm-sem* (*local.adjoint I σ ν*) *ϑ1* = *dterm-sem* (*local.adjoint I σ ω*) *ϑ1*
assume *IH2*:⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈*SIGT ϑ2*. *case SFunctions σ i of Some a*

$\Rightarrow$ *FVT a | None* $\Rightarrow$ {}) $\Longrightarrow$ *dsafe* $\vartheta 2$ $\Longrightarrow$ *dterm-sem* (*local.adjoint I* σ ν) $\vartheta 2$ = *dterm-sem* (*local.adjoint I* σ ω) $\vartheta 2$
assume *VA*:*Vagree* ν ω ($\bigcup i{\in}SIGT$ (*Plus* $\vartheta 1$ $\vartheta 2$). *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {})
assume *safe*:*dsafe* (*Plus* $\vartheta 1$ $\vartheta 2$)
then show *?case*
using *IH1*[*OF SIGT-plus1*[*OF VA*]] *IH2*[*OF SIGT-plus2*[*OF VA*]] **by** *auto*
next
case (*Times* $\vartheta 1$ $\vartheta 2$)
assume *IH1*:$\bigwedge\nu$ ω. *Vagree* ν ω ($\bigcup i{\in}SIGT$ $\vartheta 1$. *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {}) $\Longrightarrow$ *dsafe* $\vartheta 1$ $\Longrightarrow$ *dterm-sem* (*local.adjoint I* σ ν) $\vartheta 1$ = *dterm-sem* (*local.adjoint I* σ ω) $\vartheta 1$
assume *IH2*:$\bigwedge\nu$ ω. *Vagree* ν ω ($\bigcup i{\in}SIGT$ $\vartheta 2$. *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {}) $\Longrightarrow$ *dsafe* $\vartheta 2$ $\Longrightarrow$ *dterm-sem* (*local.adjoint I* σ ν) $\vartheta 2$ = *dterm-sem* (*local.adjoint I* σ ω) $\vartheta 2$
assume *VA*:*Vagree* ν ω ($\bigcup i{\in}SIGT$ (*Times* $\vartheta 1$ $\vartheta 2$). *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {})
assume *safe*:*dsafe* (*Times* $\vartheta 1$ $\vartheta 2$)
then show *?case*
using *IH1*[*OF SIGT-times1*[*OF VA*]] *IH2*[*OF SIGT-times2*[*OF VA*]] **by** *auto*
next
case (*Function x1a x2a*)
assume *IH*:$\bigwedge x$. $\bigwedge\nu$ ω. $x \in$ *range x2a* $\Longrightarrow$ *Vagree* ν ω ($\bigcup i{\in}SIGT$ *x. case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {}) $\Longrightarrow$
dsafe x $\Longrightarrow$ *dterm-sem* (*local.adjoint I* σ ν) *x* = *dterm-sem* (*local.adjoint I* σ ω) *x*
assume *safe*:*dsafe* (*Function x1a x2a*)
from *safe* **have** *safes*:$\bigwedge j$. *dsafe* (*x2a j*) **by** *auto*
from *IH* **have** *IH′*:$\bigwedge j$. *Vagree* ν ω ($\bigcup i{\in}SIGT$ (*x2a j*). *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {}) $\Longrightarrow$
dterm-sem (*local.adjoint I* σ ν) (*x2a j*) = *dterm-sem* (*local.adjoint I* σ ω) (*x2a j*)
using *rangeI safes* **by** *auto*
assume *VA*:*Vagree* ν ω ($\bigcup i{\in}SIGT$ (\$*f x1a x2a*). *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {})
from *VA* **have** *VAs*:$\bigwedge j$. *Vagree* ν ω ($\bigcup i{\in}SIGT$ (*x2a j*). *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {})
unfolding *Vagree-def SIGT.simps* **using** *rangeI* **by** *blast*
have *SIGT*:*x1a* $\in$ *SIGT* (\$*f x1a x2a*) **by** *auto*
have *VAsub*:$\bigwedge a$. *SFunctions* σ *x1a* = *Some a* $\Longrightarrow$ (*FVT a*) $\subseteq$ ($\bigcup i{\in}SIGT$ (\$*f x1a x2a*). *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a | None* $\Rightarrow$ {})
using *SIGT* **by** *auto*
have *VAf*:$\bigwedge a$. *SFunctions* σ *x1a* = *Some a* $\Longrightarrow$ *Vagree* ν ω (*FVT a*)
using *agree-sub*[*OF VAsub VA*] **by** *auto*
then show *?case*
using *IH′*[*OF VAs*] **apply** (*auto simp add*: *fun-eq-iff*)
apply(*cases SFunctions* σ *x1a*)
defer
subgoal for *x1 x2 a*

proof −
assume *VA*:($\bigwedge a$. *SFunctions* σ *x1a* = *Some a* $\Longrightarrow$ *Vagree* ν ω (*FVT a*))
assume *sems*:($\bigwedge j$. $\forall$ *x1 x2*. *dterm-sem* (*local.adjoint I* σ ν) (*x2a j*) (*x1*,*x2*) = *dterm-sem* (*local.adjoint I* σ ω) (*x2a j*) (*x1*,*x2*))
assume *some*:*SFunctions* σ *x1a* = *Some a*
note *FVT* = *VAf*[*OF some*]
have *dsafe*:$\bigwedge f\, f'$. *SFunctions* σ *f* = *Some f'* $\Longrightarrow$ *dsafe f'*
using *dfree dfree-is-dsafe* **by** *auto*
have *dsem*:$\bigwedge R$. *dterm-sem* (*extendf I R*) *a* ν = *dterm-sem* (*extendf I R*) *a* ω
using *coincidence-dterm*[*OF dsafe*[*OF some*] *FVT*] **by** *auto*
have $\bigwedge R$. *Functions* (*local.adjoint I* σ ν) *x1a R* = *Functions* (*local.adjoint I* σ ω) *x1a R*
using *dsem some* **unfolding** *adjoint-def* **by** *auto*
then show *Functions* (*local.adjoint I* σ ν) *x1a* (χ *i*. *dterm-sem* (*local.adjoint I* σ ω) (*x2a i*) (*x1*,*x2*)) =
Functions (*local.adjoint I* σ ω) *x1a* (χ *i*. *dterm-sem* (*local.adjoint I* σ ω) (*x2a i*) (*x1*,*x2*))
by *auto*
qed
unfolding *adjoint-def* **apply** *auto*
done
next
case (*Differential* ϑ)
assume *IH*:$\bigwedge \nu\ \omega$. *Vagree* ν ω ($\bigcup i{\in}SIGT\ \vartheta$. *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a* | *None* $\Rightarrow$ {}) $\Longrightarrow$ *dsafe* ϑ $\Longrightarrow$ *dterm-sem* (*local.adjoint I* σ ν) ϑ = *dterm-sem* (*local.adjoint I* σ ω) ϑ
assume *VA*: *Vagree* ν ω ($\bigcup i{\in}SIGT$ (*Differential* ϑ). *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a* | *None* $\Rightarrow$ {})
assume *safe*:*dsafe* (*Differential* ϑ)
then have *free*:*dfree* ϑ **by** (*auto dest*: *dsafe.cases*)
from *VA* **have** *VA'*:*Vagree* ν ω ($\bigcup i{\in}SIGT\ \vartheta$. *case SFunctions* σ *i of Some a* $\Rightarrow$ *FVT a* | *None* $\Rightarrow$ {})
by *auto*
have *dsafe*:$\bigwedge f\, f'$. *SFunctions* σ *f* = *Some f'* $\Longrightarrow$ *dsafe f'*
using *dfree dfree-is-dsafe* **by** *auto*
have *sem*:*sterm-sem* (*local.adjoint I* σ ν) ϑ = *sterm-sem* (*local.adjoint I* σ ω) ϑ
using *uadmit-sterm-adjoint'*[*OF dsafe fsafe VA', of* λ *x y*. *x* λ *x y*. *x I*] **by** *auto*
have *good1*:*is-interp* (*adjoint I* σ ν) **using** *adjoint-safe*[*OF good-interp dfree*] **by** *auto*
have *good2*:*is-interp* (*adjoint I* σ ω) **using** *adjoint-safe*[*OF good-interp dfree*] **by** *auto*
have *frech*:*frechet* (*local.adjoint I* σ ν) ϑ = *frechet* (*local.adjoint I* σ ω) ϑ
apply (*auto simp add*: *fun-eq-iff*)
subgoal for *a b*
using *sterm-determines-frechet* [*OF good1 good2 free free sem, of* (*a*,*b*)] **by** *auto*
done
then show *dterm-sem* (*local.adjoint I* σ ν) (*Differential* ϑ) = *dterm-sem* (*local.adjoint*

$I\ \sigma\ \omega$) (*Differential* ϑ)
by (*auto simp add*: *directional-derivative-def*)
qed (*auto*)

lemma *uadmit-dterm-adjoint*:
assumes *TUA*:*TUadmit* $\sigma\ \vartheta\ U$
assumes *VA*:*Vagree* $\nu\ \omega\ (-U)$
assumes *dfree*:$\bigwedge f\ f'$. *SFunctions* $\sigma\ f$ = *Some* $f' \Longrightarrow$ *dfree* f'
assumes *fsafe*:$\bigwedge f\ f'$. *SPredicates* $\sigma\ f$ = *Some* $f' \Longrightarrow$ *fsafe* f'
assumes *dsafe*:*dsafe* ϑ
assumes *good-interp*:*is-interp I*
shows *dterm-sem* (*adjoint I* $\sigma\ \nu$) ϑ = *dterm-sem* (*adjoint I* $\sigma\ \omega$) ϑ
proof −
have *duh*:$\bigwedge A\ B$. $A \cap B = \{\} \Longrightarrow A \subseteq -B$
by *auto*
have $\bigwedge x$. $x \in (\bigcup i{\in}SIGT\ \vartheta$. *case SFunctions* $\sigma\ i$ *of Some* $x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \Longrightarrow x \in (-U)$
using *TUA* **unfolding** *TUadmit-def* **by** *auto*
then have *sub1*:$(\bigcup i{\in}SIGT\ \vartheta$. *case SFunctions* $\sigma\ i$ *of Some* $x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \subseteq -U$
by *auto*
then have *VA'*:*Vagree* $\nu\ \omega\ (\bigcup i{\in}SIGT\ \vartheta$. *case SFunctions* $\sigma\ i$ *of Some* $x \Rightarrow FVT\ x \mid None \Rightarrow \{\})$
using *agree-sub*[*OF sub1 VA*] **by** *auto*
then show *?thesis* **using** *uadmit-dterm-adjoint'*[*OF dfree fsafe good-interp VA' dsafe*]
by *auto*
qed

lemma *uadmit-dterm-ntadjoint'*:
assumes *dfree*:$\bigwedge i$. *dsafe* $(\sigma\ i)$
assumes *good-interp*:*is-interp I*
shows $\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega\ (\bigcup\ i{\in}\{i.\ Inr\ i \in SIGT\ \vartheta\}.\ FVT\ (\sigma\ i)) \Longrightarrow$ *dsafe* $\vartheta \Longrightarrow$ *dterm-sem* (*adjointFO I* $\sigma\ \nu$) ϑ = *dterm-sem* (*adjointFO I* $\sigma\ \omega$) ϑ
proof (*induct* ϑ)
case (*Plus* $\vartheta 1\ \vartheta 2\ \nu\ \omega$)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega\ (\bigcup\ i{\in}\{i.\ Inr\ i \in SIGT\ \vartheta 1\}.\ FVT\ (\sigma\ i)) \Longrightarrow$ *dsafe* $\vartheta 1 \Longrightarrow$ *dterm-sem* (*adjointFO I* $\sigma\ \nu$) $\vartheta 1$ = *dterm-sem* (*adjointFO I* $\sigma\ \omega$) $\vartheta 1$
assume *IH2*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega\ (\bigcup\ i{\in}\{i.\ Inr\ i \in SIGT\ \vartheta 2\}.\ FVT\ (\sigma\ i)) \Longrightarrow$ *dsafe* $\vartheta 2 \Longrightarrow$ *dterm-sem* (*adjointFO I* $\sigma\ \nu$) $\vartheta 2$ = *dterm-sem* (*adjointFO I* $\sigma\ \omega$) $\vartheta 2$
assume *VA*:*Vagree* $\nu\ \omega\ (\bigcup\ i{\in}\{i.\ Inr\ i \in SIGT\ (Plus\ \vartheta 1\ \vartheta 2)\}.\ FVT\ (\sigma\ i))$
then have *VA1*:*Vagree* $\nu\ \omega\ (\bigcup\ i{\in}\{i.\ Inr\ i \in SIGT\ \vartheta 1\}.\ FVT\ (\sigma\ i))$
and *VA2*:*Vagree* $\nu\ \omega\ (\bigcup\ i{\in}\{i.\ Inr\ i \in SIGT\ \vartheta 2\}.\ FVT\ (\sigma\ i))$
unfolding *Vagree-def* **by** *auto*
assume *safe*:*dsafe* (*Plus* $\vartheta 1\ \vartheta 2$)
show *?case*
using *IH1*[*OF VA1*] *IH2*[*OF VA2*] *safe* **by** *auto*

next
case (*Times ϑ1 ϑ2 ν ω*)
assume *IH1*:⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ1*}. *FVT* (*σ i*)) ⟹ *dsafe ϑ1* ⟹ *dterm-sem* (*adjointFO I σ ν*) *ϑ1* = *dterm-sem* (*adjointFO I σ ω*) *ϑ1*
assume *IH2*:⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ2*}. *FVT* (*σ i*)) ⟹ *dsafe ϑ2* ⟹ *dterm-sem* (*adjointFO I σ ν*) *ϑ2* = *dterm-sem* (*adjointFO I σ ω*) *ϑ2*
assume *VA*: *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* (*Times ϑ1 ϑ2*)}. *FVT* (*σ i*))
then have *VA1*: *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ1*}. *FVT* (*σ i*))
and *VA2*: *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT ϑ2*}. *FVT* (*σ i*))
unfolding *Vagree-def* **by** *auto*
assume *safe*:*dsafe* (*Times ϑ1 ϑ2*)
show *?case*
using *IH1*[*OF VA1*] *IH2*[*OF VA2*] *safe* **by** *auto*
next
case (*Function x1a x2a*)
assume *IH*:⋀*x*. ⋀*ν ω*. *x* ∈ *range x2a* ⟹ *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT x*}. *FVT* (*σ i*)) ⟹
dsafe x ⟹ *dterm-sem* (*adjointFO I σ ν*) *x* = *dterm-sem* (*adjointFO I σ ω*) *x*
assume *safe*:*dsafe* (*Function x1a x2a*)
from *safe* **have** *safes*:⋀*j*. *dsafe* (*x2a j*) **by** *auto*
from *IH* **have** *IH′*:⋀*j*. *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* (*x2a j*)}. *FVT* (*σ i*)) ⟹
dterm-sem (*adjointFO I σ ν*) (*x2a j*) = *dterm-sem* (*adjointFO I σ ω*) (*x2a j*)
using *rangeI safes* **by** *auto*
assume *VA*: *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* (*$f x1a x2a*)}. *FVT* (*σ i*))
from *VA* **have** *VAs*:⋀*j*. *Vagree ν ω* (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* (*x2a j*)}. *FVT* (*σ i*))
unfolding *Vagree-def SIGT.simps* **using** *rangeI* **by** *blast*
have *SIGT*:*x1a* ∈ *SIGT* (*$f x1a x2a*) **by** *auto*
have *VAsub*:⋀*a*. *x1a* = *Inr a* ⟹ (*FVT* (*σ a*)) ⊆ (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* (*$f x1a x2a*)}. *FVT* (*σ i*))
using *SIGT* **by** *auto*
have *VAf*:⋀*a*. *x1a* = *Inr a* ⟹ *Vagree ν ω* (*FVT* (*σ a*))
using *agree-sub*[*OF VAsub VA*] **by** *auto*
then show *?case*
using *IH′*[*OF VAs*] **apply** (*auto simp add*: *fun-eq-iff*)
apply(*cases x1a*)
defer
subgoal for *x1 x2 a*
proof −
assume *VA*:(⋀*a*. *x1a* = *Inr a* ⟹ *Vagree ν ω* (*FVT* (*σ a*)))
assume *sems*:(⋀*j*. ∀ *x1 x2*. *dterm-sem* (*adjointFO I σ ν*) (*x2a j*) (*x1*,*x2*) = *dterm-sem* (*adjointFO I σ ω*) (*x2a j*) (*x1*,*x2*))
assume *some*:*x1a* = *Inr a*
note *FVT* = *VAf*[*OF some*]
have *dsafe*:⋀*i*. *dsafe* (*σ i*)

using *dfree dfree-is-dsafe* **by** *auto*
have *dsem*:$\bigwedge R$. *dterm-sem I* (σ *a*) ν = *dterm-sem I* (σ *a*) ω
using *coincidence-dterm*[*OF dsafe FVT*] **by** *auto*
have $\bigwedge R$. *Functions* (*adjointFO I* σ ν) *x1a R* = *Functions* (*adjointFO I* σ ω) *x1a R*
using *dsem some* **unfolding** *adjointFO-def* **by** *auto*
then show *Functions* (*adjointFO I* σ ν) *x1a* (χ *i. dterm-sem* (*adjointFO I* σ ω) (*x2a i*) (*x1*,*x2*)) =
Functions (*adjointFO I* σ ω) *x1a* (χ *i. dterm-sem* (*adjointFO I* σ ω) (*x2a i*) (*x1*,*x2*))
by *auto*
qed
unfolding *adjointFO-def* **apply** *auto*
done
next
case (*Differential* ϑ)
assume *IH*:$\bigwedge \nu\ \omega$. *Vagree* ν ω ($\bigcup$ *i*$\in${*i. Inr i* $\in$ *SIGT* ϑ}. *FVT* (σ *i*)) $\Longrightarrow$ *dsafe* ϑ $\Longrightarrow$ *dterm-sem* (*adjointFO I* σ ν) ϑ = *dterm-sem* (*adjointFO I* σ ω) ϑ
assume *VA*:*Vagree* ν ω ($\bigcup$ *i*$\in${*i. Inr i* $\in$ *SIGT* (*Differential* ϑ)}. *FVT* (σ *i*))
assume *safe*:*dsafe* (*Differential* ϑ)
then have *free*:*dfree* ϑ **by** (*auto dest*: *dsafe.cases*)
from *VA* **have** *VA'*:*Vagree* ν ω ($\bigcup$ *i*$\in${*i. Inr i* $\in$ *SIGT* ϑ}. *FVT* (σ *i*))
by *auto*
have *dsafe*:$\bigwedge i$. *dsafe* (σ *i*)
using *dfree dfree-is-dsafe* **by** *auto*
have *sem*:*sterm-sem* (*adjointFO I* σ ν) ϑ = *sterm-sem* (*adjointFO I* σ ω) ϑ
using *uadmit-sterm-ntadjoint'*[*OF dsafe VA'*] **by** *auto*
have *good1*:*is-interp* (*adjointFO I* σ ν) **using** *adjointFO-safe*[*OF good-interp dsafe*, *of* λi. *i*] **by** *auto*
have *good2*:*is-interp* (*adjointFO I* σ ω) **using** *adjointFO-safe*[*OF good-interp dsafe*, *of* λi. *i*] **by** *auto*
have *frech*:*frechet* (*adjointFO I* σ ν) ϑ = *frechet* (*adjointFO I* σ ω) ϑ
apply (*auto simp add*: *fun-eq-iff*)
subgoal for *a b*
using *sterm-determines-frechet* [*OF good1 good2 free free sem*, *of* (*a*,*b*)] **by** *auto*
done
then show *dterm-sem* (*adjointFO I* σ ν) (*Differential* ϑ) = *dterm-sem* (*adjointFO I* σ ω) (*Differential* ϑ)
by (*auto simp add*: *directional-derivative-def*)
qed (*auto*)

lemma *uadmit-dterm-ntadjoint*:
assumes *TUA*:*NTUadmit* σ ϑ *U*
assumes *VA*:*Vagree* ν ω (−*U*)
assumes *dfree*:$\bigwedge i$. *dsafe* (σ *i*)
assumes *dsafe*:*dsafe* ϑ
assumes *good-interp*:*is-interp I*
shows *dterm-sem* (*adjointFO I* σ ν) ϑ = *dterm-sem* (*adjointFO I* σ ω) ϑ

proof −
have *duh*:⋀*A B*. *A* ∩ *B* = {} ⟹ *A* ⊆ −*B*
by *auto*
have *duh*:⋀*A B*. *A* ∩ *B* = {} ⟹ *A* ⊆ −*B*
by *auto*
have ⋀*x*. *x* ∈ (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* ϑ}. *FVT* (σ *i*)) ⟹ *x* ∈ (−*U*)
using *TUA* **unfolding** *NTUadmit-def* **by** *auto*
then have *sub1*:(⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* ϑ}. *FVT* (σ *i*)) ⊆ −*U*
by *auto*
then have *VA'*:*Vagree* ν ω (⋃ *i*∈{*i*. *Inr i* ∈ *SIGT* ϑ}. *FVT* (σ *i*))
using *agree-sub*[*OF sub1 VA*] **by** *auto*
then show *?thesis* **using** *uadmit-dterm-ntadjoint'*[*OF dfree good-interp VA' dsafe*]
by *auto*
qed

definition *ssafe* ::('*sf*, '*sc*, '*sz*) *subst* ⇒ *bool*
where *ssafe* σ ≡
(∀ *i f'*. *SFunctions* σ *i* = *Some f'* ⟶ *dfree f'*) ∧
(∀ *f f'*. *SPredicates* σ *f* = *Some f'* ⟶ *fsafe f'*) ∧
(∀ *f f'*. *SPrograms* σ *f* = *Some f'* ⟶ *hpsafe f'*) ∧
(∀ *f f'*. *SODEs* σ *f* = *Some f'* ⟶ *osafe f'*) ∧
(∀ *C C'*. *SContexts* σ *C* = *Some C'* ⟶ *fsafe C'*)

lemma *uadmit-dterm-adjointS*:
assumes *ssafe*:*ssafe* σ
assumes *good-interp*:*is-interp I*
fixes ν ω
assumes *VA*:*Vagree* ν ω (⋃*i*∈*SIGT* ϑ. *case SFunctions* σ *i of Some x* ⇒ *FVT x* | *None* ⇒ {})
assumes *dsafe*:*dsafe* ϑ
shows *dterm-sem* (*adjoint I* σ ν) ϑ = *dterm-sem* (*adjoint I* σ ω) ϑ
proof −
show *?thesis*
apply(*rule uadmit-dterm-adjoint'*)
using *good-interp ssafe VA dsafe* **unfolding** *ssafe-def* **by** *auto*
qed

lemma *adj-sub-assign-fact*:⋀*i j e*. *i*∈*SIGT e* ⟹ *j* ∈ (*case SFunctions* σ *i of Some x* ⇒ *FVT x* | *None* ⇒ {}) ⟹ *Inl i* ∈({*Inl x* |*x*. *x* ∈ *dom* (*SFunctions* σ)} ∪ {*Inr* (*Inl x*) |*x*. *x* ∈ *dom* (*SContexts* σ)} ∪ {*Inr* (*Inr x*) |*x*. *x* ∈ *dom* (*SPredicates* σ)} ∪ {*Inr* (*Inr x*) |*x*. *x* ∈ *dom* (*SPrograms* σ)}) ∩ {*Inl x* |*x*. *x* ∈ *SIGT e*}
unfolding *SDom-def* **apply** *auto*
subgoal for *i j*
apply (*cases SFunctions* σ *i*)
by *auto*
done

lemma *adj-sub-geq1-fact*:$\bigwedge i\ j\ x1\ x2.\ i{\in}SIGT\ x1 \Longrightarrow j \in (case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \Longrightarrow Inl\ i \in(\{Inl\ x\ |x.\ x \in dom\ (SFunctions\ \sigma)\} \cup \{Inr\ (Inl\ x)\ |x.\ x \in dom\ (SContexts\ \sigma)\} \cup \{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPredicates\ \sigma)\} \cup$
$\{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPrograms\ \sigma)\}) \cap$
$\{Inl\ x\ |x.\ x \in SIGT\ x1 \vee x \in SIGT\ x2\}$
unfolding *SDom-def* **apply** *auto*
subgoal for *i j*
apply (*cases SFunctions σ i*)
by *auto*
done

lemma *adj-sub-geq2-fact*:$\bigwedge i\ j\ x1\ x2.\ i{\in}SIGT\ x2 \Longrightarrow j \in (case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \Longrightarrow Inl\ i \in(\{Inl\ x\ |x.\ x \in dom\ (SFunctions\ \sigma)\} \cup \{Inr\ (Inl\ x)\ |x.\ x \in dom\ (SContexts\ \sigma)\} \cup \{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPredicates\ \sigma)\} \cup$
$\{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPrograms\ \sigma)\}) \cap$
$\{Inl\ x\ |x.\ x \in SIGT\ x1 \vee x \in SIGT\ x2\}$
unfolding *SDom-def* **apply** *auto*
subgoal for *i j*
apply (*cases SFunctions σ i*)
by *auto*
done
lemma *adj-sub-prop-fact*:$\bigwedge i\ j\ x1\ x2\ k.\ i{\in}SIGT\ (x2\ k) \Longrightarrow j \in (case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \Longrightarrow Inl\ i \in(\{Inl\ x\ |x.\ x \in dom\ (SFunctions\ \sigma)\} \cup \{Inr\ (Inl\ x)\ |x.\ x \in dom\ (SContexts\ \sigma)\} \cup \{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPredicates\ \sigma)\} \cup$
$\{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPrograms\ \sigma)\}) \cap$
$insert\ (Inr\ (Inr\ x1))\ \{Inl\ x\ |x.\ \exists xa.\ x \in SIGT\ (x2\ xa)\}$
unfolding *SDom-def* **apply** *auto*
subgoal for *i j*
apply (*cases SFunctions σ i*)
by *auto*
done

lemma *adj-sub-ode-fact*:$\bigwedge i\ j\ x1\ x2.\ Inl\ i \in SIGO\ x1 \Longrightarrow j \in (case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \Longrightarrow Inl\ i \in(\{Inl\ x\ |x.\ x \in dom\ (SFunctions\ \sigma)\} \cup \{Inr\ (Inl\ x)\ |x.\ x \in dom\ (SContexts\ \sigma)\} \cup \{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPredicates\ \sigma)\} \cup$
$\{Inr\ (Inr\ x)\ |x.\ x \in dom\ (SPrograms\ \sigma)\}) \cap$
$(SIGF\ x2 \cup \{Inl\ x\ |x.\ Inl\ x \in SIGO\ x1\} \cup \{Inr\ (Inr\ x)\ |x.\ Inr\ x \in SIGO\ x1\})$
unfolding *SDom-def* **apply** *auto*
subgoal for *i j*
apply (*cases SFunctions σ i*)
by *auto*
done

lemma *adj-sub-assign*:$\bigwedge e\ \sigma\ x.\ (\bigcup i{\in}SIGT\ e.\ case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow$

FVT x | None ⇒ {}) ⊆ (⋃ a∈SDom σ ∩ SIGP (x := e). SFV σ a)
subgoal for *e σ x*
unfolding *SDom-def* **apply** *auto*
subgoal for *i j*
apply (*cases SFunctions σ j*)
apply *auto*
subgoal for *a*
using *adj-sub-assign-fact*[*of j e i*]
by (*metis* (*mono-tags*, *lifting*) *SFV.simps*(*1*) *option.simps*(*5*))
done
done
done

lemma *adj-sub-diff-assign*:⋀*e σ x*. (⋃ *i*∈*SIGT e*. *case SFunctions σ i of Some x*
⇒ *FVT x | None* ⇒ {}) ⊆ (⋃ *a*∈*SDom σ* ∩ *SIGP* (*DiffAssign x e*). *SFV σ a*)
subgoal for *e σ x*
unfolding *SDom-def* **apply** *auto*
subgoal for *i j*
apply (*cases SFunctions σ j*)
apply *auto*
subgoal for *a*
using *adj-sub-assign-fact*[*of j e i*]
by (*metis* (*mono-tags*, *lifting*) *SFV.simps*(*1*) *option.simps*(*5*))
done
done
done

lemma *adj-sub-geq1*:⋀*σ x1 x2*. (⋃ *i*∈*SIGT x1*. *case SFunctions σ i of Some x* ⇒
FVT x | None ⇒ {}) ⊆ (⋃ *a*∈*SDom σ* ∩ *SIGF* (*Geq x1 x2*). *SFV σ a*)
subgoal for *σ x1 x2*
unfolding *SDom-def* **apply** *auto*
subgoal for *x i*
apply (*cases SFunctions σ i*)
apply *auto*
subgoal for *a*
using *adj-sub-geq1-fact*[*of i x1 x σ*]
by (*metis* (*mono-tags*, *lifting*) *SFV.simps*(*1*) *option.simps*(*5*))
done
done
done

lemma *adj-sub-geq2*:⋀*σ x1 x2*. (⋃ *i*∈*SIGT x2*. *case SFunctions σ i of Some x* ⇒
FVT x | None ⇒ {}) ⊆ (⋃ *a*∈*SDom σ* ∩ *SIGF* (*Geq x1 x2*). *SFV σ a*)
subgoal for *σ x1 x2*
unfolding *SDom-def* **apply** *auto*
subgoal for *x i*
apply (*cases SFunctions σ i*)
apply *auto*
subgoal for *a*

using *adj-sub-geq2-fact*[*of i x2 x σ*]
by (*metis* (*mono-tags*, *lifting*) *SFV.simps*(1) *option.simps*(5))
done
done
done

lemma *adj-sub-prop*:⋀*σ x1 x2 j* . (⋃ *i*∈*SIGT* (*x2 j*). *case SFunctions σ i of Some x* ⇒ *FVT x* | *None* ⇒ {}) ⊆ (⋃ *a*∈*SDom σ* ∩ *SIGF* (*$φ x1 x2*). *SFV σ a*)
subgoal for *σ x1 x2 j*
unfolding *SDom-def* **apply** *auto*
subgoal for *x i*
apply (*cases SFunctions σ i*)
apply *auto*
subgoal for *a*
using *adj-sub-prop-fact*[*of i x2 j x σ x1*]
by (*metis* (*mono-tags*, *lifting*) *SFV.simps*(1) *option.simps*(5))
done
done
done

lemma *adj-sub-ode*:⋀*σ x1 x2*. (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO x1*}. *case SFunctions σ i of None* ⇒ {} | *Some x* ⇒ *FVT x*) ⊆ (⋃ *a*∈*SDom σ* ∩ *SIGP* (*EvolveODE x1 x2*). *SFV σ a*)
subgoal for *σ x1 x2*
unfolding *SDom-def* **apply** *auto*
subgoal for *x i*
apply (*cases SFunctions σ i*)
apply *auto*
subgoal for *a*
using *adj-sub-ode-fact*[*of i x1 x σ x2*]
by (*metis* (*mono-tags*, *lifting*) *SFV.simps*(1) *option.simps*(5))
done
done
done

lemma *uadmit-ode-adjoint′*:
fixes *σ I*
assumes *ssafe*:*ssafe σ*
assumes *good-interp*:*is-interp I*
shows⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO ODE*}. *case SFunctions σ i of None* ⇒ {} | *Some x* ⇒ *FVT x*)⟹ *osafe ODE* ⟹ *ODE-sem* (*adjoint I σ ν*) *ODE* = *ODE-sem* (*adjoint I σ ω*) *ODE*
proof (*induction ODE*)
case (*OVar x*)
then show *?case* **unfolding** *adjoint-def* **by** *auto*
next
case (*OSing x1a x2*)
assume *VA*:*Vagree ν ω* (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO* (*OSing x1a x2*)}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*)

assume *osafe*:*osafe* (*OSing x1a x2*)
then have *dfree*:*dfree x2* **by** (*auto dest*: *osafe.cases*)
have *safes*:(⋀*f f'*. *SFunctions σ f* = *Some f'* ⟹ *dsafe f'*)
(⋀*f f'*. *SPredicates σ f* = *Some f'* ⟹ *fsafe f'*)
using *ssafe* **unfolding** *ssafe-def* **using** *dfree-is-dsafe* **by** *auto*
have *sem*:*sterm-sem* (*local.adjoint I σ ν*) *x2* = *sterm-sem* (*local.adjoint I σ ω*) *x2*
using *uadmit-sterm-adjoint'*[*of σ ν ω x2 I*, *OF safes*, *of* (λ *x y*. *x*) (λ *x y*. *x*)] *VA*
by *auto*
show *?case*
apply *auto*
apply (*rule ext*)
subgoal for *x*
apply (*rule vec-extensionality*)
using *sem* **by** *auto*
done
next
case (*OProd ODE1 ODE2*)
assume *IH1*:⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO ODE1*}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*) ⟹
osafe ODE1 ⟹ *ODE-sem* (*local.adjoint I σ ν*) *ODE1* = *ODE-sem* (*local.adjoint I σ ω*) *ODE1*
assume *IH2*:⋀*ν ω*. *Vagree ν ω* (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO ODE2*}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*) ⟹
osafe ODE2 ⟹ *ODE-sem* (*local.adjoint I σ ν*) *ODE2* = *ODE-sem* (*local.adjoint I σ ω*) *ODE2*
assume *VA*: *Vagree ν ω* (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO* (*OProd ODE1 ODE2*)}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*)
assume *safe*:*osafe* (*OProd ODE1 ODE2*)
from *safe* **have** *safe1*:*osafe ODE1* **and** *safe2*:*osafe ODE2* **by** (*auto dest*: *osafe.cases*)
have *sub1*:(⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO ODE1*}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*) ⊆ (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO* (*OProd ODE1 ODE2*)}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*)
by *auto*
have *sub2*:(⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO ODE2*}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*) ⊆ (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO* (*OProd ODE1 ODE2*)}. *case SFunctions σ i of None* ⇒ {} | *Some a* ⇒ *FVT a*)
by *auto*
then show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
qed

lemma *uadmit-ode-ntadjoint'*:
fixes *σ I*
assumes *ssafe*:⋀*i*. *dsafe* (*σ i*)
assumes *good-interp*:*is-interp I*
shows⋀*ν ω*. *Vagree ν ω* (⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO ODE*}. *FVT* (*σ y*)) ⟹

osafe ODE $\Longrightarrow$ *ODE-sem* (*adjointFO I* σ ν) *ODE* = *ODE-sem* (*adjointFO I* σ ω) *ODE*
proof (*induction ODE*)
case (*OVar x*)
then show *?case* **unfolding** *adjointFO-def* **by** *auto*
next
case (*OSing x1a x2*)
assume *VA*:*Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO* (*OSing x1a x2*)}. *FVT* (σ *y*))
assume *osafe*:*osafe* (*OSing x1a x2*)
then have *dfree*:*dfree x2* **by** (*auto dest*: *osafe.cases*)
have *sem*:*sterm-sem* (*adjointFO I* σ ν) *x2* = *sterm-sem* (*adjointFO I* σ ω) *x2*
using *uadmit-sterm-ntadjoint'*[*of* σ ν ω *x2 I*, *OF ssafe*] *VA*
by *auto*
show *?case*
apply *auto*
apply (*rule ext*)
subgoal for *x*
apply (*rule vec-extensionality*)
using *sem* **by** *auto*
done
next
case (*OProd ODE1 ODE2*)
assume *IH1*:$\bigwedge\nu$ ω. *Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO ODE1*}. *FVT* (σ *y*)) $\Longrightarrow$
osafe ODE1 $\Longrightarrow$ *ODE-sem* (*adjointFO I* σ ν) *ODE1* = *ODE-sem* (*adjointFO I* σ ω) *ODE1*
assume *IH2*:$\bigwedge\nu$ ω. *Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO ODE2*}. *FVT* (σ *y*)) $\Longrightarrow$
osafe ODE2 $\Longrightarrow$ *ODE-sem* (*adjointFO I* σ ν) *ODE2* = *ODE-sem* (*adjointFO I* σ ω) *ODE2*
assume *VA*:*Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO* (*OProd ODE1 ODE2*)}. *FVT* (σ *y*))
assume *safe*:*osafe* (*OProd ODE1 ODE2*)
from *safe* **have** *safe1*:*osafe ODE1* **and** *safe2*:*osafe ODE2* **by** (*auto dest*: *osafe.cases*)
have *sub1*:($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO ODE1*}. *FVT* (σ *y*)) ⊆ ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO* (*OProd ODE1 ODE2*)}. *FVT* (σ *y*))
by *auto*
have *sub2*:($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO ODE2*}. *FVT* (σ *y*)) ⊆ ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO* (*OProd ODE1 ODE2*)}. *FVT* (σ *y*))
by *auto*
then show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
qed

lemma *adjoint-ode-vars*:
shows *ODE-vars* (*local.adjoint I* σ ν) *ODE* = *ODE-vars* (*local.adjoint I* σ ω) *ODE*

apply(*induction ODE*)
unfolding *adjoint-def* **by** *auto*

lemma *uadmit-mkv-adjoint*:
assumes *ssafe*:*ssafe* σ
assumes *good-interp*:*is-interp I*
assumes *VA*:*Vagree* ν ω ($\bigcup i \in \{i \mid i.\ (Inl\ i \in SIGO\ ODE)\}$. *case SFunctions* σ *i of Some x* $\Rightarrow$ *FVT x* | *None* $\Rightarrow$ $\{\}$)
assumes *osafe*:*osafe ODE*
shows *mk-v* (*adjoint I* σ ν) *ODE* = *mk-v* (*adjoint I* σ ω) *ODE*
apply(*rule ext*)
subgoal for *R*
apply(*rule ext*)
subgoal for *solt*
apply(*rule agree-UNIV-eq*)
using *mk-v-agree*[*of* (*adjoint I* σ ν) *ODE R solt*]
using *mk-v-agree*[*of* (*adjoint I* σ ω) *ODE R solt*]
using *uadmit-ode-adjoint'*[*OF ssafe good-interp VA osafe*]
unfolding *Vagree-def*
apply *auto*
subgoal for *i*
apply (*cases Inl i* $\in$ *Inl* ‘ *ODE-vars* (*adjoint I* σ ω) *ODE*)
proof −
assume *sem-eq*:*ODE-sem* (*local.adjoint I* σ ν) *ODE* = *ODE-sem* (*local.adjoint I* σ ω) *ODE*
have *vars-eq*:*ODE-vars* (*local.adjoint I* σ ν) *ODE* = *ODE-vars* (*local.adjoint I* σ ω) *ODE*
apply(*induction ODE*)
unfolding *adjoint-def* **by** *auto*
assume *thing1*:
$\forall i.$ (*Inl i* $\in$ *Inl* ‘ *ODE-vars* (*local.adjoint I* σ ν) *ODE* $\longrightarrow$ *fst* (*mk-v* (*local.adjoint I* σ ν) *ODE R solt*) $ *i* = *solt* $ *i*) $\wedge$
(*Inl i* $\in$ *Inr* ‘ *ODE-vars* (*local.adjoint I* σ ν) *ODE* $\longrightarrow$ *fst* (*mk-v* (*local.adjoint I* σ ν) *ODE R solt*) $ *i* = *solt* $ *i*)
assume *thing2*:
$\forall i.$ (*Inl i* $\in$ *Inl* ‘ *ODE-vars* (*local.adjoint I* σ ω) *ODE* $\longrightarrow$ *fst* (*mk-v* (*local.adjoint I* σ ω) *ODE R solt*) $ *i* = *solt* $ *i*) $\wedge$
(*Inl i* $\in$ *Inr* ‘ *ODE-vars* (*local.adjoint I* σ ω) *ODE* $\longrightarrow$ *fst* (*mk-v* (*local.adjoint I* σ ω) *ODE R solt*) $ *i* = *solt* $ *i*)
assume *inl*:*Inl i* $\in$ *Inl* ‘ *ODE-vars* (*local.adjoint I* σ ω) *ODE*
from *thing1* **and** *inl* **have** *eq1*: *fst* (*mk-v* (*local.adjoint I* σ ν) *ODE R solt*) $ *i* = *solt* $ *i*
using *vars-eq* **by** *auto*
from *thing2* **and** *inl* **have** *eq2*: *fst* (*mk-v* (*local.adjoint I* σ ω) *ODE R solt*) $ *i* = *solt* $ *i*
using *vars-eq* **by** *auto*
show *fst* (*mk-v* (*local.adjoint I* σ ν) *ODE R solt*) $ *i* = *fst* (*mk-v* (*local.adjoint I* σ ω) *ODE R solt*) $ *i*
using *eq1 eq2* **by** *auto*

next
assume *sem-eq*:*ODE-sem* (*local.adjoint I σ ν*) *ODE* = *ODE-sem* (*local.adjoint I σ ω*) *ODE*
assume *thing1*:∀ *i*. *Inl i* ∉ *Inl* ‘ *ODE-vars* (*local.adjoint I σ ν*) *ODE* ∧ *Inl i* ∉ *Inr* ‘ *ODE-vars* (*local.adjoint I σ ν*) *ODE* ⟶ *fst* (*mk-v* (*local.adjoint I σ ν*) *ODE R solt*) $ *i* = *fst R* $ *i*
assume *thing2*:∀ *i*. *Inl i* ∉ *Inl* ‘ *ODE-vars* (*local.adjoint I σ ω*) *ODE* ∧ *Inl i* ∉ *Inr* ‘ *ODE-vars* (*local.adjoint I σ ω*) *ODE* ⟶ *fst* (*mk-v* (*local.adjoint I σ ω*) *ODE R solt*) $ *i* = *fst R* $ *i*
assume *inl*:*Inl i* ∉ *Inl* ‘ *ODE-vars* (*local.adjoint I σ ω*) *ODE*
have *vars-eq*:*ODE-vars* (*local.adjoint I σ ν*) *ODE* = *ODE-vars* (*local.adjoint I σ ω*) *ODE*
apply(*induction ODE*)
unfolding *adjoint-def* **by** *auto*
from *thing1* **and** *inl* **have** *eq1*: *fst* (*mk-v* (*local.adjoint I σ ν*) *ODE R solt*) $ *i* = *fst R* $ *i*
using *vars-eq* **by** *auto*
from *thing2* **and** *inl* **have** *eq2*: *fst* (*mk-v* (*local.adjoint I σ ω*) *ODE R solt*) $ *i* = *fst R* $ *i*
using *vars-eq* **by** *auto*
show *fst* (*mk-v* (*local.adjoint I σ ν*) *ODE R solt*) $ *i* = *fst* (*mk-v* (*local.adjoint I σ ω*) *ODE R solt*) $ *i*
using *eq1 eq2* **by** *auto*
qed
subgoal for *i*
apply (*cases Inr i* ∈ *Inr* ‘ *ODE-vars* (*adjoint I σ ω*) *ODE*)
proof −
assume *sem-eq*:*ODE-sem* (*local.adjoint I σ ν*) *ODE* = *ODE-sem* (*local.adjoint I σ ω*) *ODE*
assume *thing1*:∀ *i*. (*Inr i* ∈ *Inl* ‘ *ODE-vars* (*local.adjoint I σ ν*) *ODE* ⟶ *snd* (*mk-v* (*local.adjoint I σ ν*) *ODE R solt*) $ *i* = *ODE-sem* (*local.adjoint I σ ω*) *ODE solt* $ *i*) ∧ (*Inr i* ∈ *Inr* ‘ *ODE-vars* (*local.adjoint I σ ν*) *ODE* ⟶ *snd* (*mk-v* (*local.adjoint I σ ν*) *ODE R solt*) $ *i* = *ODE-sem* (*local.adjoint I σ ω*) *ODE solt* $ *i*)
assume *thing2*:∀ *i*. (*Inr i* ∈ *Inl* ‘ *ODE-vars* (*local.adjoint I σ ω*) *ODE* ⟶ *snd* (*mk-v* (*local.adjoint I σ ω*) *ODE R solt*) $ *i* = *ODE-sem* (*local.adjoint I σ ω*) *ODE solt* $ *i*) ∧ (*Inr i* ∈ *Inr* ‘ *ODE-vars* (*local.adjoint I σ ω*) *ODE* ⟶ *snd* (*mk-v* (*local.adjoint I σ ω*) *ODE R solt*) $ *i* = *ODE-sem* (*local.adjoint I σ ω*) *ODE solt* $ *i*)
assume *inr*:*Inr i* ∈ *Inr* ‘ *ODE-vars* (*local.adjoint I σ ω*) *ODE*
have *vars-eq*:*ODE-vars* (*local.adjoint I σ ν*) *ODE* = *ODE-vars* (*local.adjoint I σ ω*) *ODE*
apply(*induction ODE*)
unfolding *adjoint-def* **by** *auto*
show *snd* (*mk-v* (*local.adjoint I σ ν*) *ODE R solt*) $ *i* = *snd* (*mk-v* (*local.adjoint I σ ω*) *ODE R solt*) $ *i*
using *thing1 thing2 vars-eq inr* **by** *auto*

```
      next
      assume sem-eq:ODE-sem (local.adjoint I σ ν) ODE = ODE-sem (local.adjoint
I σ ω) ODE
       assume thing1:∀ i. Inr i ∉ Inl ‘ ODE-vars (local.adjoint I σ ν) ODE ∧ Inr
i ∉ Inr ‘ ODE-vars (local.adjoint I σ ν) ODE ⟶
           snd (mk-v (local.adjoint I σ ν) ODE R solt) $ i = snd R $ i
         assume thing2:∀ i. Inr i ∉ Inl ‘ ODE-vars (local.adjoint I σ ω) ODE ∧
Inr i ∉ Inr ‘ ODE-vars (local.adjoint I σ ω) ODE ⟶
           snd (mk-v (local.adjoint I σ ω) ODE R solt) $ i = snd R $ i
         assume inr:Inr i ∉ Inr ‘ ODE-vars (local.adjoint I σ ω) ODE
       have vars-eq:ODE-vars (local.adjoint I σ ν) ODE = ODE-vars (local.adjoint
I σ ω) ODE
         apply(induction ODE)
           unfolding adjoint-def by auto
        have eq1:snd (mk-v (local.adjoint I σ ν) ODE R solt) $ i = snd R $ i
          using thing1 sem-eq vars-eq inr by auto
        have eq2:snd (mk-v (local.adjoint I σ ω) ODE R solt) $ i = snd R $ i
          using thing2 sem-eq vars-eq inr by auto
           show snd (mk-v (local.adjoint I σ ν) ODE R solt) $ i = snd (mk-v
(local.adjoint I σ ω) ODE R solt) $ i
          using eq1 eq2 by auto
      qed
     done
   done
  done

lemma adjointFO-ode-vars:
  shows ODE-vars (adjointFO I σ ν) ODE = ODE-vars (adjointFO I σ ω) ODE
  apply(induction ODE)
    unfolding adjointFO-def by auto

lemma uadmit-mkv-ntadjoint:
  assumes ssafe:⋀i. dsafe (σ i)
  assumes good-interp:is-interp I
  assumes VA:Vagree ν ω (⋃ y∈{y. Inl (Inr y) ∈ SIGO ODE}. FVT (σ y))
  assumes osafe:osafe ODE
  shows mk-v (adjointFO I σ ν) ODE = mk-v (adjointFO I σ ω) ODE
  apply(rule ext)
  subgoal for R
    apply(rule ext)
    subgoal for solt
      apply(rule agree-UNIV-eq)
      using mk-v-agree[of (adjointFO I σ ν) ODE R solt]
      using mk-v-agree[of (adjointFO I σ ω) ODE R solt]
      using uadmit-ode-ntadjoint'[OF ssafe good-interp VA osafe]
      unfolding Vagree-def
      apply auto
      using adjointFO-ode-vars by metis+
    done
```

done

lemma *uadmit-prog-fml-adjoint'*:
fixes σ I
assumes *ssafe*:*ssafe* σ
assumes *good-interp*:*is-interp* I
shows $\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ $(\bigcup x{\in}SDom\ \sigma \cap SIGP\ \alpha.\ SFV\ \sigma\ x) \Longrightarrow$ *hpsafe* $\alpha \Longrightarrow$ *prog-sem* (*adjoint* $I\ \sigma\ \nu$) α = *prog-sem* (*adjoint* $I\ \sigma\ \omega$) α
and $\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ $(\bigcup x{\in}SDom\ \sigma \cap SIGF\ \varphi.\ SFV\ \sigma\ x) \Longrightarrow$ *fsafe* $\varphi \Longrightarrow$ *fml-sem* (*adjoint* $I\ \sigma\ \nu$) φ = *fml-sem* (*adjoint* $I\ \sigma\ \omega$) φ
proof (*induct* α **and** φ)
case (*Pvar* x)
then show *?case* **unfolding** *adjoint-def* **by** *auto*
next
case (*Assign* $x\ e$)
assume *VA*:*Vagree* $\nu\ \omega$ $(\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x := e).\ SFV\ \sigma\ a)$
assume *safe*:*hpsafe* $(x := e)$
from *safe* **have** *dsafe*:*dsafe* e **by** (*auto dest*: *hpsafe.cases*)
have *sub*:$(\bigcup i{\in}SIGT\ e.\ case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\})$ $\subseteq$ $(\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x := e).\ SFV\ \sigma\ a)$
using *adj-sub-assign*[*of* $\sigma\ e\ x$] **by** *auto*
have *dterm-sem* (*local.adjoint* $I\ \sigma\ \nu$) e = *dterm-sem* (*local.adjoint* $I\ \sigma\ \omega$) e
by (*rule uadmit-dterm-adjointS*[*OF ssafe good-interp agree-sub*[*OF sub VA*] *dsafe*])
then show *?case* **by** (*auto simp add*: *vec-eq-iff*)
next
case (*DiffAssign* $x\ e$)
assume *VA*:*Vagree* $\nu\ \omega$ $(\bigcup a{\in}SDom\ \sigma \cap SIGP\ (DiffAssign\ x\ e).\ SFV\ \sigma\ a)$
assume *safe*:*hpsafe* (*DiffAssign* $x\ e$)
from *safe* **have** *dsafe*:*dsafe* e **by** (*auto dest*: *hpsafe.cases*)
have *sub*:$(\bigcup i{\in}SIGT\ e.\ case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\})$ $\subseteq$ $(\bigcup a{\in}SDom\ \sigma \cap SIGP\ (DiffAssign\ x\ e).\ SFV\ \sigma\ a)$
using *adj-sub-diff-assign*[*of* $\sigma\ e$] **by** *auto*
have *dterm-sem* (*local.adjoint* $I\ \sigma\ \nu$) e = *dterm-sem* (*local.adjoint* $I\ \sigma\ \omega$) e
by (*rule uadmit-dterm-adjointS*[*OF ssafe good-interp agree-sub*[*OF sub VA*] *dsafe*])
then show *?case* **by** (*auto simp add*: *vec-eq-iff*)
next
case (*Test* x)
assume *IH*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ $(\bigcup a{\in}SDom\ \sigma \cap SIGF\ x.\ SFV\ \sigma\ a) \Longrightarrow$ *fsafe* $x \Longrightarrow$ *fml-sem* (*adjoint* $I\ \sigma\ \nu$) x = *fml-sem* (*adjoint* $I\ \sigma\ \omega$) x
assume *VA*:*Vagree* $\nu\ \omega$ $(\bigcup a{\in}SDom\ \sigma \cap SIGP\ (?\ x).\ SFV\ \sigma\ a)$
assume *hpsafe*:*hpsafe* $(?\ x)$
then have *fsafe*:*fsafe* x **by** (*auto dest*: *hpsafe.cases*)
have *sub*:$(\bigcup a{\in}SDom\ \sigma \cap SIGF\ x.\ SFV\ \sigma\ a) \subseteq (\bigcup a{\in}SDom\ \sigma \cap SIGP\ (?\ x).\ SFV\ \sigma\ a)$
by *auto*
have *fml-sem* (*adjoint* $I\ \sigma\ \nu$) x = *fml-sem* (*adjoint* $I\ \sigma\ \omega$) x
using *IH*[*OF agree-sub*[*OF sub VA*] *fsafe*] **by** *auto*

```
  then show ?case by auto
next
  case (EvolveODE x1 x2)
  assume IH:⋀ν ω. Vagree ν ω (⋃a∈SDom σ ∩ SIGF x2. SFV σ a) ⟹ fsafe
x2 ⟹ fml-sem (local.adjoint I σ ν) x2 = fml-sem (local.adjoint I σ ω) x2
  assume VA:Vagree ν ω (⋃a∈SDom σ ∩ SIGP (EvolveODE x1 x2). SFV σ a)
  assume safe:hpsafe (EvolveODE x1 x2)
  then have osafe:osafe x1 and fsafe:fsafe x2 by (auto dest: hpsafe.cases)
   have sub1:(⋃a∈SDom σ ∩ SIGF x2. SFV σ a) ⊆ (⋃a∈SDom σ ∩ SIGP
(EvolveODE x1 x2). SFV σ a)
    by auto
  then have VAF:Vagree ν ω (⋃a∈SDom σ ∩ SIGF x2. SFV σ a)
    using agree-sub[OF sub1 VA] by auto
  note IH' = IH[OF VAF fsafe]
  have sub:(⋃i∈{i |i. Inl i ∈ SIGO x1}. case SFunctions σ i of None ⇒ {} | Some
x ⇒ FVT x) ⊆ (⋃a∈SDom σ ∩ SIGP (EvolveODE x1 x2). SFV σ a)
    using adj-sub-ode[of σ x1 x2] by auto
  moreover have IH2:ODE-sem (local.adjoint I σ ν) x1 = ODE-sem (local.adjoint
I σ ω) x1
    apply (rule uadmit-ode-adjoint')
      subgoal by (rule ssafe)
     subgoal by (rule good-interp)
    subgoal using agree-sub[OF sub VA] by auto
    subgoal by (rule osafe)
    done
  have mkv:mk-v (adjoint I σ ν) x1 = mk-v (adjoint I σ ω) x1
    apply (rule uadmit-mkv-adjoint)
      using ssafe good-interp osafe agree-sub[OF sub VA] by auto
  show ?case
    apply auto
     subgoal for aa ba bb sol t
       apply(rule exI[where x = sol])
       apply(rule conjI)
        subgoal by auto
       apply(rule exI[where x=t])
       apply(rule conjI)
        subgoal using mkv by auto
       apply(rule conjI)
        subgoal by auto using IH2 mkv IH' by auto
    subgoal for aa ba bb sol t
      apply(rule exI[where x = sol])
      apply(rule conjI)
       subgoal by auto
      apply(rule exI[where x=t])
      apply(rule conjI)
       subgoal using mkv by auto
      apply(rule conjI)
       subgoal by auto using IH2 mkv IH' by auto
    done
```

next
case (*Choice x1 x2*)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x1.\ SFV\ \sigma\ a$) $\Longrightarrow$ *hpsafe x1* $\Longrightarrow$ *prog-sem* (*local.adjoint I* $\sigma\ \nu$) *x1* = *prog-sem* (*local.adjoint I* $\sigma\ \omega$) *x1*
assume *IH2*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x2.\ SFV\ \sigma\ a$) $\Longrightarrow$ *hpsafe x2* $\Longrightarrow$ *prog-sem* (*local.adjoint I* $\sigma\ \nu$) *x2* = *prog-sem* (*local.adjoint I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x1 \cup\cup x2).\ SFV\ \sigma\ a$)
assume *safe*:*hpsafe* (*x1* $\cup\cup$ *x2*)
from *safe* **have**
safe1:*hpsafe x1*
and *safe2*:*hpsafe x2*
by (*auto dest*: *hpsafe.cases*)
have *sub1*:($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x1.\ SFV\ \sigma\ a$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x1 \cup\cup x2).\ SFV\ \sigma\ a$)
by *auto*
have *sub2*:($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x2.\ SFV\ \sigma\ a$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x1 \cup\cup x2).\ SFV\ \sigma\ a$)
by *auto*
then show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
next
case (*Sequence x1 x2*)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x1.\ SFV\ \sigma\ a$) $\Longrightarrow$ *hpsafe x1* $\Longrightarrow$ *prog-sem* (*local.adjoint I* $\sigma\ \nu$) *x1* = *prog-sem* (*local.adjoint I* $\sigma\ \omega$) *x1*
assume *IH2*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x2.\ SFV\ \sigma\ a$) $\Longrightarrow$ *hpsafe x2* $\Longrightarrow$ *prog-sem* (*local.adjoint I* $\sigma\ \nu$) *x2* = *prog-sem* (*local.adjoint I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x1\ ;;\ x2).\ SFV\ \sigma\ a$)
assume *safe*:*hpsafe* (*x1* ;; *x2*)
from *safe* **have**
safe1:*hpsafe x1*
and *safe2*:*hpsafe x2*
by (*auto dest*: *hpsafe.cases*)
have *sub1*:($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x1.\ SFV\ \sigma\ a$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x1\ ;;\ x2).\ SFV\ \sigma\ a$)
by *auto*
have *sub2*:($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x2.\ SFV\ \sigma\ a$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x1\ ;;\ x2).\ SFV\ \sigma\ a$)
by *auto*
then show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
next
case (*Loop x*)
assume *IH*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x.\ SFV\ \sigma\ a$) $\Longrightarrow$ *hpsafe x* $\Longrightarrow$ *prog-sem* (*local.adjoint I* $\sigma\ \nu$) *x* = *prog-sem* (*local.adjoint I* $\sigma\ \omega$) *x*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x{**}).\ SFV\ \sigma\ a$)
assume *safe*:*hpsafe* (*x*$**$)
from *safe* **have**
safe:*hpsafe x*
by (*auto dest*: *hpsafe.cases*)

have *sub*:($\bigcup a{\in}SDom\ \sigma \cap SIGP\ x.\ SFV\ \sigma\ a$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGP\ (x{**}).\ SFV\ \sigma\ a$)
by *auto*
show *?case* **using** *IH*[*OF agree-sub*[*OF sub VA*] *safe*] **by** *auto*
next
case (*Geq x1 x2*)
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGF\ (Geq\ x1\ x2).\ SFV\ \sigma\ a$)
assume *safe*:*fsafe* (*Geq x1 x2*)
then have *dsafe1*:*dsafe x1* **and** *dsafe2*:*dsafe x2* **by** (*auto dest*: *fsafe.cases*)
have *sub1*:($\bigcup i{\in}SIGT\ x1.$ *case SFunctions* $\sigma\ i$ *of Some* $x \Rightarrow FVT\ x \mid None \Rightarrow \{\}$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGF\ (Geq\ x1\ x2).\ SFV\ \sigma\ a$)
using *adj-sub-geq1*[*of* σ *x1 x2*] **by** *auto*
have *sub2*:($\bigcup i{\in}SIGT\ x2.$ *case SFunctions* $\sigma\ i$ *of Some* $x \Rightarrow FVT\ x \mid None \Rightarrow \{\}$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGF\ (Geq\ x1\ x2).\ SFV\ \sigma\ a$)
using *adj-sub-geq2*[*of* σ *x2 x1*] **by** *auto*
have *dterm-sem* (*local.adjoint I* $\sigma\ \nu$) *x1* = *dterm-sem* (*local.adjoint I* $\sigma\ \omega$) *x1*
by (*rule uadmit-dterm-adjointS*[*OF ssafe good-interp agree-sub*[*OF sub1 VA*] *dsafe1*])
moreover have *dterm-sem* (*local.adjoint I* $\sigma\ \nu$) *x2* = *dterm-sem* (*local.adjoint I* $\sigma\ \omega$) *x2*
by (*rule uadmit-dterm-adjointS*[*OF ssafe good-interp agree-sub*[*OF sub2 VA*] *dsafe2*])
ultimately show *?case* **by** *auto*
next
case (*Prop x1 x2* $\nu\ \omega$)
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup a{\in}SDom\ \sigma \cap SIGF\ (\$\varphi\ x1\ x2).\ SFV\ \sigma\ a$)
assume *safe*:*fsafe* ($\$\varphi$ *x1 x2*)
then have *safes*:$\bigwedge i.$ *dsafe* (*x2 i*) **using** *dfree-is-dsafe* **by** *auto*
have *subs*:$\bigwedge j.$ ($\bigcup i{\in}SIGT\ (x2\ j).$ *case SFunctions* $\sigma\ i$ *of Some* $x \Rightarrow FVT\ x \mid None \Rightarrow \{\}$) $\subseteq$ ($\bigcup a{\in}SDom\ \sigma \cap SIGF\ (\$\varphi\ x1\ x2).\ SFV\ \sigma\ a$)
subgoal for *j* **using** *adj-sub-prop*[*of* σ *x2 j x1*] **by** *auto*
done
have $\bigwedge i.$ *dterm-sem* (*local.adjoint I* $\sigma\ \nu$) (*x2 i*) = *dterm-sem* (*local.adjoint I* $\sigma\ \omega$) (*x2 i*)
by (*rule uadmit-dterm-adjointS*[*OF ssafe good-interp agree-sub*[*OF subs VA*] *safes*])
then have *vec-eq*:$\bigwedge R.$ ($\chi\ i.$ *dterm-sem* (*local.adjoint I* $\sigma\ \nu$) (*x2 i*) *R*) = ($\chi\ i.$ *dterm-sem* (*local.adjoint I* $\sigma\ \omega$) (*x2 i*) *R*)
by (*auto simp add*: *vec-eq-iff*)
from *VA* **have** *VAs*:$\bigwedge j.$ *Vagree* $\nu\ \omega$ ($\bigcup i{\in}SIGT\ (x2\ j).$ *case SFunctions* $\sigma\ i$ *of Some* $a \Rightarrow FVT\ a \mid None \Rightarrow \{\}$)
unfolding *Vagree-def SIGT.simps* **using** *rangeI*
by (*metis* (*no-types*, *lifting*) *subsetD subs*)
have *SIGF*:$\bigwedge a.$ *SPredicates* σ *x1* = *Some a* $\Longrightarrow$ *Inr* (*Inr x1*) $\in$ *SDom* $\sigma \cap$ *SIGF* ($\$\varphi$ *x1 x2*) **unfolding** *SDom-def*
by *auto*
have *VAsub*:$\bigwedge a.$ *SPredicates* σ *x1* = *Some a* $\Longrightarrow$ (*FVF a*) $\subseteq$ ($\bigcup i{\in}SDom\ \sigma \cap SIGF\ (\$\varphi\ x1\ x2).\ SFV\ \sigma\ i$)
using *SIGF* **by** *auto*

have *VAf*:$\bigwedge a$. *SPredicates* σ *x1* = *Some a* $\Longrightarrow$ *Vagree* ν ω (*FVF a*)
using *agree-sub*[*OF VAsub VA*] **by** *auto*
then show *?case*
apply(*cases SPredicates* σ *x1*)
defer
subgoal for *a*
proof −
assume *some*:*SPredicates* σ *x1* = *Some a*
note *FVF* = *VAf*[*OF some*]
have *dsafe*:$\bigwedge f\ f'$. *SFunctions* σ *f* = *Some f'* $\Longrightarrow$ *dsafe f'*
using *ssafe dfree-is-dsafe* **unfolding** *ssafe-def* **by** *auto*
have *dsem*:$\bigwedge R$. ($\nu \in$ *fml-sem* (*extendf I R*) *a*) = ($\omega \in$ *fml-sem* (*extendf I R*) *a*)
subgoal for *R*
apply (*rule coincidence-formula*)
subgoal using *ssafe* **unfolding** *ssafe-def* **using** *some* **by** *auto*
subgoal unfolding *Iagree-def* **by** *auto*
subgoal by (*rule FVF*)
done
done
have *pred-eq*:$\bigwedge R$. *Predicates* (*local.adjoint I* σ ν) *x1 R* = *Predicates* (*local.adjoint I* σ ω) *x1 R*
using *dsem some* **unfolding** *adjoint-def* **by** *auto*
show *fml-sem* (*local.adjoint I* σ ν) ($\$\varphi$ *x1 x2*) = *fml-sem* (*local.adjoint I* σ ω) ($\$\varphi$ *x1 x2*)
apply *auto*
subgoal for *a b* **using** *pred-eq*[*of* (χ *i*. *dterm-sem* (*local.adjoint I* σ ν) (*x2 i*) (*a*, *b*))] *vec-eq* **by** *auto*
subgoal for *a b* **using** *pred-eq*[*of* (χ *i*. *dterm-sem* (*local.adjoint I* σ ν) (*x2 i*) (*a*, *b*))] *vec-eq* **by** *auto*
done
qed
unfolding *adjoint-def* **using** *local.adjoint-def local.vec-eq* **apply** *auto*
done
next
case (*Not x*)
assume *IH*:$\bigwedge \nu\ \omega$. *Vagree* ν ω ($\bigcup a \in$ *SDom* $\sigma \cap$ *SIGF x*. *SFV* σ *a*) $\Longrightarrow$ *fsafe x* $\Longrightarrow$ *fml-sem* (*local.adjoint I* σ ν) *x* = *fml-sem* (*local.adjoint I* σ ω) *x*
assume *VA*:*Vagree* ν ω ($\bigcup a \in$ *SDom* $\sigma \cap$ *SIGF* (*Not x*). *SFV* σ *a*)
assume *safe*:*fsafe* (*Not x*)
from *safe* **have**
safe:*fsafe x*
by (*auto dest*: *fsafe.cases*)
have *sub*:($\bigcup a \in$ *SDom* $\sigma \cap$ *SIGF x*. *SFV* σ *a*) $\subseteq$ ($\bigcup a \in$ *SDom* $\sigma \cap$ *SIGF* (*Not x*). *SFV* σ *a*)
by *auto*
show *?case* **using** *IH*[*OF agree-sub*[*OF sub VA*] *safe*] **by** *auto*
next
case (*And x1 x2*)

assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x1*. *SFV* σ *a*) $\Longrightarrow$ *fsafe x1* $\Longrightarrow$ *fml-sem* (*local.adjoint I* $\sigma\ \nu$) *x1* = *fml-sem* (*local.adjoint I* $\sigma\ \omega$) *x1*
assume *IH2*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x2*. *SFV* σ *a*) $\Longrightarrow$ *fsafe x2* $\Longrightarrow$ *fml-sem* (*local.adjoint I* $\sigma\ \nu$) *x2* = *fml-sem* (*local.adjoint I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*And x1 x2*). *SFV* σ *a*)
assume *safe*:*fsafe* (*And x1 x2*)
from *safe* **have**
safe1:*fsafe x1*
and *safe2*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub1*:($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x1*. *SFV* σ *a*) $\subseteq$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*And x1 x2*). *SFV* σ *a*)
by *auto*
have *sub2*:($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x2*. *SFV* σ *a*) $\subseteq$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*And x1 x2*). *SFV* σ *a*)
by *auto*
show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
next
case (*Exists x1 x2*)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x2*. *SFV* σ *a*) $\Longrightarrow$ *fsafe x2* $\Longrightarrow$ *fml-sem* (*local.adjoint I* $\sigma\ \nu$) *x2* = *fml-sem* (*local.adjoint I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*Exists x1 x2*). *SFV* σ *a*)
assume *safe*:*fsafe* (*Exists x1 x2*)
from *safe* **have** *safe1*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub1*:($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x2*. *SFV* σ *a*) $\subseteq$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*Exists x1 x2*). *SFV* σ *a*)
by *auto*
show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] **by** *auto*
next
case (*Diamond x1 x2*)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGP x1*. *SFV* σ *a*) $\Longrightarrow$ *hpsafe x1* $\Longrightarrow$ *prog-sem* (*local.adjoint I* $\sigma\ \nu$) *x1* = *prog-sem* (*local.adjoint I* $\sigma\ \omega$) *x1*
assume *IH2*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x2*. *SFV* σ *a*) $\Longrightarrow$ *fsafe x2* $\Longrightarrow$ *fml-sem* (*local.adjoint I* $\sigma\ \nu$) *x2* = *fml-sem* (*local.adjoint I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*Diamond x1 x2*). *SFV* σ *a*)
assume *safe*:*fsafe* (*Diamond x1 x2*)
from *safe* **have**
safe1:*hpsafe x1*
and *safe2*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub1*:($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGP x1*. *SFV* σ *a*) $\subseteq$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*Diamond x1 x2*). *SFV* σ *a*)
by *auto*
have *sub2*:($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF x2*. *SFV* σ *a*) $\subseteq$ ($\bigcup$ *a*∈*SDom* $\sigma \cap$ *SIGF* (*Diamond x1 x2*). *SFV* σ *a*)
by *auto*
show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF*

sub2 VA] *safe2*] **by** *auto*
next
case (*InContext x1 x2*)
assume *IH1*:⋀*ν ω*. *Vagree ν ω* (⋃ *a*∈*SDom σ* ∩ *SIGF x2*. *SFV σ a*) ⟹ *fsafe x2* ⟹ *fml-sem* (*local.adjoint I σ ν*) *x2* = *fml-sem* (*local.adjoint I σ ω*) *x2*
assume *VA*: *Vagree ν ω* (⋃ *a*∈*SDom σ* ∩ *SIGF* (*InContext x1 x2*). *SFV σ a*)
assume *safe*:*fsafe* (*InContext x1 x2*)
from *safe* **have** *safe1*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub*:(⋃ *a*∈*SDom σ* ∩ *SIGF x2*. *SFV σ a*) ⊆ (⋃ *a*∈*SDom σ* ∩ *SIGF* (*InContext x1 x2*). *SFV σ a*)
by *auto*
show *?case* **using** *IH1*[*OF agree-sub*[*OF sub VA*] *safe1*]
unfolding *adjoint-def* **by** *auto*
qed

lemma *uadmit-prog-adjoint*:
assumes *PUA*:*PUadmit σ a U*
assumes *VA*: *Vagree ν ω* (−*U*)
assumes *hpsafe*:*hpsafe a*
assumes *ssafe*:*ssafe σ*
assumes *good-interp*:*is-interp I*
shows *prog-sem* (*adjoint I σ ν*) *a* = *prog-sem* (*adjoint I σ ω*) *a*
proof −
have *sub*:(⋃ *x*∈*SDom σ* ∩ *SIGP a*. *SFV σ x*) ⊆ −*U* **using** *PUA* **unfolding** *PUadmit-def* **by** *auto*
have *VA′*: *Vagree ν ω* (⋃ *x*∈*SDom σ* ∩ *SIGP a*. *SFV σ x*) **using** *agree-sub*[*OF sub VA*] **by** *auto*
show *?thesis*
apply(*rule uadmit-prog-fml-adjoint′*[*OF ssafe good-interp*])
subgoal by (*rule VA′*)
subgoal by (*rule hpsafe*)
done
qed

lemma *uadmit-fml-adjoint*:
assumes *FUA*:*FUadmit σ φ U*
assumes *VA*: *Vagree ν ω* (−*U*)
assumes *fsafe*:*fsafe φ*
assumes *ssafe*:*ssafe σ*
assumes *good-interp*:*is-interp I*
shows *fml-sem* (*adjoint I σ ν*) *φ* = *fml-sem* (*adjoint I σ ω*) *φ*
proof −
have *sub*:(⋃ *x*∈*SDom σ* ∩ *SIGF φ*. *SFV σ x*) ⊆ −*U* **using** *FUA* **unfolding** *FUadmit-def* **by** *auto*
have *VA′*: *Vagree ν ω* (⋃ *x*∈*SDom σ* ∩ *SIGF φ*. *SFV σ x*) **using** *agree-sub*[*OF sub VA*] **by** *auto*
show *?thesis*
apply(*rule uadmit-prog-fml-adjoint′*[*OF ssafe good-interp*])

subgoal by (*rule VA'*)
subgoal by (*rule fsafe*)
done
qed

lemma *ntadj-sub-assign*:$\bigwedge e\ \sigma\ x.\ (\bigcup y{\in}\{y.\ Inr\ y\ \in\ SIGT\ e\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y)\ \in\ SIGP\ (Assign\ x\ e)\}.\ FVT\ (\sigma\ y))$
by *auto*

lemma *ntadj-sub-diff-assign*:$\bigwedge e\ \sigma\ x.\ (\bigcup y{\in}\{y.\ Inl\ y\ \in\ SIGT\ e\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inl\ y)\ \in\ SIGP\ (DiffAssign\ x\ e)\}.\ FVT\ (\sigma\ y))$
by *auto*

lemma *ntadj-sub-geq1*:$\bigwedge\sigma\ x1\ x2.\ (\bigcup y{\in}\{y.\ Inl\ y\ \in\ SIGT\ x1\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inl\ y)\ \in\ SIGF\ (Geq\ x1\ x2)\}.\ FVT\ (\sigma\ y))$
by *auto*

lemma *ntadj-sub-geq2*:$\bigwedge\sigma\ x1\ x2.\ (\bigcup y{\in}\{y.\ Inl\ y\ \in\ SIGT\ x2\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inl\ y)\ \in\ SIGF\ (Geq\ x1\ x2)\}.\ FVT\ (\sigma\ y))$
by *auto*

lemma *ntadj-sub-prop*:$\bigwedge\sigma\ x1\ x2\ j.\ (\bigcup y{\in}\{y.\ Inl\ y\ \in\ SIGT\ (x2\ j)\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inl\ y)\ \in\ SIGF\ (\$\varphi\ x1\ x2)\}.FVT\ (\sigma\ y))$
by *auto*

lemma *ntadj-sub-ode*:$\bigwedge\sigma\ x1\ x2.\ (\bigcup y{\in}\{y.\ Inl\ (Inl\ y)\ \in\ SIGO\ x1\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inl\ y)\ \in\ SIGP\ (EvolveODE\ x1\ x2)\}.\ FVT\ (\sigma\ y))$
by *auto*

lemma *uadmit-prog-fml-ntadjoint'*:
fixes $\sigma\ I$
assumes *ssafe*:$\bigwedge i.\ dsafe\ (\sigma\ i)$
assumes *good-interp*:*is-interp I*
shows $\bigwedge\nu\ \omega.\ Vagree\ \nu\ \omega\ (\bigcup x{\in}\{x.\ Inl\ (Inr\ x)\ \in\ SIGP\ \alpha\}.\ FVT\ (\sigma\ x)) \Longrightarrow hpsafe\ \alpha \Longrightarrow prog\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ \alpha = prog\text{-}sem\ (adjointFO\ I\ \sigma\ \omega)\ \alpha$
and $\bigwedge\nu\ \omega.\ Vagree\ \nu\ \omega\ (\bigcup x{\in}\{x.\ Inl\ (Inr\ x)\ \in\ SIGF\ \varphi\}.\ FVT\ (\sigma\ x)) \Longrightarrow fsafe\ \varphi \Longrightarrow fml\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ \varphi = fml\text{-}sem\ (adjointFO\ I\ \sigma\ \omega)\ \varphi$
proof (*induct* α **and** φ)
case (*Pvar x*)
then show *?case* **unfolding** *adjointFO-def* **by** *auto*
next
case (*Assign x e*)
assume *VA*:*Vagree* $\nu\ \omega\ (\bigcup y{\in}\{y.\ Inl\ (Inr\ y)\ \in\ SIGP\ (Assign\ x\ e)\}.\ FVT\ (\sigma\ y))$
assume *safe*:*hpsafe* $(x := e)$
from *safe* **have** *dsafe*:*dsafe e* **by** (*auto dest*: *hpsafe.cases*)
have *sub*:$(\bigcup y{\in}\{y.\ Inr\ y\ \in\ SIGT\ e\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y)\ \in\ SIGP\ (Assign\ x\ e)\}.\ FVT\ (\sigma\ y))$
using *ntadj-sub-assign*[*of* $\sigma\ e\ x$] **by** *auto*
have *VA'*:$(Vagree\ \nu\ \omega\ (\bigcup i{\in}\{i.\ Inr\ i\ \in\ SIGT\ e\}.\ FVT\ (\sigma\ i)))$

using *agree-sub*[*OF sub VA*] **by** *auto*
have *dterm-sem* (*adjointFO I* σ ν) *e* = *dterm-sem* (*adjointFO I* σ ω) *e*
using *uadmit-dterm-ntadjoint'*[*of* σ *I* ν ω *e*] *ssafe good-interp agree-sub*[*OF sub VA*] *dsafe* **by** *auto*
then show *?case* **by** (*auto simp add*: *vec-eq-iff*)
next
case (*DiffAssign x e*)
assume *VA*:*Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*DiffAssign x e*)}. *FVT* (σ *y*))
assume *safe*:*hpsafe* (*DiffAssign x e*)
from *safe* **have** *dsafe*:*dsafe e* **by** (*auto dest*: *hpsafe.cases*)
have *sub*:($\bigcup$ *y*∈{*y*. *Inr y* ∈ *SIGT e*}. *FVT* (σ *y*)) ⊆ ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*DiffAssign x e*)}. *FVT* (σ *y*))
using *ntadj-sub-assign*[*of* σ *e x*] **by** *auto*
have *VA'*:(*Vagree* ν ω ($\bigcup$ *i*∈{*i*. *Inr i* ∈ *SIGT e*}. *FVT* (σ *i*)))
using *agree-sub*[*OF sub VA*] **by** *auto*
have *dterm-sem* (*adjointFO I* σ ν) *e* = *dterm-sem* (*adjointFO I* σ ω) *e*
using *uadmit-dterm-ntadjoint'*[*of* σ *I* ν ω *e*] *ssafe good-interp agree-sub*[*OF sub VA*] *dsafe* **by** *auto*
then show *?case* **by** (*auto simp add*: *vec-eq-iff*)
next
case (*Test x*)
assume *IH*:$\bigwedge\nu$ ω. *Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x*}. *FVT* (σ *y*)) ⟹ *fsafe x* ⟹ *fml-sem* (*adjointFO I* σ ν) *x* = *fml-sem* (*adjointFO I* σ ω) *x*
assume *VA*:*Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*? x*)}. *FVT* (σ *y*))
assume *hpsafe*:*hpsafe* (*? x*)
then have *fsafe*:*fsafe x* **by** (*auto dest*: *hpsafe.cases*)
have *sub*:($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x*}. *FVT* (σ *y*)) ⊆ ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*? x*)}. *FVT* (σ *y*))
by *auto*
have *fml-sem* (*adjointFO I* σ ν) *x* = *fml-sem* (*adjointFO I* σ ω) *x*
using *IH*[*OF agree-sub*[*OF sub VA*] *fsafe*] **by** *auto*
then show *?case* **by** *auto*
next
case (*EvolveODE x1 x2*)
assume *IH*:$\bigwedge\nu$ ω. *Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x2*}. *FVT* (σ *y*)) ⟹ *fsafe x2* ⟹ *fml-sem* (*adjointFO I* σ ν) *x2* = *fml-sem* (*adjointFO I* σ ω) *x2*
assume *VA*:*Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*EvolveODE x1 x2*)}. *FVT* (σ *y*))
assume *safe*:*hpsafe* (*EvolveODE x1 x2*)
then have *osafe*:*osafe x1* **and** *fsafe*:*fsafe x2* **by** (*auto dest*: *hpsafe.cases*)
have *sub1*:($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x2*}. *FVT* (σ *y*)) ⊆ ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*EvolveODE x1 x2*)}. *FVT* (σ *y*))
by *auto*
then have *VAF*:*Vagree* ν ω ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x2*}. *FVT* (σ *y*))
using *agree-sub*[*OF sub1 VA*] **by** *auto*
note *IH'* = *IH*[*OF VAF fsafe*]
have *sub*:($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGO x1*}. *FVT* (σ *y*)) ⊆ ($\bigcup$ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*EvolveODE x1 x2*)}. *FVT* (σ *y*))

by *auto*
moreover have *IH2*:*ODE-sem* (*adjointFO I σ ν*) *x1* = *ODE-sem* (*adjointFO I σ ω*) *x1*
apply (*rule uadmit-ode-ntadjoint′*)
subgoal by (*rule ssafe*)
subgoal by (*rule good-interp*)
defer subgoal by (*rule osafe*)
using *agree-sub*[*OF sub VA*] **by** *auto*
have *mkv*:*mk-v* (*adjointFO I σ ν*) *x1* = *mk-v* (*adjointFO I σ ω*) *x1*
apply (*rule uadmit-mkv-ntadjoint*)
using *ssafe good-interp osafe agree-sub*[*OF sub VA*] **by** *auto*
show *?case*
apply *auto*
subgoal for *aa ba bb sol t*
apply(*rule exI*[**where** *x* = *sol*])
apply(*rule conjI*)
subgoal by *auto*
apply(*rule exI*[**where** *x*=*t*])
apply(*rule conjI*)
subgoal using *mkv* **by** *auto*
apply(*rule conjI*)
subgoal by *auto* **using** *IH2 mkv IH′* **by** *auto*
subgoal for *aa ba bb sol t*
apply(*rule exI*[**where** *x* = *sol*])
apply(*rule conjI*)
subgoal by *auto*
apply(*rule exI*[**where** *x*=*t*])
apply(*rule conjI*)
subgoal using *mkv* **by** *auto*
apply(*rule conjI*)
subgoal by *auto* **using** *IH2 mkv IH′* **by** *auto*
done
next
case (*Choice x1 x2*)
assume *IH1*:⋀*ν ω*. *Vagree ν ω* (⋃*y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP x1*}. *FVT* (*σ y*)) ⟹ *hpsafe x1* ⟹ *prog-sem* (*adjointFO I σ ν*) *x1* = *prog-sem* (*adjointFO I σ ω*) *x1*
assume *IH2*:⋀*ν ω*. *Vagree ν ω* (⋃*y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP x2*}. *FVT* (*σ y*)) ⟹ *hpsafe x2* ⟹ *prog-sem* (*adjointFO I σ ν*) *x2* = *prog-sem* (*adjointFO I σ ω*) *x2*
assume *VA*: *Vagree ν ω* (⋃*y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*x1* ∪∪ *x2*)}. *FVT* (*σ y*))
assume *safe*:*hpsafe* (*x1* ∪∪ *x2*)
from *safe* **have**
safe1:*hpsafe x1*
and *safe2*:*hpsafe x2*
by (*auto dest*: *hpsafe.cases*)
have *sub1*:(⋃*y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*x1*)}. *FVT* (*σ y*)) ⊆ (⋃*y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGP* (*x1* ∪∪ *x2*)}. *FVT* (*σ y*))
by *auto*

have *sub2*:($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x2)\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x1 \cup\cup x2)\}.\ FVT\ (\sigma\ y))$
by *auto*
then show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
next
case (*Sequence x1 x2*)
assume *IH1*:$\bigwedge\nu\ \omega.\ Vagree\ \nu\ \omega\ (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ x1\}.\ FVT\ (\sigma\ y)) \Longrightarrow hpsafe\ x1 \Longrightarrow prog\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ x1 = prog\text{-}sem\ (adjointFO\ I\ \sigma\ \omega)\ x1$
assume *IH2*:$\bigwedge\nu\ \omega.\ Vagree\ \nu\ \omega\ (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ x2\}.\ FVT\ (\sigma\ y)) \Longrightarrow hpsafe\ x2 \Longrightarrow prog\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ x2 = prog\text{-}sem\ (adjointFO\ I\ \sigma\ \omega)\ x2$
assume *VA*:*Vagree* $\nu\ \omega\ (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x1\ ;;\ x2)\}.\ FVT\ (\sigma\ y))$
assume *safe*:*hpsafe* $(x1\ ;;\ x2)$
from *safe* **have**
safe1:*hpsafe x1*
and *safe2*:*hpsafe x2*
by (*auto dest*: *hpsafe.cases*)
have *sub1*:$(\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ x1\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x1\ ;;\ x2)\}.\ FVT\ (\sigma\ y))$
by *auto*
have *sub2*:$(\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ x2\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x1\ ;;\ x2)\}.\ FVT\ (\sigma\ y))$
by *auto*
then show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
next
case (*Loop x*)
assume *IH*:$\bigwedge\nu\ \omega.\ Vagree\ \nu\ \omega\ (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ x\}.\ FVT\ (\sigma\ y)) \Longrightarrow hpsafe\ x \Longrightarrow prog\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ x = prog\text{-}sem\ (adjointFO\ I\ \sigma\ \omega)\ x$
assume *VA*:*Vagree* $\nu\ \omega\ (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x{**}\ )\}.\ FVT\ (\sigma\ y))$
assume *safe*:*hpsafe* $(x{**}\ )$
from *safe* **have**
safe:*hpsafe x*
by (*auto dest*: *hpsafe.cases*)
have *sub*:$(\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x\ )\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ (x{**}\ )\}.\ FVT\ (\sigma\ y))$
by *auto*
show *?case* **using** *IH*[*OF agree-sub*[*OF sub VA*] *safe*] **by** *auto*
next
case (*Geq x1 x2*)
assume *VA*:*Vagree* $\nu\ \omega\ (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (Geq\ x1\ x2)\}.\ FVT\ (\sigma\ y))$
assume *safe*:*fsafe* $(Geq\ x1\ x2)$
then have *dsafe1*:*dsafe x1* **and** *dsafe2*:*dsafe x2* **by** (*auto dest*: *fsafe.cases*)
have *sub1*:$(\bigcup y{\in}\{y.\ Inr\ y \in SIGT\ x1\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (Geq\ x1\ x2)\}.\ FVT\ (\sigma\ y))$
by *auto*
have *sub2*:$(\bigcup y{\in}\{y.\ Inr\ y \in SIGT\ x2\}.\ FVT\ (\sigma\ y)) \subseteq (\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in$

SIGF (*Geq x1 x2*)}. *FVT* (*σ y*))
by *auto*
have *dterm-sem* (*adjointFO I σ ν*) *x1* = *dterm-sem* (*adjointFO I σ ω*) *x1*
by (*rule uadmit-dterm-ntadjoint'*[*OF ssafe good-interp agree-sub*[*OF sub1 VA*] *dsafe1*])
moreover have *dterm-sem* (*adjointFO I σ ν*) *x2* = *dterm-sem* (*adjointFO I σ ω*) *x2*
by (*rule uadmit-dterm-ntadjoint'*[*OF ssafe good-interp agree-sub*[*OF sub2 VA*] *dsafe2*])
ultimately show *?case* **by** *auto*
next
case (*Prop x1 x2 ν ω*)
assume *VA*:*Vagree ν ω* (⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF* (*$φ x1 x2*)}. *FVT* (*σ y*))
assume *safe*:*fsafe* (*$φ x1 x2*)
then have *safes*:⋀*i*. *dsafe* (*x2 i*) **using** *dfree-is-dsafe* **by** *auto*
have *subs*:⋀*j*. (⋃ *y*∈{*y*. *Inr y* ∈ *SIGT* (*x2 j*)}. *FVT* (*σ y*)) ⊆ (⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF* (*$φ x1 x2*)}. *FVT* (*σ y*))
subgoal for *j* **by** *auto*
done
have ⋀*i*. *dterm-sem* (*adjointFO I σ ν*) (*x2 i*) = *dterm-sem* (*adjointFO I σ ω*) (*x2 i*)
by (*rule uadmit-dterm-ntadjoint'*[*OF ssafe good-interp agree-sub*[*OF subs VA*] *safes*])
then have *vec-eq*:⋀*R*. (*χ i*. *dterm-sem* (*adjointFO I σ ν*) (*x2 i*) *R*) = (*χ i*. *dterm-sem* (*adjointFO I σ ω*) (*x2 i*) *R*)
by (*auto simp add*: *vec-eq-iff*)
from *VA* **have** *VAs*:⋀*j*. *Vagree ν ω* (⋃ *y*∈{*y*. *Inr y* ∈ *SIGT* (*x2 j*)}. *FVT* (*σ y*))
subgoal for *j*
using *agree-sub*[*OF subs*[*of j*] *VA*] **by** *auto*
done
then show *?case*
using *vec-eq* **by** (*auto simp add*: *adjointFO-def*)
next
case (*Not x*)
assume *IH*:⋀*ν ω*. *Vagree ν ω* (⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x*}. *FVT* (*σ y*)) ⟹ *fsafe x* ⟹ *fml-sem* (*adjointFO I σ ν*) *x* = *fml-sem* (*adjointFO I σ ω*) *x*
assume *VA*:*Vagree ν ω* (⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF* (*Not x*)}. *FVT* (*σ y*))
assume *safe*:*fsafe* (*Not x*)
from *safe* **have**
safe:*fsafe x*
by (*auto dest*: *fsafe.cases*)
have *sub*:(⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x*}. *FVT* (*σ y*)) ⊆ (⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF* (*Not x*)}. *FVT* (*σ y*))
by *auto*
show *?case* **using** *IH*[*OF agree-sub*[*OF sub VA*] *safe*] **by** *auto*
next
case (*And x1 x2*)
assume *IH1*:⋀*ν ω*. *Vagree ν ω* (⋃ *y*∈{*y*. *Inl* (*Inr y*) ∈ *SIGF x1*}. *FVT* (*σ y*)) ⟹ *fsafe x1* ⟹ *fml-sem* (*adjointFO I σ ν*) *x1* = *fml-sem* (*adjointFO I σ ω*) *x1*

assume *IH2*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\Longrightarrow$ *fsafe x2* $\Longrightarrow$ *fml-sem* (*adjointFO I* $\sigma\ \nu$) *x2* = *fml-sem* (*adjointFO I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (And\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
assume *safe*:*fsafe* (*And x1 x2*)
from *safe* **have**
safe1:*fsafe x1*
and *safe2*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub1*:($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x1\}.\ FVT\ (\sigma\ y)$) $\subseteq$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (And\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
by *auto*
have *sub2*:($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\subseteq$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (And\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
by *auto*
show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
next
case (*Exists x1 x2*)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\Longrightarrow$ *fsafe x2* $\Longrightarrow$ *fml-sem* (*adjointFO I* $\sigma\ \nu$) *x2* = *fml-sem* (*adjointFO I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (Exists\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
assume *safe*:*fsafe* (*Exists x1 x2*)
from *safe* **have** *safe1*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub1*:($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\subseteq$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (Exists\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
by *auto*
show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] **by** *auto*
next
case (*Diamond x1 x2*)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ x1\}.\ FVT\ (\sigma\ y)$) $\Longrightarrow$ *hpsafe x1* $\Longrightarrow$ *prog-sem* (*adjointFO I* $\sigma\ \nu$) *x1* = *prog-sem* (*adjointFO I* $\sigma\ \omega$) *x1*
assume *IH2*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\Longrightarrow$ *fsafe x2* $\Longrightarrow$ *fml-sem* (*adjointFO I* $\sigma\ \nu$) *x2* = *fml-sem* (*adjointFO I* $\sigma\ \omega$) *x2*
assume *VA*:*Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (Diamond\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
assume *safe*:*fsafe* (*Diamond x1 x2*)
from *safe* **have**
safe1:*hpsafe x1*
and *safe2*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub1*:($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGP\ x1\}.\ FVT\ (\sigma\ y)$) $\subseteq$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (Diamond\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
by *auto*
have *sub2*:($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\subseteq$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (Diamond\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
by *auto*

show *?case* **using** *IH1*[*OF agree-sub*[*OF sub1 VA*] *safe1*] *IH2*[*OF agree-sub*[*OF sub2 VA*] *safe2*] **by** *auto*
next
case (*InContext x1 x2*)
assume *IH1*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\Longrightarrow$ *fsafe x2* $\Longrightarrow$ *fml-sem* (*adjointFO I* $\sigma\ \nu$) *x2* = *fml-sem* (*adjointFO I* $\sigma\ \omega$) *x2*
assume *VA*: *Vagree* $\nu\ \omega$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (InContext\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
assume *safe*:*fsafe* (*InContext x1 x2*)
from *safe* **have** *safe1*:*fsafe x2*
by (*auto dest*: *fsafe.cases*)
have *sub*:($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ x2\}.\ FVT\ (\sigma\ y)$) $\subseteq$ ($\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGF\ (InContext\ x1\ x2)\}.\ FVT\ (\sigma\ y)$)
by *auto*
show *?case* **using** *IH1*[*OF agree-sub*[*OF sub VA*] *safe1*]
unfolding *adjointFO-def* **by** *auto*
qed

lemma *uadmit-prog-ntadjoint*:
assumes *TUA*:*PUadmitFO* $\sigma\ \alpha\ U$
assumes *VA*: *Vagree* $\nu\ \omega\ (-U)$
assumes *dfree*:$\bigwedge i$. *dsafe* $(\sigma\ i)$
assumes *hpsafe*:*hpsafe* α
assumes *good-interp*:*is-interp I*
shows *prog-sem* (*adjointFO I* $\sigma\ \nu$) α = *prog-sem* (*adjointFO I* $\sigma\ \omega$) α
proof −
have *sub*:($\bigcup x{\in}\{x.\ Inl\ (Inr\ x) \in SIGP\ \alpha\}.\ FVT\ (\sigma\ x)$) $\subseteq -U$ **using** *TUA* **unfolding** *PUadmitFO-def* **by** *auto*
have *VA′*: *Vagree* $\nu\ \omega$ ($\bigcup x{\in}\{x.\ Inl\ (Inr\ x) \in SIGP\ \alpha\}.\ FVT\ (\sigma\ x)$) **using** *agree-sub*[*OF sub VA*] **by** *auto*
show *?thesis*
apply(*rule uadmit-prog-fml-ntadjoint′*[*OF dfree good-interp*])
subgoal by (*rule VA′*)
subgoal by (*rule hpsafe*)
done
qed

lemma *uadmit-fml-ntadjoint*:
assumes *TUA*:*FUadmitFO* $\sigma\ \varphi\ U$
assumes *VA*: *Vagree* $\nu\ \omega\ (-U)$
assumes *dfree*:$\bigwedge i$. *dsafe* $(\sigma\ i)$
assumes *fsafe*:*fsafe* φ
assumes *good-interp*:*is-interp I*
shows *fml-sem* (*adjointFO I* $\sigma\ \nu$) φ = *fml-sem* (*adjointFO I* $\sigma\ \omega$) φ
proof −
have *sub*:($\bigcup x{\in}\{x.\ Inl\ (Inr\ x) \in SIGF\ \varphi\}.\ FVT\ (\sigma\ x)$) $\subseteq -U$ **using** *TUA* **unfolding** *FUadmitFO-def* **by** *auto*
have *VA′*: *Vagree* $\nu\ \omega$ ($\bigcup x{\in}\{x.\ Inl\ (Inr\ x) \in SIGF\ \varphi\}.\ FVT\ (\sigma\ x)$) **using** *agree-sub*[*OF sub VA*] **by** *auto*

show *?thesis*
apply(*rule uadmit-prog-fml-ntadjoint'*[*OF dfree good-interp*])
subgoal by (*rule VA'*)
subgoal by (*rule fsafe*)
done
qed

11.3 Substitution theorems for terms

lemma *nsubst-sterm*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes *ν*::'*sz state*
shows *TadmitFFO σ ϑ* ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ *sterm-sem I* (*TsubstFO ϑ σ*) (*fst ν*) = *sterm-sem* (*adjointFO I σ ν*) *ϑ* (*fst ν*)
proof (*induction rule*: *TadmitFFO.induct*)
case (*TadmitFFO-Fun1 σ args f*)
then show *?case* **by**(*auto simp add*: *adjointFO-def*)
next
case (*TadmitFFO-Fun2 σ args f*)
then show *?case*
apply(*auto simp add*: *adjointFO-def*)
by (*simp add*: *dsem-to-ssem*)
qed (*auto*)

lemma *nsubst-sterm'*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes *a b*::'*sz simple-state*
shows *TadmitFFO σ ϑ* ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ *sterm-sem I* (*TsubstFO ϑ σ*) *a* = *sterm-sem* (*adjointFO I σ* (*a*,*b*)) *ϑ a*
using *nsubst-sterm* **by** (*metis fst-conv*)

lemma *ntsubst-preserves-free*:
dfree ϑ ⟹ (⋀*i*. *dfree* (*σ i*)) ⟹ *dfree*(*TsubstFO ϑ σ*)
proof (*induction rule*: *dfree.induct*)
case (*dfree-Fun args i*) **then show** *?case*
by (*cases i*) (*auto intro*:*dfree.intros*)
qed (*auto intro*: *dfree.intros*)

lemma *tsubst-preserves-free*:
dfree ϑ ⟹ (⋀*i f'*. *SFunctions σ i* = *Some f'* ⟹ *dfree f'*) ⟹ *dfree*(*Tsubst ϑ σ*)
proof (*induction rule*: *dfree.induct*)
case (*dfree-Fun args i*) **then show** *?case*
by (*cases SFunctions σ i*) (*auto intro*:*dfree.intros ntsubst-preserves-free*)
qed (*auto intro*: *dfree.intros*)

lemma *subst-sterm*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes *ν*::'*sz state*
shows

```
    TadmitF σ ϑ  ⟹
    (⋀i f'. SFunctions σ i = Some f' ⟹ dfree f') ⟹
     sterm-sem I (Tsubst ϑ σ) (fst ν) = sterm-sem (adjoint I σ ν) ϑ (fst ν)
proof (induction rule: TadmitF.induct)
  case (TadmitF-Fun1  σ args f f') then
    have subFree: TadmitFFO (λi. Tsubst (args i) σ) f'
      and frees:⋀i. dfree (Tsubst (args i) σ)
      and TFA:⋀i. TadmitF σ (args i)
      and NTFA:TadmitFFO (λi. Tsubst (args i) σ) f'
    and theIH:⋀i. sterm-sem I (Tsubst (args i) σ) (fst ν) = sterm-sem (local.adjoint
I σ ν) (args i) (fst ν)
       by auto
      from frees have safes:⋀i. dsafe (Tsubst (args i) σ)
       by (simp add: dfree-is-dsafe)
  assume subFreeer:(⋀i f'. SFunctions σ i = Some f' ⟹ dfree f')
  note admit = TadmitF-Fun1.hyps(1) and sfree = TadmitF-Fun1.prems(1)
  have IH:(⋀i. sterm-sem I (Tsubst (args i) σ) (fst ν) = sterm-sem (adjoint I σ
ν) (args i) (fst ν))
    using  admit TadmitF-Fun1.prems TadmitF-Fun1.IH by auto
   have vec-eq:(χ i. sterm-sem (local.adjoint I σ ν) (args i) (fst ν)) = (χ i.
sterm-sem I (Tsubst (args i) σ) (fst ν))
    apply(rule vec-extensionality)
    using IH by auto
  assume some:SFunctions σ f = Some f'
  let ?sub = (λ i. Tsubst (args i) σ)
  have IH2:sterm-sem I (TsubstFO f' ?sub) (fst ν) = sterm-sem (adjointFO I ?sub
ν) f' (fst ν)
    apply(rule nsubst-sterm)
     apply(rule subFree)
    by (rule safes)
  show ?case
    apply (simp add: some)
    unfolding vec-eq IH2
     by (auto simp add: some adjoint-free[OF subFreeer, of σ (λ x y. x) I ν]
adjointFO-free[OF frees])
next
  case (TadmitF-Fun2  σ args f)
  assume none:SFunctions σ f = None
  note admit = TadmitF-Fun2.hyps(1) and sfree = TadmitF-Fun2.prems(1)
  have IH:(⋀i. TadmitF σ (args i) ⟹
      sterm-sem I (Tsubst (args i) σ) (fst ν) = sterm-sem (adjoint I σ ν) (args i)
(fst ν))
    using  TadmitF-Fun2.prems TadmitF-Fun2.IH by auto
   have eqs:⋀i. sterm-sem I (Tsubst (args i) σ) (fst ν) = sterm-sem (adjoint I σ
ν) (args i) (fst ν)
    by (auto simp add: IH admit)
  show ?case
    by(auto simp add: none IH adjoint-def vec-extensionality eqs)
  qed auto
```

lemma *nsubst-dterm'*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes *ν*::'*sz state*
assumes *good-interp*:*is-interp I*
shows *TadmitFO σ ϑ* ⟹ *dfree ϑ* ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ *dterm-sem I* (*TsubstFO ϑ σ*) *ν* = *dterm-sem* (*adjointFO I σ ν*) *ϑ ν*
proof (*induction rule*: *TadmitFO.induct*)
case (*TadmitFO-Fun σ args f*)
assume *admit*:⋀*i*. *TadmitFO σ* (*args i*)
assume *IH*:⋀*i*. *dfree* (*args i*) ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ *dterm-sem I* (*TsubstFO* (*args i*) *σ*) *ν* = *dterm-sem* (*adjointFO I σ ν*) (*args i*) *ν*
assume *free*:*dfree* (*$f f args*)
assume *safe*:⋀*i*. *dsafe* (*σ i*)
from *free* **have** *frees*: ⋀*i*. *dfree* (*args i*) **by** (*auto dest*: *dfree.cases*)
have *sem*:⋀*i*. *dterm-sem I* (*TsubstFO* (*args i*) *σ*) *ν* = *dterm-sem* (*adjointFO I σ ν*) (*args i*) *ν*
using *IH*[*OF frees safe*] **by** *auto*
have *vecEq*: (*χ i*. *dterm-sem* (*adjointFO I σ ν*) (*args i*) *ν*) =
(*χ i*. *dterm-sem*
(⦇*Functions* = *case-sum* (*Functions I*) (*λf'* -. *dterm-sem I* (*σ f'*) *ν*), *Predicates* = *Predicates I*, *Contexts* = *Contexts I*,
Programs = *Programs I*, *ODEs* = *ODEs I*, *ODEBV* = *ODEBV I*⦈)
(*args i*) *ν*)
apply(*rule vec-extensionality*)
by (*auto simp add*: *adjointFO-def*)
show *dterm-sem I* (*TsubstFO* (*$f f args*) *σ*) *ν* = *dterm-sem* (*adjointFO I σ ν*) (*$f f args*) *ν*
apply (*cases f*)
apply (*auto simp add*: *vec-extensionality adjointFO-def*)
using *sem* **apply** *auto*
subgoal for *a* **using** *vecEq* **by** *auto*
done
next
case (*TadmitFO-Diff σ ϑ*)
hence *admit*:*TadmitFFO σ ϑ*
and *admitU*:*NTUadmit σ ϑ UNIV*

and *safe*: *dfree* (*Differential ϑ*)
and *freeSub*:⋀*i*. *dsafe* (*σ i*)
by *auto*
from *safe* **have** *False* **by** (*auto dest*: *dfree.cases*)
then show *dterm-sem I* (*TsubstFO* (*Differential ϑ*) *σ*) *ν* = *dterm-sem* (*adjointFO I σ ν*) (*Differential ϑ*) *ν*
by *auto*
qed (*auto simp add*: *TadmitFO.cases*)

lemma *ntsubst-free-to-safe*:
dfree ϑ ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ *dsafe* (*TsubstFO ϑ σ*)

proof (*induction rule*: *dfree.induct*)
case (*dfree-Fun args i*) **then show** *?case*
by (*cases i*) (*auto intro*:*dsafe.intros ntsubst-preserves-free*)
qed (*auto intro*: *dsafe.intros*)

lemma *ntsubst-preserves-safe*:
dsafe $\vartheta \Longrightarrow (\bigwedge i.\ dfree\ (\sigma\ i)) \Longrightarrow dsafe\ (TsubstFO\ \vartheta\ \sigma)$
proof (*induction rule*: *dsafe.induct*)
case (*dsafe-Fun args i*) **then show** *?case*
by (*cases i*) (*auto intro*:*dsafe.intros ntsubst-preserves-free dfree-is-dsafe*)
next
case (*dsafe-Diff* ϑ) **then show** *?case*
by (*auto intro*:*dsafe.intros ntsubst-preserves-free*)
qed (*auto simp add*: *ntsubst-preserves-free intro*: *dsafe.intros*)

lemma *tsubst-preserves-safe*:
dsafe $\vartheta \Longrightarrow$ $(\bigwedge i\ f'.\ SFunctions\ \sigma\ i = Some\ f' \Longrightarrow dfree\ f') \Longrightarrow dsafe(Tsubst\ \vartheta\ \sigma)$
proof (*induction rule*: *dsafe.induct*)
case (*dsafe-Fun args i*)
assume *dsafes*:$\bigwedge i.\ dsafe\ (args\ i)$
assume *IH*:$\bigwedge j.\ (\bigwedge i\ f'.\ SFunctions\ \sigma\ i = Some\ f' \Longrightarrow dfree\ f') \Longrightarrow dsafe\ (Tsubst\ (args\ j)\ \sigma)$
assume *frees*:$\bigwedge i\ f.\ SFunctions\ \sigma\ i = Some\ f \Longrightarrow dfree\ f$
have *IH'*:$\bigwedge i.\ dsafe\ (Tsubst\ (args\ i)\ \sigma)$
using *frees IH* **by** *auto*
then show *?case*
apply *auto*
apply(*cases SFunctions* σ *i*)
subgoal using *IH frees* **by** *auto*
subgoal for *a* **using** *frees*[*of i a*] *ntsubst-free-to-safe*[*of a*] *IH'* **by** *auto*
done
qed (*auto intro*: *dsafe.intros tsubst-preserves-free*)

lemma *subst-dterm*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
assumes *good-interp*:*is-interp I*
shows
Tadmit σ $\vartheta \Longrightarrow$
dsafe $\vartheta \Longrightarrow$
$(\bigwedge i\ f'.\ SFunctions\ \sigma\ i = Some\ f' \Longrightarrow dfree\ f') \Longrightarrow$
$(\bigwedge f\ f'.\ SPredicates\ \sigma\ f = Some\ f' \Longrightarrow fsafe\ f') \Longrightarrow$
$(\bigwedge \nu.\ dterm\text{-}sem\ I\ (Tsubst\ \vartheta\ \sigma)\ \nu = dterm\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ \vartheta\ \nu)$
proof (*induction rule*: *Tadmit.induct*)
case (*Tadmit-Fun1* σ *args f f'* ν)
note *safe* = *Tadmit-Fun1.prems*(*1*) **and** *sfree* = *Tadmit-Fun1.prems*(*2*) **and** *TA* = *Tadmit-Fun1.hyps*(*1*)
and *some* = *Tadmit-Fun1.hyps*(*2*) **and** *NTA* = *Tadmit-Fun1.hyps*(*3*)
hence *safes*:$\bigwedge i.\ dsafe\ (args\ i)$ **by** *auto*
have *IH*:$(\bigwedge \nu'.\ \bigwedge i.\ dsafe\ (args\ i) \Longrightarrow$

```
    dterm-sem I (Tsubst (args i) σ) ν = dterm-sem (adjoint I σ ν) (args i) ν)
   using  Tadmit-Fun1.prems Tadmit-Fun1.IH by auto
  have eqs:⋀i ν'. dterm-sem I (Tsubst (args i) σ) ν = dterm-sem (adjoint I σ ν)
(args i) ν
   by (auto simp add: IH safes)
  let ?sub = (λ i. Tsubst (args i) σ)
  have subSafe:(⋀i. dsafe (?sub i))
   using tsubst-preserves-safe[OF safes sfree]
   by (simp add: safes sfree tsubst-preserves-safe)
  have freef:dfree f' using sfree some by auto
  have IH2:dterm-sem I (TsubstFO f' ?sub) ν = dterm-sem (adjointFO I ?sub ν)
f' ν
   by (simp add: nsubst-dterm'[OF good-interp NTA freef subSafe])
  have vec:(χ i. dterm-sem I (Tsubst (args i) σ) ν) = (χ i. dterm-sem (local.adjoint
I σ ν) (args i) ν)
   apply(auto simp add: vec-eq-iff)
   subgoal for i
    using IH[of i, OF safes[of i]]
    by auto
   done
  show ?case
   using IH safes eqs apply (auto simp add:  IH2  some good-interp)
   using some unfolding adjoint-def adjointFO-def by auto
next
  case (Tadmit-Fun2 σ args f ν)
  note safe = Tadmit-Fun2.prems(1) and sfree = Tadmit-Fun2.prems(2) and TA
= Tadmit-Fun2.hyps(1)
  and none = Tadmit-Fun2.hyps(2)
  hence safes:⋀i. dsafe (args i) by auto
  have IH:(⋀ν'. ⋀i. dsafe (args i) ⟹
     dterm-sem I (Tsubst (args i) σ) ν = dterm-sem (adjoint I σ ν) (args i) ν)
  using  Tadmit-Fun2.prems Tadmit-Fun2.IH by auto
  have Ieq:Functions I f = Functions (adjoint I σ ν) f
   using none unfolding adjoint-def by auto
  have vec:(χ i. dterm-sem I (Tsubst (args i) σ) ν) = (χ i. dterm-sem (adjoint I
σ ν) (args i) ν)
   apply(auto simp add: vec-eq-iff)
   subgoal for i using IH[of i, OF safes[of i]] by auto
   done
  show ?case using none IH Ieq vec by auto
next
  case (Tadmit-Diff σ ϑ)  then
  have TA:Tadmit σ ϑ
   and TUA:TUadmit σ ϑ UNIV
   and IH:dsafe ϑ ⟹ (⋀i f'. SFunctions σ i = Some f' ⟹ dfree f') ⟹ (⋀ν.
dterm-sem I (Tsubst ϑ σ) ν = dterm-sem (local.adjoint I σ ν) ϑ ν)
   and safe:dsafe (Differential ϑ)
   and sfree:⋀i f'1. SFunctions σ i = Some f'1 ⟹ dfree f'1
   and spsafe:⋀f f'. SPredicates σ f = Some f'  ⟹ fsafe f'
```

by *auto*
from *sfree* **have** *sdsafe*:$\bigwedge f\ f'$. *SFunctions* $\sigma\ f$ = *Some* $f' \Longrightarrow$ *dsafe* f'
using *dfree-is-dsafe* **by** *auto*
have *VA*:$\bigwedge \nu\ \omega$. *Vagree* $\nu\ \omega$ $(-UNIV)$ **unfolding** *Vagree-def* **by** *auto*
from *safe* **have** *free*:*dfree* ϑ **by** (*auto dest*: *dsafe.cases intro*: *dfree.intros*)
from *free* **have** *tsafe*:*dsafe* ϑ **using** *dfree-is-dsafe* **by** *auto*
have *freeSubst*:*dfree* (*Tsubst* $\vartheta\ \sigma$)
using *tsubst-preserves-free*[*OF free sfree*]
using *Tadmit-Diff.prems*(*2*) *free tsubst-preserves-free* **by** *blast*
have *IH'*:$\bigwedge \nu$. *dterm-sem I* (*Tsubst* $\vartheta\ \sigma$) ν = *dterm-sem* (*local.adjoint I* $\sigma\ \nu$) ϑ ν
using *IH*[*OF tsafe sfree*] **by** *auto*
have *sem-eq*:$\bigwedge \nu'$. *dsafe* $\vartheta \Longrightarrow$ *is-interp I* $\Longrightarrow$ *dterm-sem* (*local.adjoint I* $\sigma\ \nu$) ϑ = *dterm-sem* (*local.adjoint I* $\sigma\ \nu'$) ϑ
subgoal for ν'
using *uadmit-dterm-adjoint*[*OF TUA VA sfree spsafe*, *of* ($\lambda\ x\ y.\ x$) ($\lambda\ x\ y.\ x$) *I* $\nu\ \nu'$]
by *auto*
done
have *IH''*:$\bigwedge \nu'$. *dterm-sem I* (*Tsubst* $\vartheta\ \sigma$) ν' = *dterm-sem* (*local.adjoint I* $\sigma\ \nu$) $\vartheta\ \nu'$
subgoal for ν'
using *sem-eq*[*OF tsafe good-interp*, *of* ν'] *IH'*[*of* ν'] **by** *auto*
done
have *sem-eq*:*sterm-sem I* (*Tsubst* $\vartheta\ \sigma$) = *sterm-sem* (*local.adjoint I* $\sigma\ \nu$) ϑ
apply (*auto simp add*: *fun-eq-iff*)
subgoal for ν'
apply (*cases* ν')
subgoal for ν''
apply *auto*
using *dsem-to-ssem*[*OF free*, *of* (*local.adjoint I* $\sigma\ \nu$) (ν',ν')] *dsem-to-ssem*[*OF freeSubst*, *of I* (ν',ν')] *IH'*[*of* (ν)]
apply *auto*
using *IH''* **by** *auto*
done
done
show *?case*
apply (*auto simp add*: *directional-derivative-def fun-eq-iff*)
using *sterm-determines-frechet*[*OF*
good-interp
adjoint-safe[*OF good-interp sfree*]
tsubst-preserves-free[*OF free sfree*]
free sem-eq]
by *auto*
qed *auto*

11.4 Substitution theorems for ODEs

lemma *osubst-preserves-safe*:

assumes *ssafe*:*ssafe* σ
shows (*osafe ODE* $\Longrightarrow$ *Oadmit* σ *ODE U* $\Longrightarrow$ *osafe* (*Osubst ODE* σ))
proof (*induction rule*: *osafe.induct*)
case (*osafe-Var c*)
then show *?case* **using** *ssafe* **unfolding** *ssafe-def* **by** (*cases SODEs* σ *c*, *auto intro*: *osafe.intros*)
next
case (*osafe-Sing* ϑ *x*)
then show *?case*
using *tsubst-preserves-free ssafe* **unfolding** *ssafe-def* **by** (*auto intro*: *osafe.intros*)
next
case (*osafe-Prod ODE1 ODE2*)
moreover have *Oadmit* σ *ODE1 U Oadmit* σ *ODE2 U ODE-dom* (*Osubst ODE1* σ) $\cap$ *ODE-dom* (*Osubst ODE2* σ) = {}
using *osafe-Prod.prems* **by** (*auto dest*: *Oadmit.cases*)
ultimately show *?case* **by** (*auto intro*: *osafe.intros*)
qed

lemma *nosubst-preserves-safe*:
assumes *sfree*:$\bigwedge i$. *dfree* (σ *i*)
fixes α ::('*a* + '*d*, '*b*, '*c*) *hp* **and** φ ::('*a* + '*d*, '*b*, '*c*) *formula*
shows (*osafe ODE* $\Longrightarrow$ *OUadmitFO* σ *ODE U* $\Longrightarrow$ *osafe* (*OsubstFO ODE* σ))
proof (*induction rule*: *osafe.induct*)
case (*osafe-Var c*)
then show *?case* **by** (*auto intro*: *osafe.intros*)
next
case (*osafe-Sing* ϑ *x*)
then show *?case* **using** *sfree ntsubst-preserves-free*[*of* ϑ σ] **unfolding** *OUadmitFO-def* **by** (*auto intro*: *osafe.intros*)
next
case (*osafe-Prod ODE1 ODE2*)
assume *safe1*:*osafe ODE1*
and *safe2*:*osafe ODE2*
and *disj*:*ODE-dom ODE1* $\cap$ *ODE-dom ODE2* = {}
and *IH1*:*OUadmitFO* σ *ODE1 U* $\Longrightarrow$ *osafe* (*OsubstFO ODE1* σ)
and *IH2*:*OUadmitFO* σ *ODE2 U* $\Longrightarrow$ *osafe* (*OsubstFO ODE2* σ)
and *NOUA*:*OUadmitFO* σ (*OProd ODE1 ODE2*) *U*
have *nosubst-preserves-ODE-dom*:$\bigwedge$*ODE*. *ODE-dom* (*OsubstFO ODE* σ) = *ODE-dom ODE*
subgoal for *ODE*
apply(*induction ODE*)
by *auto*
done
have *disj'*:*ODE-dom* (*OsubstFO ODE1* σ) $\cap$ *ODE-dom* (*OsubstFO ODE2* σ) = {}
using *disj nosubst-preserves-ODE-dom* **by** *auto*
from *NOUA* **have** *NOUA1*:*OUadmitFO* σ *ODE1 U* **and** *NOUA2*:*OUadmitFO* σ *ODE2 U* **unfolding** *OUadmitFO-def* **by** *auto*
then show *?case* **using** *IH1*[*OF NOUA1*] *IH2*[*OF NOUA2*] *disj'* **by** (*auto intro*:

osafe.intros)
qed

lemma *nsubst-dterm*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes ν::'*sz state*
fixes ν'::'*sz state*
assumes *good-interp*:*is-interp I*
shows *TadmitFO* σ ϑ $\Longrightarrow$ *dsafe* ϑ $\Longrightarrow$ ($\bigwedge i$. *dsafe* (σ *i*)) $\Longrightarrow$ *dterm-sem I* (*TsubstFO* ϑ σ) ν = *dterm-sem* (*adjointFO I* σ ν) ϑ ν
proof (*induction rule*: *TadmitFO.induct*)
case (*TadmitFO-Diff* σ ϑ) **then**
have *subFree*:($\bigwedge i$. *dsafe* (σ *i*))
and *NTFA*:*TadmitFFO* σ ϑ
and *substFree*:*dfree* (*TsubstFO* ϑ σ)
and *dsafe*:*dsafe* (*Differential* ϑ)
and *subSafe*:$\bigwedge i$. *dsafe* (σ *i*)
and *NTU*:*NTUadmit* σ ϑ *UNIV*
by *auto*
have *dfree*:*dfree* ϑ **using** *dsafe* **by** *auto*
then show *?case*
apply *auto*
apply (*unfold directional-derivative-def*)
apply (*rule sterm-determines-frechet*)
subgoal using *good-interp* **by** *auto*
subgoal using *adjointFO-safe*[*OF good-interp*, *of* σ] *subSafe* **by** *auto*
subgoal using *substFree* **by** *auto*
subgoal using *dfree* **by** *auto*
subgoal
apply(*rule ext*)
subgoal for *x*
using *nsubst-sterm'*[*of* σ ϑ *I* (*fst* ν) (*snd* ν), *OF NTFA subSafe*] **apply** *auto*
proof −
assume *sem*:*sterm-sem I* (*TsubstFO* ϑ σ) (*fst* ν) = *sterm-sem* (*adjointFO I* σ ν) ϑ (*fst* ν)
have *VA*:$\bigwedge\nu$ ω. *Vagree* ν (*x*,*snd* ν) (−*UNIV*) **unfolding** *Vagree-def* **by** *auto*
show *sterm-sem I* (*TsubstFO* ϑ σ) *x* = *sterm-sem* (*adjointFO I* σ ν) ϑ *x*
using *uadmit-sterm-ntadjoint*[*OF NTU VA subSafe*, *OF good-interp*, *of* (*x*, *snd* ν)]
nsubst-sterm[*OF NTFA subSafe*, *of I* ν]
apply *auto*
using *NTU VA dfree-is-dsafe dsafe subSafe substFree good-interp uadmit-sterm-ntadjoint*
by (*metis NTFA fst-eqD nsubst-sterm*)
qed
done
done

next
case (*TadmitFO-Fun σ args f*)
then show *?case* **apply** *auto* **apply**(*cases f*) **unfolding** *adjointFO-def* **by** *auto*
qed (*auto*)

lemma *nsubst-ode*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes *ν*::'*sz state*
fixes *ν'*::'*sz state*
assumes *good-interp*:*is-interp I*
shows *osafe ODE* ⟹ *OadmitFO σ ODE U* ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ *ODE-sem I* (*OsubstFO ODE σ*) (*fst ν*)= *ODE-sem* (*adjointFO I σ ν*) *ODE* (*fst ν*)
proof (*induction rule*: *osafe.induct*)
case (*osafe-Var c*)
then show *?case* **unfolding** *OUadmitFO-def adjointFO-def* **by** *auto*
next
case (*osafe-Sing ϑ x*)
then show *?case* **apply** *auto*
using *nsubst-sterm'* [*of σ ϑ I* (*fst ν*) (*snd ν*)] **by** *auto*
next
case (*osafe-Prod ODE1 ODE2*) **then**
have *NO1*:*OadmitFO σ ODE1 U* **and** *NO2*:*OadmitFO σ ODE2 U*
unfolding *OUadmitFO-def* **by** *auto*
have *ODE-sem I* (*OsubstFO ODE1 σ*) (*fst ν*) = *ODE-sem* (*adjointFO I σ ν*) *ODE1* (*fst ν*)
ODE-sem I (*OsubstFO ODE2 σ*) (*fst ν*) = *ODE-sem* (*adjointFO I σ ν*) *ODE2* (*fst ν*) **using** *osafe-Prod.IH osafe-Prod.prems osafe-Prod.hyps*
using *NO1 NO2* **by** *auto*
then show *?case* **by** *auto*
qed

lemma *osubst-preserves-BVO*:
shows *BVO* (*OsubstFO ODE σ*) = *BVO ODE*
proof (*induction ODE*)
qed (*auto*)

lemma *osubst-preserves-ODE-vars*:
shows *ODE-vars I* (*OsubstFO ODE σ*) = *ODE-vars* (*adjointFO I σ ν*) *ODE*
proof (*induction ODE*)
qed (*auto simp add*: *adjointFO-def*)

lemma *osubst-preserves-semBV*:
shows *semBV I* (*OsubstFO ODE σ*) = *semBV* (*adjointFO I σ ν*) *ODE*
proof (*induction ODE*)
qed (*auto simp add*: *adjointFO-def*)

lemma *nsubst-mkv*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes *ν*::'*sz state*

```
  fixes ν'::'sz state
  assumes good-interp:is-interp I
  assumes NOU:OadmitFO σ ODE U
  assumes osafe:osafe ODE
  assumes frees:(⋀i. dsafe (σ i))
  shows (mk-v I (OsubstFO ODE σ) ν (fst ν'))
    = (mk-v (adjointFO I σ ν') ODE ν (fst ν'))
  apply(rule agree-UNIV-eq)
  using mk-v-agree[of adjointFO I σ ν' ODE ν fst ν']
  using mk-v-agree[of I OsubstFO ODE σ ν fst ν']
  unfolding Vagree-def
  using nsubst-ode[OF good-interp osafe NOU frees, of ν']
  apply auto
   subgoal for i
     apply(erule allE[where x=i])+
     apply(cases Inl i ∈ semBV I (OsubstFO ODE σ))
      using  osubst-preserves-ODE-vars
      by (metis (full-types))+
  subgoal for i
    apply(erule allE[where x=i])+
    apply(cases Inr i ∈ BVO ODE)
     using  osubst-preserves-ODE-vars
     by (metis (full-types))+
  done

lemma ODE-unbound-zero:
  fixes i
  shows Inl i ∉ BVO ODE ⟹ ODE-sem I ODE x $ i = 0
proof (induction ODE)
qed (auto)

lemma ODE-bound-effect:
  fixes s t sol ODE X b
  assumes s:s ∈ {0..t}
  assumes sol:(sol solves-ode (λ-. ODE-sem I ODE)) {0..t}  X
  shows Vagree (sol 0,b) (sol s, b) (-(BVO ODE))
proof -
  have ⋀i. Inl i ∉ BVO ODE ⟹  (∀ s. s ∈ {0..t} ⟶ sol s $ i = sol 0 $ i)
    apply auto
    apply (rule constant-when-zero)
        using s sol apply auto
    using ODE-unbound-zero solves-ode-subset
    by fastforce+
  then show Vagree (sol 0, b) (sol s, b) (- BVO ODE)
    unfolding Vagree-def
    using s  by (metis Compl-iff fst-conv  snd-conv)
qed

lemma NO-sub:OadmitFO σ ODE A ⟹ B ⊆ A ⟹ OadmitFO σ ODE B
```

by(*induction ODE*, *auto simp add*: *OUadmitFO-def*)

lemma *NO-to-NOU*:*OadmitFO σ ODE S* ⟹ *OUadmitFO σ ODE S*
by(*induction ODE*, *auto simp add*: *OUadmitFO-def*)

11.5 Substitution theorems for formulas and programs

lemma *nsubst-hp-fml*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
assumes *good-interp*:*is-interp I*
shows (*NPadmit σ α* ⟶ (*hpsafe α* ⟶ (∀ *i*. *dsafe* (*σ i*)) ⟶ (∀ *ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PsubstFO α σ*)) = ((*ν*, *ω*) ∈ *prog-sem* (*adjointFO I σ ν*) *α*)))) ∧
(*NFadmit σ φ* ⟶ (*fsafe φ* ⟶ (∀ *i*. *dsafe* (*σ i*)) ⟶ (∀ *ν*. (*ν* ∈ *fml-sem I* (*FsubstFO φ σ*)) = (*ν* ∈ *fml-sem* (*adjointFO I σ ν*) *φ*))))
proof (*induction rule*: *NPadmit-NFadmit.induct*)
case (*NPadmit-Pvar σ a*)
then show *?case* **unfolding** *adjointFO-def* **by** *auto*
next
case (*NPadmit-ODE σ ODE φ*) **then**
have *NOU*:*OadmitFO σ ODE* (*BVO ODE*)
and *NFA*:*NFadmit σ φ*
and *NFU*:*FUadmitFO σ φ* (*BVO ODE*)
and *fsafe*:*fsafe* (*FsubstFO φ σ*)
and *IH*:*fsafe φ* ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ (⋀*ν*. (*ν* ∈ *fml-sem I* (*FsubstFO φ σ*)) = (*ν* ∈ *fml-sem* (*adjointFO I σ ν*) *φ*))
and *osafe'*:*osafe* (*OsubstFO ODE σ*)
by *auto*
have *hpsafe* (*EvolveODE ODE φ*) ⟹ (⋀*i*. *dsafe* (*σ i*)) ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PsubstFO* (*EvolveODE ODE φ*) *σ*)) = ((*ν*, *ω*) ∈ *prog-sem* (*adjointFO I σ ν*) (*EvolveODE ODE φ*)))
proof −
assume *safe*:*hpsafe* (*EvolveODE ODE φ*)
then have *osafe*:*osafe ODE* **and** *fsafe*:*fsafe φ* **by** *auto*
assume *frees*:(⋀*i*. *dsafe* (*σ i*))
fix *ν ω*
show ((*ν*, *ω*) ∈ *prog-sem I* (*PsubstFO* (*EvolveODE ODE φ*) *σ*)) = ((*ν*, *ω*) ∈ *prog-sem* (*adjointFO I σ ν*) (*EvolveODE ODE φ*))
proof (*auto*)
fix *b*
and *sol* :: *real* ⇒(*real*, '*sz*) *vec*
and *t* :: *real*
assume *eq1*:*ν* = (*sol 0*, *b*)
assume *eq2*:*ω* = *mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) (*sol t*)
assume *t*:*0* ≤ *t*
assume *sol*:(*sol solves-ode* (λ*a*. *ODE-sem I* (*OsubstFO ODE σ*))) {*0*..*t*} {*x*. *mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) *x* ∈ *fml-sem I* (*FsubstFO φ σ*)}
have *agree-sem*:⋀*t*. *Vagree* (*mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) (*sol t*)) (*sol 0*, *b*) (− (*Inl* ‘ *ODE-vars I* (*OsubstFO ODE σ*) ∪ *Inr* ‘ *ODE-vars I* (*OsubstFO ODE σ*)))

subgoal for t
using *mk-v-agree*[*of I OsubstFO ODE* σ (*sol 0*, *b*) *sol t*] **unfolding** *Vagree-def* **apply** *auto*
done
done
have *bv-sub*:$(-BVO\ ODE) \subseteq -$ (*Inl* ‘ *ODE-vars I* (*OsubstFO ODE* σ) $\cup$ *Inr* ‘ *ODE-vars I* (*OsubstFO ODE* σ))
by(*induction ODE*, *auto*)
have *agree*:$\bigwedge t$. *Vagree* (*mk-v I* (*OsubstFO ODE* σ) (*sol 0*, *b*) (*sol t*)) (*sol 0*, *b*) (− *BVO ODE*)
using *agree-sub*[*OF bv-sub agree-sem*] **by** *auto*
— Necessary
have *mkv*:$\bigwedge t$. *mk-v I* (*OsubstFO ODE* σ) (*sol 0*, *b*) (*sol t*) = *mk-v* (*adjointFO I* σ (*sol t*, *b*)) *ODE* (*sol 0*, *b*) (*sol t*)
using *nsubst-mkv*[*OF good-interp NOU osafe frees*]
by *auto*
have *hmm*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *Vagree* (*sol 0*,*b*) (*sol s*, *b*) (−(*BVO ODE*))
using *ODE-bound-effect sol*
by (*metis osubst-preserves-BVO*)
have *FVT-sub*:($\bigcup y \in \{y.\ Inl\ (Inr\ y) \in SIGO\ ODE\}.\ FVT\ (\sigma\ y)$) $\subseteq$ (−(*BVO ODE*))
using *NOU NO-to-NOU OUadmitFO-def*
by *fastforce*
have *agrees*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *Vagree* (*sol 0*,*b*) (*sol s*, *b*) ($\bigcup y \in \{y.\ Inl\ (Inr\ y) \in SIGO\ ODE\}.\ FVT\ (\sigma\ y)$)
subgoal for s **using** *agree-sub*[*OF FVT-sub hmm*[*of s*]] **by** *auto* **done**
have $\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *mk-v* (*adjointFO I* σ (*sol s*, *b*)) *ODE* = *mk-v* (*adjointFO I* σ (*sol 0*, *b*)) *ODE*
subgoal for s
apply (*rule uadmit-mkv-ntadjoint*)
prefer *3*
using *NOU hmm*[*of s*] *NO-to-NOU* **unfolding** *OUadmitFO-def Vagree-def*
apply *fastforce*
using *frees good-interp osafe* **by** *auto*
done
then have *mkva*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *mk-v* (*adjointFO I* σ (*sol s*, *b*)) *ODE* (*sol 0*, *b*) (*sol s*) = *mk-v* (*adjointFO I* σ (*sol 0*, *b*)) *ODE* (*sol 0*, *b*) (*sol s*)
by *presburger*
have *main-eq*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *mk-v I* (*OsubstFO ODE* σ) (*sol 0*, *b*) (*sol s*) = *mk-v* (*adjointFO I* σ (*sol 0*, *b*)) *ODE* (*sol 0*, *b*) (*sol s*)
using *mkv mkva* **by** *auto*
note *mkvt* = *main-eq*[*of t*]
have *fml-eq1*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
(*mk-v I* (*OsubstFO ODE* σ) (*sol 0*, *b*) (*sol s*) $\in$ *fml-sem I* (*FsubstFO* φ σ))
= (*mk-v I* (*OsubstFO ODE* σ) (*sol 0*, *b*) (*sol s*) $\in$ *fml-sem* (*adjointFO I* σ (*mk-v I* (*OsubstFO ODE* σ) (*sol 0*, *b*) (*sol s*))) φ)
using *IH*[*OF fsafe frees*] **by** *auto*
have *fml-eq2*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$

((*mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) (*sol s*) ∈ *fml-sem* (*adjointFO I σ* (*mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) (*sol s*))) *φ*)
=(*mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) (*sol s*) ∈ *fml-sem* (*adjointFO I σ* (*sol 0*, *b*)) *φ*))
subgoal for *s*
using *NFU frees fsafe good-interp mk-v-agree osubst-preserves-ODE-vars uadmit-fml-ntadjoint*
using *agree* **by** *blast*
done
have *fml-eq3*:⋀*s*. *s* ∈ {*0*..*t*} ⟹
(*mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) (*sol s*) ∈ *fml-sem* (*adjointFO I σ* (*sol 0*, *b*)) *φ*) = (*mk-v* (*adjointFO I σ* (*sol 0*,*b*)) *ODE* (*sol 0*, *b*) (*sol s*) ∈ *fml-sem* (*adjointFO I σ* (*sol 0*, *b*)) *φ*)
using *main-eq* **by** *auto*
have *fml-eq*: ⋀*s*. *s* ∈ {*0*..*t*} ⟹
(*mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) (*sol s*) ∈ *fml-sem I* (*FsubstFO φ σ*))
= (*mk-v* (*adjointFO I σ* (*sol 0*,*b*)) *ODE* (*sol 0*, *b*) (*sol s*) ∈ *fml-sem* (*adjointFO I σ* (*sol 0*, *b*)) *φ*)
using *fml-eq1 fml-eq2 fml-eq3* **by** *meson*
have *sem-eq*:⋀*t*. *ODE-sem I* (*OsubstFO ODE σ*) (*sol t*) = *ODE-sem* (*adjointFO I σ* (*sol t*, *b*)) *ODE* (*sol t*)
subgoal for *t*
using *nsubst-ode*[*OF good-interp osafe NOU frees*, *of* (*sol t*,*b*)] **by** *auto*
done
have *sem-fact*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ *ODE-sem I* (*OsubstFO ODE σ*) (*sol s*) = *ODE-sem* (*adjointFO I σ* (*sol 0*, *b*)) *ODE* (*sol s*)
subgoal for *s*
using *nsubst-ode*[*OF good-interp osafe NOU frees*, *of* (*sol s*, *b*)]
uadmit-ode-ntadjoint'[*OF frees good-interp agrees*[*of s*] *osafe*]
by *auto*
done
have *sol'*:(*sol solves-ode* (λ-. *ODE-sem* (*adjointFO I σ* (*sol 0*, *b*)) *ODE*)) {*0*..*t*}
{*x*. *mk-v I* (*OsubstFO ODE σ*) (*sol 0*, *b*) *x* ∈ *fml-sem I* (*FsubstFO φ σ*)}
apply (*rule solves-ode-congI*)
apply (*rule sol*)
subgoal for *ta* **by** *auto*
subgoal for *ta* **by** (*rule sem-fact*[*of ta*])
subgoal by (*rule refl*)
subgoal by (*rule refl*)
done
have *sub*:⋀*s*. *s* ∈ {*0*..*t*}
⟹ *sol s* ∈ {*x*. (*mk-v* (*adjointFO I σ* (*sol 0*,*b*)) *ODE* (*sol 0*, *b*) *x* ∈ *fml-sem* (*adjointFO I σ* (*sol 0*, *b*)) *φ*)}
using *fml-eq rangeI t sol solves-ode-domainD* **by** *fastforce*
have *sol''*:(*sol solves-ode* (λ*c*. *ODE-sem* (*adjointFO I σ* (*sol 0*, *b*)) *ODE*)) {*0*..*t*}
{*x*. *mk-v* (*adjointFO I σ* (*sol 0*, *b*)) *ODE* (*sol 0*, *b*) *x* ∈ *fml-sem* (*adjointFO I σ*

```
(sol 0, b)) φ}
      apply (rule solves-odeI)
       subgoal using sol' solves-ode-vderivD by blast
      using sub by auto
    show ∃ sola. sol 0 = sola 0 ∧
      (∃ ta. mk-v I (OsubstFO ODE σ) (sol 0, b) (sol t) = mk-v (adjointFO I σ
(sol 0, b)) ODE (sola 0, b) (sola ta) ∧
          0 ≤ ta ∧
            (sola solves-ode (λa. ODE-sem (adjointFO I σ (sol 0, b)) ODE))
{0..ta}
            {x. mk-v (adjointFO I σ (sol 0, b)) ODE (sola 0, b) x ∈ fml-sem
(adjointFO I σ (sol 0, b)) φ})
      apply(rule exI[where x=sol])
      apply(rule conjI)
       subgoal by (rule refl)
      apply(rule exI[where x=t])
      apply(rule conjI)
       subgoal using  mkvt t by auto
      apply(rule conjI)
       subgoal by (rule t)
      subgoal by (rule sol'')
      done
  next
    fix b
     and sol::real ⇒ (real, 'sz) vec
     and t::real
    assume eq1:ν = (sol 0, b)
    assume eq2:ω = mk-v (adjointFO I σ (sol 0, b)) ODE (sol 0, b) (sol t)
    assume t:0 ≤ t
     assume sol:(sol solves-ode (λa. ODE-sem (adjointFO I σ (sol 0, b)) ODE))
{0..t}
      {x. mk-v (adjointFO I σ (sol 0, b)) ODE (sol 0, b) x ∈ fml-sem (adjointFO I
σ (sol 0, b)) φ}
    have agree-sem:⋀t. Vagree (mk-v I (OsubstFO ODE σ) (sol 0, b) (sol t)) (sol
0, b) (− (Inl ` ODE-vars I (OsubstFO ODE σ) ∪ Inr ` ODE-vars I (OsubstFO
ODE σ)))
      subgoal for t
      using mk-v-agree[of I OsubstFO ODE σ (sol 0, b) sol t] unfolding Vagree-def
apply auto
       done
      done
    have bv-sub:(−BVO ODE) ⊆ − (Inl ` ODE-vars I (OsubstFO ODE σ) ∪ Inr
` ODE-vars I (OsubstFO ODE σ))
      by(induction ODE, auto)
    have agree:⋀t. Vagree (mk-v I (OsubstFO ODE σ) (sol 0, b) (sol t)) (sol 0, b)
(− BVO ODE)
      using agree-sub[OF bv-sub agree-sem] by auto
    — Necessary
    have mkv:⋀t. mk-v I (OsubstFO ODE σ) (sol 0, b) (sol t) = mk-v (adjointFO
```

I σ (sol t, b)) ODE (sol 0, b) (sol t)
using *nsubst-mkv[OF good-interp NOU osafe frees]*
by *auto*
have *hmm:⋀s. s ∈ {0..t} ⟹ Vagree (sol 0,b) (sol s, b) (−(BVO ODE))*
using *ODE-bound-effect sol*
by *(metis osubst-preserves-ODE-vars)*
have *FVT-sub:(⋃ y∈{y. Inl (Inr y) ∈ SIGO ODE}. FVT (σ y)) ⊆ (−(BVO ODE))*
using *NOU NO-to-NOU* **unfolding** *OUadmitFO-def* **by** *fastforce*
have *agrees:⋀s. s ∈ {0..t} ⟹ Vagree (sol 0,b) (sol s, b) (⋃ y∈{y. Inl (Inr y) ∈ SIGO ODE}. FVT (σ y))*
subgoal for *s* **using** *agree-sub[OF FVT-sub hmm[of s]]* **by** *auto* **done**
have *⋀s. s ∈ {0..t} ⟹ mk-v (adjointFO I σ (sol s, b)) ODE = mk-v (adjointFO I σ (sol 0, b)) ODE*
subgoal for *s*
apply *(rule uadmit-mkv-ntadjoint)*
prefer *3*
using *NOU hmm[of s] NO-to-NOU* **unfolding** *OUadmitFO-def Vagree-def*
apply *fastforce*
using *frees good-interp osafe* **by** *auto*
done
then have *mkva:⋀s. s ∈ {0..t} ⟹ mk-v (adjointFO I σ (sol s, b)) ODE (sol 0, b) (sol s) = mk-v (adjointFO I σ (sol 0, b)) ODE (sol 0, b) (sol s)*
by *presburger*
have *main-eq:⋀s. s ∈ {0..t} ⟹ mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s) = mk-v (adjointFO I σ (sol 0, b)) ODE (sol 0, b) (sol s)*
using *mkv mkva* **by** *auto*
note *mkvt = main-eq[of t]*
have *fml-eq1:⋀s. s ∈ {0..t} ⟹*
(mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s) ∈ fml-sem I (FsubstFO φ σ))
= (mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s) ∈ fml-sem (adjointFO I σ (mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s))) φ)
using *IH[OF fsafe frees]* **by** *auto*
have *fml-eq2:⋀s. s ∈ {0..t} ⟹*
((mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s) ∈ fml-sem (adjointFO I σ (mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s))) φ)
=(mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s) ∈ fml-sem (adjointFO I σ (sol 0, b)) φ))
using *NFU frees fsafe good-interp mk-v-agree osubst-preserves-ODE-vars uadmit-fml-ntadjoint agree* **by** *blast*

have *fml-eq3:⋀s. s ∈ {0..t} ⟹*
(mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s) ∈ fml-sem (adjointFO I σ (sol 0, b)) φ) = (mk-v (adjointFO I σ (sol 0,b)) ODE (sol 0, b) (sol s) ∈ fml-sem (adjointFO I σ (sol 0, b)) φ)
using *main-eq* **by** *auto*
have *fml-eq: ⋀s. s ∈ {0..t} ⟹*
(mk-v I (OsubstFO ODE σ) (sol 0, b) (sol s) ∈ fml-sem I (FsubstFO φ σ))
= (mk-v (adjointFO I σ (sol 0,b)) ODE (sol 0, b) (sol s) ∈ fml-sem

(*adjointFO I σ (sol 0, b)) φ*)
using *fml-eq1 fml-eq2 fml-eq3* **by** *meson*
have *sem-eq*:⋀*t. ODE-sem I (OsubstFO ODE σ) (sol t) = ODE-sem (adjointFO I σ (sol t, b)) ODE (sol t)*
subgoal for *t*
using *nsubst-ode*[*OF good-interp osafe NOU frees, of (sol t,b)*] **by** *auto*
done
have *sem-fact*:⋀*s. s ∈ {0..t} ⟹ ODE-sem I (OsubstFO ODE σ) (sol s) = ODE-sem (adjointFO I σ (sol 0, b)) ODE (sol s)*
subgoal for *s*
using *nsubst-ode*[*OF good-interp osafe NOU frees, of (sol s, b)*]
uadmit-ode-ntadjoint'[*OF frees good-interp agrees*[*of s*] *osafe*]
by *auto*
done
have *sol'*:
(*sol solves-ode (λa. ODE-sem I (OsubstFO ODE σ))) {0..t} {x. mk-v (adjointFO I σ (sol 0, b)) ODE (sol 0, b) x ∈ fml-sem (adjointFO I σ (sol 0, b)) φ}*
apply (*rule solves-ode-congI*)
apply (*rule sol*)
subgoal for *ta* **by** *auto*
subgoal for *ta* **using** *sem-fact*[*of ta*] **by** *auto*
subgoal by (*rule refl*)
subgoal by (*rule refl*)
done
have *sub*:⋀*s. s ∈ {0..t}*
⟹ sol s ∈ {x. (mk-v (adjointFO I σ (sol 0,b)) ODE (sol 0, b) x ∈ fml-sem (adjointFO I σ (sol 0, b)) φ)}
using *fml-eq rangeI t sol solves-ode-domainD* **by** *fastforce*
have *sol''*:(*sol solves-ode (λa. ODE-sem I (OsubstFO ODE σ))) {0..t} {x. mk-v I (OsubstFO ODE σ) (sol 0, b) x ∈ fml-sem I (FsubstFO φ σ)}*
apply (*rule solves-odeI*)
subgoal using *sol' solves-ode-vderivD* **by** *blast*
using *sub fml-eq* **by** *blast*
show ∃*sola. sol 0 = sola 0 ∧*
(∃*ta. mk-v (adjointFO I σ (sol 0, b)) ODE (sol 0, b) (sol t) = mk-v I (OsubstFO ODE σ) (sola 0, b) (sola ta) ∧*
0 ≤ ta ∧
(*sola solves-ode (λa. ODE-sem I (OsubstFO ODE σ))) {0..ta} {x. mk-v I (OsubstFO ODE σ) (sola 0, b) x ∈ fml-sem I (FsubstFO φ σ)}*)
apply(*rule exI*[**where** *x=sol*])
apply(*rule conjI*)
subgoal by (*rule refl*)
apply(*rule exI*[**where** *x=t*])
apply(*rule conjI*)
subgoal using *t mkvt* **by** *auto*
apply(*rule conjI*)
subgoal by (*rule t*)
subgoal by (*rule sol''*)

done
qed
qed
then show *?case* **by** *auto*
next
case (*NPadmit-Assign* σ ϑ x)
then show *?case* **using** *nsubst-dterm*[*OF good-interp*, *of* σ ϑ] **by** *auto*
next
case (*NPadmit-DiffAssign* σ ϑ x)
then show *?case* **using** *nsubst-dterm*[*OF good-interp*, *of* σ ϑ] **by** *auto*
next
case (*NFadmit-Geq* σ $\vartheta 1$ $\vartheta 2$)
then show *?case*
using *nsubst-dterm*[*OF good-interp*, *of* σ $\vartheta 1$]
using *nsubst-dterm*[*OF good-interp*, *of* σ $\vartheta 2$] **by** *auto*
next
case (*NFadmit-Prop* σ *args f*)
assume *NTA*:$\bigwedge i$. *TadmitFO* σ (*args i*)
have $\bigwedge \nu$. *fsafe* (\$$\varphi$ *f args*) $\Longrightarrow$ ($\bigwedge i$. *dsafe* (σ *i*)) $\Longrightarrow$ ($\nu \in$ *fml-sem I* (*FsubstFO* (\$$\varphi$ *f args*) σ)) = ($\nu \in$ *fml-sem* (*adjointFO I* σ ν) (\$$\varphi$ *f args*))
proof −
fix ν
assume *safe*:*fsafe* (\$$\varphi$ *f args*)
from *safe* **have** *safes*:$\bigwedge i$. *dsafe* (*args i*) **using** *dfree-is-dsafe* **by** *auto*
assume *subFree*:($\bigwedge i$. *dsafe* (σ *i*))
have *vec-eq*:(χ *i*. *dterm-sem* (*adjointFO I* σ ν) (*args i*) ν) = (χ *i*. *dterm-sem I* (*TsubstFO* (*args i*) σ) ν)
apply(*rule vec-extensionality*)
using *nsubst-dterm*[*OF good-interp NTA safes subFree*] **by** *auto*
then show *?thesis* ν **unfolding** *adjointFO-def* **by** *auto*
qed
then show *?case* **by** *auto*
next
case (*NPadmit-Sequence* σ *a b*) **then**
have *PUA*:*PUadmitFO* σ *b* (*BVP* (*PsubstFO a* σ))
and *PA*:*NPadmit* σ *a*
and *IH1*:*hpsafe a* $\Longrightarrow$ ($\bigwedge i$. *dsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge \nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*PsubstFO a* σ)) = ((ν, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) *a*))
and *IH2*:*hpsafe b* $\Longrightarrow$ ($\bigwedge i$. *dsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge \nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*PsubstFO b* σ)) = ((ν, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) *b*))
and *hpsafeS*:*hpsafe* (*PsubstFO a* σ)
by *auto*
have *hpsafe* (*a* ;; *b*) $\Longrightarrow$ ($\bigwedge i$. *dsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge \nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*PsubstFO* (*a* ;; *b*) σ)) = ((ν, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) (*a* ;; *b*)))
proof −
assume *hpsafe*:*hpsafe* (*a* ;; *b*)
assume *ssafe*:($\bigwedge i$. *dsafe* (σ *i*))
from *hpsafe* **have** *safe1*:*hpsafe a* **and** *safe2*:*hpsafe b* **by** (*auto dest*: *hpsafe.cases*)
fix ν ω

have *agree*:$\bigwedge\mu.\ (\nu,\ \mu) \in$ *prog-sem I* (*PsubstFO a* σ) $\Longrightarrow$ *Vagree* ν μ (−*BVP*(*PsubstFO a* σ))
subgoal for μ
using *bound-effect*[*OF good-interp*, *of* (*PsubstFO a* σ) ν , *OF hpsafeS*] **by** *auto*
done
have *sem-eq*:$\bigwedge\mu.\ (\nu,\ \mu) \in$ *prog-sem I* (*PsubstFO a* σ) $\Longrightarrow$
$((\mu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ ν) *b*) =
$((\mu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ μ) *b*)
subgoal for μ
proof −
assume *assm*:$(\nu,\ \mu) \in$ *prog-sem I* (*PsubstFO a* σ)
show $((\mu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ ν) *b*) = $((\mu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ μ) *b*)
using *uadmit-prog-ntadjoint* [*OF PUA agree*[*OF assm*] *ssafe safe2 good-interp*]
by *auto*
qed
done
have $((\nu,\ \omega) \in$ *prog-sem I* (*PsubstFO* (*a* ;; *b*) σ)) = ($\exists$ μ. $(\nu,\ \mu) \in$ *prog-sem I* (*PsubstFO a* σ) $\wedge$ $(\mu,\ \omega) \in$ *prog-sem I* (*PsubstFO b* σ))
by *auto*
moreover have ... = ($\exists$ μ. $(\nu,\ \mu) \in$ *prog-sem I* (*PsubstFO a* σ) $\wedge$ $(\mu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ μ) *b*)
using *IH2*[*OF safe2 ssafe*] **by** *auto*
moreover have ... = ($\exists$ μ. $(\nu,\ \mu) \in$ *prog-sem I* (*PsubstFO a* σ) $\wedge$ $(\mu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ ν) *b*)
using *sem-eq* **by** *auto*
moreover have ... = ($\exists$ μ. $(\nu,\ \mu) \in$ *prog-sem* (*adjointFO I* σ ν) *a* $\wedge$ $(\mu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ ν) *b*)
using *IH1*[*OF safe1 ssafe*] **by** *auto*
ultimately
show $((\nu,\ \omega) \in$ *prog-sem I* (*PsubstFO* (*a* ;; *b*) σ)) = $((\nu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ ν) (*a* ;; *b*))
by *auto*
qed
then show *?case* **by** *auto*
next
case (*NPadmit-Loop* σ *a*) **then**
have *PA*:*NPadmit* σ *a*
and *PUA*:*PUadmitFO* σ *a* (*BVP* (*PsubstFO a* σ))
and *IH*:*hpsafe a* $\Longrightarrow$ ($\bigwedge i.$ *dsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge\nu\ \omega.$ $((\nu,\ \omega) \in$ *prog-sem I* (*PsubstFO a* σ)) = $((\nu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ ν) *a*))
and *hpsafeS*:*hpsafe* (*PsubstFO a* σ)
by *auto*
have *hpsafe* (*a***) $\Longrightarrow$ ($\bigwedge i.$ *dsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge\nu\ \omega.$ $((\nu,\ \omega) \in$ *prog-sem I* (*PsubstFO* (*a***) σ)) = $((\nu,\ \omega) \in$ *prog-sem* (*adjointFO I* σ ν) (*a***)))
proof −
assume *hpsafe* (*a***)

then have *hpsafe*:*hpsafe a* **by** (*auto dest*: *hpsafe.cases*)
assume *ssafe*:($\bigwedge i$. *dsafe* (σ *i*))
have *agree*:$\bigwedge \nu\ \mu$. (ν, μ) $\in$ *prog-sem I* (*PsubstFO a* σ) $\Longrightarrow$ *Vagree* ν μ ($-$*BVP*(*PsubstFO a* σ))
subgoal for ν μ
using *bound-effect*[*OF good-interp*, *of* (*PsubstFO a* σ) ν, *OF hpsafeS*]
by *auto*
done
have *sem-eq*:$\bigwedge \nu\ \mu\ \omega$. ($\nu$, μ) $\in$ *prog-sem I* (*PsubstFO a* σ) $\Longrightarrow$
((μ, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) *a*) =
((μ, ω) $\in$ *prog-sem* (*adjointFO I* σ μ) *a*)
subgoal for ν μ ω
proof $-$
assume *assm*:(ν, μ) $\in$ *prog-sem I* (*PsubstFO a* σ)
show ((μ, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) *a*) = ((μ, ω) $\in$ *prog-sem* (*adjointFO I* σ μ) *a*)
using *uadmit-prog-ntadjoint*[*OF PUA agree*[*OF assm*] *ssafe hpsafe good-interp*] **by** *auto*
qed
done
fix ν ω
have *UN-rule*:$\bigwedge$ *a S S'*. ($\bigwedge n$ *b*. (*a*,*b*) $\in$ *S n* $\longleftrightarrow$ (*a*,*b*) $\in$ *S' n*) $\Longrightarrow$ ($\bigwedge b$. (*a*,*b*) $\in$ ($\bigcup n$. *S n*) $\longleftrightarrow$ (*a*,*b*) $\in$ ($\bigcup n$. *S' n*))
by *auto*
have *eqL*:((ν, ω) $\in$ *prog-sem I* (*PsubstFO* (*a***) σ)) = ((ν, ω) $\in$ ($\bigcup n$. (*prog-sem I* (*PsubstFO a* σ)) ^^ *n*))
using *rtrancl-is-UN-relpow* **by** *auto*
moreover have *eachEq*:$\bigwedge n$. (($\bigwedge \nu\ \omega$. ((ν, ω) $\in$ (*prog-sem I* (*PsubstFO a* σ)) ^^ *n*) = ((ν, ω) $\in$ (*prog-sem* (*adjointFO I* σ ν) *a*)^^ *n*)))
proof $-$
fix *n*
show (($\bigwedge \nu\ \omega$. ((ν, ω) $\in$ (*prog-sem I* (*PsubstFO a* σ)) ^^ *n*) = ((ν, ω) $\in$ (*prog-sem* (*adjointFO I* σ ν) *a*)^^ *n*)))
proof (*induct n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*) **then**
have *IH2*:$\bigwedge \nu\ \omega$. ((ν, ω) $\in$ *prog-sem I* (*PsubstFO a* σ) ^^ *n*) = ((ν, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) *a* ^^ *n*)
by *auto*
have *relpow*:$\bigwedge R\ n$. *R* ^^ *Suc n* = *R O R* ^^ *n*
using *relpow.simps*(*2*) *relpow-commute* **by** *metis*
show *?case*
apply (*simp only*: *relpow*[*of n prog-sem I* (*PsubstFO a* σ)] *relpow*[*of n prog-sem* (*adjointFO I* σ ν) *a*])
apply(*unfold relcomp-unfold*)
apply *auto*
subgoal for *ab b*

```
          apply(rule exI[where x=ab])
          apply(rule exI[where x=b])
          using IH2 IH[OF hpsafe ssafe] sem-eq[of ν (ab,b) ω] apply auto
            using uadmit-prog-ntadjoint[OF PUA agree ssafe hpsafe good-interp]
IH[OF hpsafe ssafe]
           apply (metis (no-types, lifting))
            using uadmit-prog-ntadjoint[OF PUA agree ssafe hpsafe good-interp]
IH[OF hpsafe ssafe]
           apply (metis (no-types, lifting))
         done
        subgoal for ab b
          apply(rule exI[where x=ab])
          apply(rule exI[where x=b])
          using IH2 IH[OF hpsafe ssafe] sem-eq[of ν (ab,b) ω] apply auto
            using uadmit-prog-ntadjoint[OF PUA agree ssafe hpsafe good-interp]
IH[OF hpsafe ssafe]
           apply (metis (no-types, lifting))
            using uadmit-prog-ntadjoint[OF PUA agree ssafe hpsafe good-interp]
IH[OF hpsafe ssafe]
           apply (metis (no-types, lifting))
         done
       done
     qed
     qed
   moreover have ((ν, ω) ∈ (⋃n. (prog-sem I (PsubstFO a σ)) ^^ n)) = ((ν, ω)
∈ (⋃ n.(prog-sem (adjointFO I σ ν) a) ^^ n))
     apply(rule UN-rule)
     using eachEq by auto
   moreover have eqR:((ν, ω) ∈ prog-sem (adjointFO I σ ν) (a**)) = ((ν, ω) ∈
(⋃n. (prog-sem (adjointFO I σ ν) a) ^^ n))
      using rtrancl-is-UN-relpow by auto
    ultimately show ((ν, ω) ∈ prog-sem I (PsubstFO (a**) σ)) = ((ν, ω) ∈
prog-sem (adjointFO I σ ν) (a**))
     by auto
 qed
 then show ?case by auto
next
 case (NFadmit-Exists σ φ x)
 then have IH:fsafe φ ⟹ (⋀i. dsafe (σ i)) ⟹ (⋀ν. (ν ∈ fml-sem I (FsubstFO
φ σ)) = (ν ∈ fml-sem (adjointFO I σ ν) φ))
   and FUA:FUadmitFO σ φ {Inl x}
   by blast+
 have fsafe:fsafe (Exists x φ) ⟹ fsafe φ
   by (auto dest: fsafe.cases)
  have eq:fsafe (Exists x φ) ⟹ (⋀i. dsafe (σ i)) ⟹ (⋀ν. (ν ∈ fml-sem I
(FsubstFO (Exists x φ) σ)) = (ν ∈ fml-sem (adjointFO I σ ν)  (Exists x φ)))
 proof −
   assume fsafe:fsafe (Exists x φ)
   from fsafe have fsafe':fsafe φ by (auto dest: fsafe.cases)
```

assume *ssafe*:($\bigwedge i.$ *dsafe* (σ i))
fix ν
have *agree*:$\bigwedge r.$ *Vagree* ν (*repv* ν *x r*) (− {*Inl x*})
unfolding *Vagree-def* **by** *auto*
have *sem-eq*:$\bigwedge r.$ ((*repv* ν *x r*) ∈ *fml-sem* (*adjointFO I* σ (*repv* ν *x r*)) φ) =
((*repv* ν *x r*) ∈ *fml-sem* (*adjointFO I* σ ν) φ)
using *uadmit-fml-ntadjoint*[*OF FUA agree ssafe fsafe′ good-interp*] **by** *auto*
have (ν ∈ *fml-sem I* (*FsubstFO* (*Exists x* φ) σ)) = (∃ *r*. (*repv* ν *x r*) ∈ *fml-sem I* (*FsubstFO* φ σ))
by *auto*
moreover have ... = (∃ *r*. (*repv* ν *x r*) ∈ *fml-sem* (*adjointFO I* σ (*repv* ν *x r*)) φ)
using *IH*[*OF fsafe′ ssafe*] **by** *auto*
moreover have ... = (∃ *r*. (*repv* ν *x r*) ∈ *fml-sem* (*adjointFO I* σ ν) φ)
using *sem-eq* **by** *auto*
moreover have ... = (ν ∈ *fml-sem* (*adjointFO I* σ ν) (*Exists x* φ))
by *auto*
ultimately show (ν ∈ *fml-sem I* (*FsubstFO* (*Exists x* φ) σ)) = (ν ∈ *fml-sem* (*adjointFO I* σ ν) (*Exists x* φ))
by *auto*
qed
then show *?case* **by** *auto*
next
case (*NFadmit-Diamond* σ φ *a*) **then**
have *PA*:*NPadmit* σ *a*
and *FUA*:*FUadmitFO* σ φ (*BVP* (*PsubstFO a* σ))
and *IH1*:*fsafe* φ ⟹ ($\bigwedge i.$ *dsafe* (σ i)) ⟹ ($\bigwedge \nu.$ (ν ∈ *fml-sem I* (*FsubstFO* φ σ)) = (ν ∈ *fml-sem* (*adjointFO I* σ ν) φ))
and *IH2*:*hpsafe a* ⟹ ($\bigwedge i.$ *dsafe* (σ i)) ⟹ ($\bigwedge \nu$ $\omega.$ ((ν, ω) ∈ *prog-sem I* (*PsubstFO a* σ)) = ((ν, ω) ∈ *prog-sem* (*adjointFO I* σ ν) *a*))
and *hpsafeF*:*hpsafe* (*PsubstFO a* σ)
by *auto*
have *fsafe* (⟨ *a* ⟩ φ) ⟹ ($\bigwedge i.$ *dsafe* (σ i)) ⟹ ($\bigwedge \nu.$ (ν ∈ *fml-sem I* (*FsubstFO* (⟨ *a* ⟩ φ) σ)) = (ν ∈ *fml-sem* (*adjointFO I* σ ν) (⟨ *a* ⟩ φ)))
proof −
assume *fsafe*:*fsafe* (⟨ *a* ⟩ φ)
assume *ssafe*:($\bigwedge i.$ *dsafe* (σ i))
from *fsafe* **have** *fsafe′*:*fsafe* φ **and** *hpsafe*:*hpsafe a* **by** (*auto dest*: *fsafe.cases*)
fix ν
have *agree*:$\bigwedge \omega.$ (ν, ω) ∈ *prog-sem I* (*PsubstFO a* σ) ⟹ *Vagree* ν ω (−*BVP*(*PsubstFO a* σ))
using *bound-effect*[*OF good-interp*, *of* (*PsubstFO a* σ) ν, *OF hpsafeF*] **by** *auto*
have *sem-eq*:$\bigwedge \omega.$ (ν, ω) ∈ *prog-sem I* (*PsubstFO a* σ) ⟹
(ω ∈ *fml-sem* (*adjointFO I* σ ν) φ) =
(ω ∈ *fml-sem* (*adjointFO I* σ ω) φ)
using *uadmit-fml-ntadjoint* [*OF FUA agree ssafe fsafe′ good-interp*] **by** *auto*
have (ν ∈ *fml-sem I* (*FsubstFO* (⟨ *a* ⟩ φ) σ)) = (∃ ω. (ν, ω) ∈ *prog-sem I* (*PsubstFO a* σ) ∧ ω ∈ *fml-sem I* (*FsubstFO* φ σ))

by *auto*
moreover have ... = ($\exists$ ω. (ν, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) *a* $\wedge$ ω $\in$ *fml-sem* (*adjointFO I* σ ω) φ)
using *IH1*[*OF fsafe' ssafe*] *IH2*[*OF hpsafe ssafe*, *of* ν] **by** *auto*
moreover have ... = ($\exists$ ω. (ν, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) *a* $\wedge$ ω $\in$ *fml-sem* (*adjointFO I* σ ν) φ)
using *sem-eq IH2 hpsafe ssafe* **by** *blast*
moreover have ... = (ν $\in$ *fml-sem* (*adjointFO I* σ ν) ($\langle$ *a* $\rangle$ φ))
by *auto*
ultimately show *?thesis* ν **by** *auto*
qed
then show *?case* **by** *auto*
next
case (*NFadmit-Context* σ φ *C*) **then**
have *FA*:*NFadmit* σ φ
and *FUA*:*FUadmitFO* σ φ *UNIV*
and *IH*:*fsafe* φ $\Longrightarrow$ ($\bigwedge$*i*. *dsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge\nu$. (ν $\in$ *fml-sem I* (*FsubstFO* φ σ)) = (ν $\in$ *fml-sem* (*adjointFO I* σ ν) φ))
by *auto*
have *fsafe* (*InContext C* φ) $\Longrightarrow$
($\bigwedge$*i*. *dsafe* (σ *i*))$\Longrightarrow$ ($\bigwedge\nu$. (ν $\in$ *fml-sem I* (*FsubstFO* (*InContext C* φ) σ)) = (ν $\in$ *fml-sem* (*adjointFO I* σ ν) (*InContext C* φ)))
proof −
assume *safe*:*fsafe* (*InContext C* φ)
then have *fsafe*:*fsafe* φ **by** (*auto dest*: *fsafe.cases*)
assume *ssafe*:$\bigwedge$*i*. *dsafe* (σ *i*)
fix ν
have *Ieq*: *Contexts* (*adjointFO I* σ ν) *C* = *Contexts I C*
unfolding *adjointFO-def* **by** *auto*
have *IH'*:$\bigwedge\nu$. (ν $\in$ *fml-sem I* (*FsubstFO* φ σ)) = (ν $\in$ *fml-sem* (*adjointFO I* σ ν) φ)
using *IH*[*OF fsafe ssafe*] **by** *auto*
have *agree*:$\bigwedge\omega$. *Vagree* ν ω (−*UNIV*) **unfolding** *Vagree-def* **by** *auto*
have *adj-eq*:$\bigwedge\omega$. *fml-sem* (*adjointFO I* σ ν) φ = *fml-sem* (*adjointFO I* σ ω) φ
using *uadmit-fml-ntadjoint*[*OF FUA agree ssafe fsafe good-interp*] **by** *auto*
then have *sem*:*fml-sem I* (*FsubstFO* φ σ) = *fml-sem* (*adjointFO I* σ ν) φ
using *IH' agree adj-eq* **by** *auto*
show *?thesis* ν **using** *Ieq sem* **by** *auto*
qed
then show *?case* **by** *auto*
qed (*auto*)

lemma *nsubst-fml*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes ν::'*sz state*
assumes *good-interp*:*is-interp I*
assumes *NFA*:*NFadmit* σ φ
assumes *fsafe*:*fsafe* φ
assumes *frees*:($\forall$ *i*. *dsafe* (σ *i*))

shows ($\nu \in$ *fml-sem I* (*FsubstFO* φ σ)) = ($\nu \in$ *fml-sem* (*adjointFO I* σ ν) φ)
using *good-interp NFA fsafe frees*
by (*auto simp add*: *nsubst-hp-fml*)

lemma *nsubst-hp*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
fixes ν::'*sz state*
assumes *good-interp*:*is-interp I*
assumes *NPA*:*NPadmit* σ α
assumes *hpsafe*:*hpsafe* α
assumes *frees*:$\bigwedge i$. *dsafe* (σ *i*)
shows ((ν, ω) $\in$ *prog-sem I* (*PsubstFO* α σ)) = ((ν, ω) $\in$ *prog-sem* (*adjointFO I* σ ν) α)
using *good-interp NPA hpsafe frees nsubst-hp-fml* **by** *blast*

lemma *psubst-sterm*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
assumes *good-interp*:*is-interp I*
shows (*sterm-sem I* ϑ = *sterm-sem* (*PFadjoint I* σ) ϑ)
proof (*induction* ϑ)
qed (*auto simp add*: *PFadjoint-def*)

lemma *psubst-dterm*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
assumes *good-interp*:*is-interp I*
shows (*dsafe* ϑ $\Longrightarrow$ *dterm-sem I* ϑ = *dterm-sem* (*PFadjoint I* σ) ϑ)
proof (*induction* ϑ)
case (*Differential* ϑ)
assume *safe*:*dsafe* (*Differential* ϑ)
from *safe* **have** *free*:*dfree* ϑ **by** *auto*
assume *sem*:*dsafe* ϑ $\Longrightarrow$ *dterm-sem I* ϑ = *dterm-sem* (*PFadjoint I* σ) ϑ
have $\bigwedge\nu$. *frechet I* ϑ (*fst* ν) (*snd* ν) = *frechet* (*PFadjoint I* σ) ϑ (*fst* ν) (*snd* ν)
apply(*rule sterm-determines-frechet*)
using *good-interp free* **apply** *auto*
subgoal unfolding *is-interp-def PFadjoint-def* **by** *auto*
using *psubst-sterm*[*of I* ϑ] **by** *auto*
then show *?case*
by (*auto simp add*: *directional-derivative-def*)
qed (*auto simp add*: *PFadjoint-def*)

lemma *psubst-ode*:
assumes *good-interp*:*is-interp I*
shows *ODE-sem I ODE* = *ODE-sem* (*PFadjoint I* σ) *ODE*
proof (*induction ODE*)
case (*OVar x*)
then show *?case* **unfolding** *PFadjoint-def* **by** *auto*
next
case (*OSing x1a x2*)
then show *?case* **apply** *auto* **apply** (*rule ext*) **apply** (*rule vec-extensionality*)

using *psubst-sterm*[*OF good-interp*, *of x2* σ] **by** *auto*
next
case (*OProd ODE1 ODE2*)
then show *?case* **by** *auto*
qed

lemma *psubst-fml*:
fixes *I*::('*sf*, '*sc*, '*sz*) *interp*
assumes *good-interp*:*is-interp I*
shows (*PPadmit* σ α $\longrightarrow$ *hpsafe* α $\longrightarrow$ ($\forall$ *i*. *fsafe* (σ *i*)) $\longrightarrow$ ($\forall$ ν ω. (ν,ω) $\in$ *prog-sem I* (*PPsubst* α σ) = ((ν,ω) $\in$ *prog-sem* (*PFadjoint I* σ) α))) $\wedge$
(*PFadmit* σ φ $\longrightarrow$ *fsafe* φ $\longrightarrow$ ($\forall$ *i*. *fsafe* (σ *i*)) $\longrightarrow$ ($\forall$ ν. ν $\in$ *fml-sem I* (*PFsubst* φ σ) = (ν $\in$ *fml-sem* (*PFadjoint I* σ) φ)))
proof (*induction rule*: *PPadmit-PFadmit.induct*)
case (*PPadmit-ODE* σ φ *ODE*)
assume *PF*:*PFadmit* σ φ
assume *PFU*:*PFUadmit* σ φ (*BVO ODE*)
assume *IH*:*fsafe* φ $\longrightarrow$ ($\forall$ *i*. *fsafe* (σ *i*)) $\longrightarrow$ ($\forall \nu$. (ν $\in$ *fml-sem I* (*PFsubst* φ σ)) = (ν $\in$ *fml-sem* (*PFadjoint I* σ) φ))
have *hpsafe* (*EvolveODE ODE* φ) $\Longrightarrow$
($\forall$ *i*. *fsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge \nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*PPsubst* (*EvolveODE ODE* φ) σ)) = ((ν, ω) $\in$ *prog-sem* (*PFadjoint I* σ) (*EvolveODE ODE* φ)))
proof −
assume *safe*:*hpsafe* (*EvolveODE ODE* φ)
from *safe* **have** *fsafe*:*fsafe* φ **by** *auto*
assume *ssafe*:($\forall$ *i*. *fsafe* (σ *i*))
have *fml-eq*:$\bigwedge \nu$. (ν $\in$ *fml-sem I* (*PFsubst* φ σ)) = (ν $\in$ *fml-sem* (*PFadjoint I* σ) φ)
using *IH ssafe fsafe* **by** *auto*
fix ν ω
show ((ν, ω) $\in$ *prog-sem I* (*PPsubst* (*EvolveODE ODE* φ) σ)) = ((ν, ω) $\in$ *prog-sem* (*PFadjoint I* σ) (*EvolveODE ODE* φ))
apply *auto*
proof −
fix *b sol t*
assume *eq1*:ν = (*sol 0*, *b*)
and *eq2*:ω = *mk-v I ODE* (*sol 0*, *b*) (*sol t*)
and *t*:*0* $\leq$ *t*
and *sol*:(*sol solves-ode* (λ*a*. *ODE-sem I ODE*)) {*0..t*} {*x*. *mk-v I ODE* (*sol 0*, *b*) *x* $\in$ *fml-sem I* (*PFsubst* φ σ)}
have *var*:*ODE-vars I ODE* = *ODE-vars* (*PFadjoint I* σ) *ODE*
by(*induction ODE*, *auto simp add*: *PFadjoint-def*)
have *mkv-eq*:$\bigwedge$*s*. *s* $\in$ {*0..t*} $\Longrightarrow$ *mk-v I ODE* (*sol 0*, *b*) (*sol s*) = *mk-v* (*PFadjoint I* σ) *ODE* (*sol 0*, *b*) (*sol s*)
apply(*rule agree-UNIV-eq*)
unfolding *Vagree-def* **apply** *auto*
subgoal for *s i*
using *mk-v-agree*[*of I ODE* (*sol 0*, *b*) *sol s*] *mk-v-agree*[*of PFadjoint I* σ *ODE* (*sol 0*, *b*) *sol s*]

unfolding *Vagree-def var*
apply (*cases Inl i ∈ Inl ‘ ODE-vars I ODE*, *auto simp add*: *var*)
by *force*
subgoal for *s i*
using *mk-v-agree*[*of I ODE* (*sol 0*, *b*) *sol s*] *mk-v-agree*[*of PFadjoint I σ ODE* (*sol 0*, *b*) *sol s*]
unfolding *Vagree-def var*
apply (*cases Inr i ∈ Inr ‘ ODE-vars I ODE*, *auto simp add*: *var psubst-ode*)
using *psubst-ode*[*OF good-interp*, *of ODE σ*] **apply** *auto*
using *psubst-ode*[*OF good-interp*, *of ODE σ*] **by** *force*
done
have *sol′*:(*sol solves-ode* (*λ-. ODE-sem* (*PFadjoint I σ*) *ODE*)) {*0..t*}
{*x. mk-v I ODE* (*sol 0*, *b*) *x ∈ fml-sem I* (*PFsubst φ σ*)}
apply (*rule solves-ode-congI*)
apply (*rule sol*)
subgoal for *ta* **by** *auto*
subgoal for *ta* **using** *psubst-ode*[*OF good-interp*, *of ODE σ*] **by** *auto*
subgoal by (*rule refl*)
subgoal by (*rule refl*)
done
have *sub*:⋀*s. s ∈* {*0..t*}
⟹ *sol s ∈* {*x.* (*mk-v* (*PFadjoint I σ*) *ODE* (*sol 0*, *b*) *x ∈ fml-sem* (*PFadjoint I σ*) *φ*)}
subgoal for *s*
using *solves-ode-domainD*[*OF sol*, *of s*] *mkv-eq*[*of s*] *fml-eq*[*of mk-v* (*PFadjoint I σ*) *ODE* (*sol 0*, *b*) (*sol s*)]
by *auto*
done
have *sol′′*:(*sol solves-ode* (*λc. ODE-sem* (*PFadjoint I σ*) *ODE*)) {*0..t*}
{*x. mk-v* (*PFadjoint I σ*) *ODE* (*sol 0*, *b*) *x ∈ fml-sem* (*PFadjoint I σ*) *φ*}
apply (*rule solves-odeI*)
subgoal using *sol′ solves-ode-vderivD* **by** *blast*
using *sub* **by** *auto*
show∃ *sola. sol 0 = sola 0 ∧*
(∃ *ta. mk-v I ODE* (*sol 0*, *b*) (*sol t*) = *mk-v* (*PFadjoint I σ*) *ODE* (*sola 0*, *b*) (*sola ta*) ∧
0 ≤ ta ∧
(*sola solves-ode* (*λa. ODE-sem* (*PFadjoint I σ*) *ODE*)) {*0..ta*}
{*x. mk-v* (*PFadjoint I σ*) *ODE* (*sola 0*, *b*) *x ∈ fml-sem* (*PFadjoint I σ*) *φ*})
apply(*rule exI*[**where** *x*=*sol*])
apply(*rule conjI*)
apply(*rule refl*)
apply(*rule exI*[**where** *x*=*t*])
apply(*rule conjI*)
subgoal using *mkv-eq t* **by** *auto*
apply(*rule conjI*)
apply(*rule t*)
apply(*rule sol′′*)

```
    done
  next
    fix b sol t
    assume eq1:ν = (sol 0, b)
      and eq2:ω = mk-v (PFadjoint I σ) ODE (sol 0, b) (sol t)
      and t:0 ≤ t
       and sol:(sol solves-ode (λa. ODE-sem (PFadjoint I σ) ODE)) {0..t} {x.
mk-v (PFadjoint I σ) ODE (sol 0, b) x ∈ fml-sem (PFadjoint I σ) φ}
    have var:ODE-vars I ODE =  ODE-vars (PFadjoint I σ) ODE
      by(induction ODE, auto simp add: PFadjoint-def)
      have mkv-eq:⋀s. s ∈ {0..t} ⟹ mk-v I ODE (sol 0, b) (sol s) = mk-v
(PFadjoint I σ) ODE (sol 0, b) (sol s)
      apply(rule agree-UNIV-eq)
      unfolding Vagree-def apply auto
       subgoal for s i
         using mk-v-agree[of I ODE (sol 0, b) sol s] mk-v-agree[of PFadjoint I σ
ODE (sol 0, b) sol s]
         unfolding Vagree-def var
         apply (cases Inl i ∈ Inl ' ODE-vars I ODE, auto simp add: var)
          by force
      subgoal for s i
        using mk-v-agree[of I ODE (sol 0, b) sol s] mk-v-agree[of PFadjoint I σ
ODE (sol 0, b) sol s]
        unfolding Vagree-def var
       apply (cases Inr i ∈ Inr ' ODE-vars I ODE, auto simp add: var psubst-ode)
         using psubst-ode[OF good-interp, of ODE σ] apply auto
        using psubst-ode[OF good-interp, of ODE σ] by force
      done
    have sol':(sol solves-ode (λ-. ODE-sem I ODE)) {0..t}
      {x. mk-v (PFadjoint I σ) ODE (sol 0, b) x ∈ fml-sem (PFadjoint I σ) φ}
      apply (rule solves-ode-congI)
        apply (rule sol)
       subgoal for ta by auto
      subgoal for ta using psubst-ode[OF good-interp, of ODE σ] by auto
     subgoal by (rule refl)
      subgoal by (rule refl)
      done
    have sub:⋀s. s ∈ {0..t}
          ⟹ sol s ∈ {x. (mk-v  I ODE (sol 0, b) x ∈ fml-sem I (PFsubst φ σ))}
      subgoal for s
           using solves-ode-domainD[OF sol, of s] mkv-eq[of s] fml-eq[of mk-v
(PFadjoint I σ ) ODE (sol 0, b) (sol s)]
       by auto
      done
    have sol'':(sol solves-ode (λc. ODE-sem I ODE)) {0..t}
      {x. mk-v I ODE (sol 0, b) x ∈ fml-sem I (PFsubst φ σ)}
      apply (rule solves-odeI)
       subgoal using sol' solves-ode-vderivD by blast
      using sub by auto
```

```
    show ∃sola. sol 0 = sola 0 ∧
      (∃ta. mk-v (PFadjoint I σ) ODE (sol 0, b) (sol t) = mk-v I ODE (sola 0,
b) (sola ta) ∧
          0 ≤ ta ∧ (sola solves-ode (λa. ODE-sem I ODE)) {0..ta} {x. mk-v I
ODE (sola 0, b) x ∈ fml-sem I (PFsubst φ σ)})
      apply(rule exI[where x=sol])
      by (metis dual-order.refl intervalE mkv-eq sol'' t)
   qed
  qed
  then show ?case
   by auto
next
  case (PPadmit-Assign σ x ϑ)
  have hpsafe (x := ϑ) ⟹ (∀i. fsafe (σ i)) ⟹ (∀ ν ω. ((ν, ω) ∈ prog-sem I
(PPsubst (x := ϑ) σ)) = ((ν, ω) ∈ prog-sem (PFadjoint I σ) (x := ϑ)))
  proof −
   assume safe:hpsafe (x := ϑ)
   then have dsafe:dsafe ϑ by auto
   assume safes:(∀i. fsafe (σ i))
   show ?thesis
    using psubst-dterm[OF good-interp dsafe, of σ] by auto
  qed
  then show ?case by auto
next
  case (PPadmit-DiffAssign σ x ϑ)
  have hpsafe (DiffAssign x ϑ) ⟹ (∀i. fsafe (σ i)) ⟹ (∀ ν ω. ((ν, ω) ∈ prog-sem
I (PPsubst (DiffAssign x ϑ) σ)) = (((ν, ω) ∈ prog-sem (PFadjoint I σ) (DiffAssign
x ϑ))))
  proof −
   assume safe:hpsafe (DiffAssign x ϑ)
   then have dsafe:dsafe ϑ by auto
   assume safes:(∀i. fsafe (σ i))
   show ?thesis
    using psubst-dterm[OF good-interp dsafe, of σ] by auto
  qed
  then show ?case by auto
next
  case (PFadmit-Geq σ ϑ1 ϑ2) then
  have fsafe (Geq ϑ1 ϑ2) ⟹ (∀i. fsafe (σ i)) ⟹ (∀ ν. (ν ∈ fml-sem I (PFsubst
(Geq ϑ1 ϑ2) σ)) = (ν ∈ fml-sem (PFadjoint I σ) (Geq ϑ1 ϑ2)))
  proof −
   assume safe:fsafe (Geq ϑ1 ϑ2)
   then have safe1:dsafe ϑ1
    and safe2:dsafe ϑ2 by auto
   assume safes:(∀i. fsafe (σ i))
   show ?thesis
    using psubst-dterm[OF good-interp safe1, of σ] psubst-dterm[OF good-interp
safe2, of σ] by auto
  qed
```

then show *?case* **by** *auto*
next
case (*PFadmit-Prop* σ *p args*) **then**
have *fsafe* (*Prop p args*) $\Longrightarrow$ ($\bigwedge i$. *fsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge \nu$.($\nu \in$ *fml-sem I* (*PFsubst* (\$$\varphi$ *p args*) σ)) = ($\nu \in$ *fml-sem* (*PFadjoint I* σ) (\$$\varphi$ *p args*)))
proof −
assume *safe*:*fsafe* (*Prop p args*) **and** *ssafe*: ($\bigwedge i$. *fsafe* (σ *i*))
fix ν
from *safe* **have** *safes*:$\bigwedge i$. *dsafe* (*args i*) **using** *dfree-is-dsafe* **by** *auto*
have *Ieq*:*Predicates I p* = *Predicates* (*PFadjoint I* σ) *p*
unfolding *PFadjoint-def* **by** *auto*
have *vec*:(χ *i*. *dterm-sem I* (*args i*) ν) = (χ *i*. *dterm-sem* (*PFadjoint I* σ) (*args i*) ν)
apply(*auto simp add*: *vec-eq-iff*)
subgoal for *i* **using** *safes*[*of i*]
by (*metis good-interp psubst-dterm*)
done
show *?thesis* ν **using** *Ieq vec* **by** *auto*
qed
then show *?case* **by** *auto*
next
case (*PPadmit-Sequence* σ *a b*) **then**
have *PUA*:*PPUadmit* σ *b* (*BVP* (*PPsubst a* σ))
and *PA*:*PPadmit* σ *a*
and *IH1*:*hpsafe a* $\Longrightarrow$ ($\bigwedge i$. *fsafe* (σ *i*)) $\Longrightarrow$ ($\forall$ ν ω. ((ν, ω) $\in$ *prog-sem I* (*PPsubst a* σ)) = ((ν, ω) $\in$ *prog-sem* (*PFadjoint I* σ) *a*))
and *IH2*:*hpsafe b* $\Longrightarrow$ ($\bigwedge i$. *fsafe* (σ *i*)) $\Longrightarrow$ ($\forall$ ν ω. ((ν, ω) $\in$ *prog-sem I* (*PPsubst b* σ)) = ((ν, ω) $\in$ *prog-sem* (*PFadjoint I* σ) *b*))
and *substSafe*:*hpsafe* (*PPsubst a* σ)
by *auto*
have *hpsafe* (*a* ;; *b*) $\Longrightarrow$ ($\bigwedge i$. *fsafe* (σ *i*)) $\Longrightarrow$ ($\bigwedge$ ν ω. ((ν, ω) $\in$ *prog-sem I* (*PPsubst* (*a* ;; *b*) σ)) = ((ν, ω) $\in$ *prog-sem* (*PFadjoint I* σ) (*a* ;; *b*)))
proof −
assume *hpsafe*:*hpsafe* (*a* ;; *b*)
assume *ssafe*:($\bigwedge i$. *fsafe* (σ *i*))
from *hpsafe* **have** *safe1*:*hpsafe a* **and** *safe2*:*hpsafe b* **by** (*auto dest*: *hpsafe.cases*)
fix ν ω
have *agree*:$\bigwedge \mu$. (ν, μ) $\in$ *prog-sem I* (*PPsubst a* σ) $\Longrightarrow$ *Vagree* ν μ (−*BVP*(*PPsubst a* σ))
subgoal for μ
using *bound-effect*[*OF good-interp*, *of* (*PPsubst a* σ) ν, *OF substSafe*] **by** *auto*
done
have *sem-eq*:$\bigwedge \mu$. (ν, μ) $\in$ *prog-sem I* (*PPsubst a* σ) $\Longrightarrow$
((μ, ω) $\in$ *prog-sem* (*PFadjoint I* σ) *b*) =
((μ, ω) $\in$ *prog-sem* (*PFadjoint I* σ) *b*)
subgoal for μ
proof −
assume *assm*:(ν, μ) $\in$ *prog-sem I* (*PPsubst a* σ)

show $((\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ b) = ((\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ b)$
using *PUA agree*[*OF assm*] *safe2 ssafe good-interp* **by** *auto*
qed
done
have $((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ (a\ ;;\ b)\ \sigma)) = (\exists\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ a\ \sigma) \wedge (\mu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ b\ \sigma))$
by *auto*
moreover have $... = (\exists\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ a\ \sigma) \wedge (\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ b)$
using *IH2*[*OF safe2 ssafe*] **by** *blast*
moreover have $... = (\exists\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ a \wedge (\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ b)$
using *IH1*[*OF safe1 ssafe*] *sem-eq* **by** *blast*
ultimately
show $((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ (a\ ;;\ b)\ \sigma)) = ((\nu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ (a\ ;;\ b))$
by *auto*
qed
then show *?case* **by** *auto*
next
case (*PPadmit-Loop* σ a) **then**
have *PA*:*PPadmit* σ a
and *PUA*:*PPUadmit* σ a $(\mathit{BVP}\ (\mathit{PPsubst}\ a\ \sigma))$
and *IH*:*hpsafe* $a \Longrightarrow (\bigwedge i.\ \mathit{fsafe}\ (\sigma\ i)) \Longrightarrow (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ a\ \sigma)) = ((\nu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ a))$
and *substSafe*:*hpsafe* $(\mathit{PPsubst}\ a\ \sigma)$
by *auto*
have *hpsafe* $(a{**}) \Longrightarrow (\bigwedge i.\ \mathit{fsafe}\ (\sigma\ i)) \Longrightarrow (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ (a{**})\ \sigma)) = ((\nu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ (a{**})))$
proof −
assume *hpsafe* $(a{**})$
then have *hpsafe*:*hpsafe* a **by** (*auto dest*: *hpsafe.cases*)
assume *ssafe*:$\bigwedge i.\ \mathit{fsafe}\ (\sigma\ i)$
have *agree*:$\bigwedge \nu\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ a\ \sigma) \Longrightarrow \mathit{Vagree}\ \nu\ \mu\ (-\mathit{BVP}(\mathit{PPsubst}\ a\ \sigma))$
subgoal for ν μ
using *bound-effect*[*OF good-interp*, *of* $(\mathit{PPsubst}\ a\ \sigma)$ ν, *OF substSafe*] **by** *auto*
done
fix ν ω
have *UN-rule*:$\bigwedge$ a S S'. $(\bigwedge n\ b.\ (a,b) \in S\ n \longleftrightarrow (a,b) \in S'\ n) \Longrightarrow (\bigwedge b.\ (a,b) \in (\bigcup n.\ S\ n) \longleftrightarrow (a,b) \in (\bigcup n.\ S'\ n))$
by *auto*
have *eqL*:$((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ (a{**})\ \sigma)) = ((\nu, \omega) \in (\bigcup n.\ (\mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ a\ \sigma))\ ^{\frown\frown}\ n))$
using *rtrancl-is-UN-relpow* **by** *auto*
moreover have *eachEq*:$\bigwedge n.\ ((\bigwedge \nu\ \omega.\ ((\nu, \omega) \in (\mathit{prog\text{-}sem}\ I\ (\mathit{PPsubst}\ a\ \sigma))\ ^{\frown\frown}\ n) = ((\nu, \omega) \in (\mathit{prog\text{-}sem}\ (\mathit{PFadjoint}\ I\ \sigma)\ a)^{\frown\frown}\ n)))$

```
  proof −
    fix n
      show ((⋀ν ω. ((ν, ω) ∈ (prog-sem I (PPsubst a σ)) ^^ n) = ((ν, ω) ∈
(prog-sem (PFadjoint I σ) a) ^^ n)))
    proof (induct n)
      case 0
      then show ?case by auto
    next
      case (Suc n) then
       have IH2:⋀ν ω. ((ν, ω) ∈ prog-sem I (PPsubst a σ) ^^ n) = ((ν, ω) ∈
prog-sem (PFadjoint I σ) a ^^ n)
        by auto
      have relpow:⋀R n. R ^^ Suc n = R O R ^^ n
        using relpow.simps(2) relpow-commute by metis
      show ?case
          apply (simp only: relpow[of n prog-sem I (PPsubst a σ)] relpow[of n
prog-sem (PFadjoint I σ) a])
       apply(unfold relcomp-unfold)
       apply auto
        subgoal for ab b
          apply(rule exI[where x=ab])
          apply(rule exI[where x=b])
          using IH2 IH[OF hpsafe ssafe] by auto
       subgoal for ab b
         apply(rule exI[where x=ab])
         apply(rule exI[where x=b])
         using IH2 IH[OF hpsafe ssafe] by auto
      done
    qed
  qed
  moreover have ((ν, ω) ∈ (⋃n. (prog-sem I (PPsubst a σ)) ^^ n)) = ((ν, ω)
∈ (⋃ n.(prog-sem (PFadjoint I σ) a) ^^ n))
    apply(rule UN-rule)
    using eachEq by auto
   moreover have eqR:((ν, ω) ∈ prog-sem (PFadjoint I σ) (a**)) = ((ν, ω) ∈
(⋃n. (prog-sem (PFadjoint I σ) a) ^^ n))
     using rtrancl-is-UN-relpow by auto
 ultimately show ((ν, ω) ∈ prog-sem I (PPsubst (a**) σ)) = ((ν, ω) ∈ prog-sem
(PFadjoint I σ) (a**))
    by auto
 qed
 then show ?case by auto
next
next
 case (PFadmit-Context σ φ C) then
 have FA:PFadmit σ φ
  and FUA:PFUadmit σ φ UNIV
  and IH:fsafe φ ⟹ (⋀i. fsafe (σ i)) ⟹ (⋀ν. (ν ∈ fml-sem I (PFsubst φ σ))
= (ν ∈ fml-sem (PFadjoint I σ) φ))
```

```
    by auto
  have fsafe (InContext C φ) ⟹
         (⋀i. fsafe (σ i)) ⟹ (⋀ν. (ν ∈ fml-sem I (PFsubst (InContext C φ) σ))
= (ν ∈ fml-sem (PFadjoint I σ) (InContext C φ)))
  proof −
    assume safe:fsafe (InContext C φ)
    then have fsafe:fsafe φ by (auto dest: fsafe.cases)
    assume ssafe:(⋀i. fsafe (σ i))
    fix ν :: (real, 'sz) vec × (real, 'sz) vec
    have IH':⋀ν. (ν ∈ fml-sem I (PFsubst φ σ)) = (ν ∈ fml-sem (PFadjoint I σ)
φ)
      using IH[OF fsafe ssafe] by auto
    have agree:⋀ω. Vagree ν ω (−UNIV) unfolding Vagree-def by auto
    then have sem:fml-sem I (PFsubst φ σ) =  fml-sem (PFadjoint I σ) φ
      using IH' agree  by auto
    show ?thesis ν  using sem
      apply auto
      apply(cases C)
        unfolding PFadjoint-def apply auto
      apply(cases C)
       by auto
  qed
  then show ?case by auto
qed (auto simp add: PFadjoint-def)

lemma subst-ode:
  fixes I:: ('sf, 'sc, 'sz) interp and ν :: 'sz state
  assumes good-interp:is-interp I
  shows osafe ODE ⟹
        ssafe σ ⟹
        Oadmit σ ODE (BVO ODE) ⟹
        ODE-sem I (Osubst ODE σ) (fst ν) = ODE-sem (adjoint I σ ν) ODE (fst
ν)
proof (induction rule: osafe.induct)
  case (osafe-Var c)
  then show ?case unfolding adjoint-def by (cases SODEs σ c, auto)
next
  case (osafe-Sing ϑ x)
  then show ?case
    using subst-sterm [of  σ ϑ I ν]
    unfolding ssafe-def by auto
next
  case (osafe-Prod ODE1 ODE2) then
 have NOU1:Oadmit σ ODE1  (BVO (OProd ODE1 ODE2)) and NOU2:Oadmit
σ ODE2  (BVO (OProd ODE1 ODE2))
    by auto
  have TUA-sub:⋀σ ϑ A B. TUadmit σ ϑ B ⟹ A ⊆ B ⟹ TUadmit σ ϑ A
    unfolding TUadmit-def by auto
  have OA-sub:⋀ODE A B. Oadmit σ ODE B ⟹ A ⊆ B ⟹ Oadmit σ ODE A
```

```
    subgoal for ODE A B
    proof (induction rule: Oadmit.induct)
      case (Oadmit-Var σ c U)
      then show ?case by auto
    next
      case (Oadmit-Sing σ ϑ U x)
      then show ?case using TUA-sub[of σ ϑ U A] by auto
    next
      case (Oadmit-Prod σ ODE1 U ODE2)
      then show ?case by auto
    qed
    done
  have sub1:(BVO ODE1) ⊆ (BVO (OProd ODE1 ODE2))
    by auto
  have sub2: (BVO ODE2) ⊆ (BVO (OProd ODE1 ODE2))
    by auto
  have ODE-sem I (Osubst ODE1 σ) (fst ν) = ODE-sem (adjoint I σ ν) ODE1
(fst ν)
    ODE-sem I (Osubst ODE2 σ) (fst ν) = ODE-sem (adjoint I σ ν) ODE2 (fst
ν) using osafe-Prod.IH osafe-Prod.prems osafe-Prod.hyps
    using OA-sub[OF NOU1 sub1] OA-sub[OF NOU2 sub2] by auto
  then show ?case by auto
qed

lemma osubst-eq-ODE-vars: ODE-vars I (Osubst ODE σ) = ODE-vars (adjoint I
σ ν) ODE
proof (induction ODE)
  case (OVar x)
  then show ?case by (cases SODEs σ x, auto simp add: adjoint-def)
qed (auto)

lemma subst-semBV:semBV (adjoint I σ ν') ODE = (semBV I (Osubst ODE
σ))
proof (induction ODE)
  case (OVar x)
  then show ?case by (cases SODEs σ x, auto simp add: adjoint-def)
qed (auto)

lemma subst-mkv:
  fixes I::('sf, 'sc, 'sz) interp
  fixes ν::'sz state
  fixes ν'::'sz state
  assumes good-interp:is-interp I
  assumes NOU:Oadmit σ ODE (BVO ODE)
  assumes osafe:osafe ODE
  assumes frees:ssafe σ
  shows (mk-v I (Osubst ODE σ) ν (fst ν'))
    = (mk-v (adjoint I σ ν') ODE ν (fst ν'))
  apply(rule agree-UNIV-eq)
```

using *mk-v-agree*[*of adjoint I σ ν′ ODE ν fst ν′*]
using *mk-v-agree*[*of I Osubst ODE σ ν fst ν′*]
unfolding *Vagree-def*
using *subst-ode*[*OF good-interp osafe frees NOU, of ν′*]
apply *auto*
subgoal for *i*
apply(*erule allE*[**where** *x*=*i*])+
apply(*cases Inl i ∈ Inl ‘ ODE-vars* (*adjoint I σ ν′*) *ODE*)
using *osubst-eq-ODE-vars*[*of I ODE σ ν′*]
apply *force*
proof −
assume *ODE-sem I* (*Osubst ODE σ*) (*fst ν′*) = *ODE-sem* (*local.adjoint I σ ν′*) *ODE* (*fst ν′*)
Inl i ∉ Inl ‘ ODE-vars (*local.adjoint I σ ν′*) *ODE ∧ Inl i ∉ Inr ‘ ODE-vars* (*local.adjoint I σ ν′*) *ODE* ⟶
fst (*mk-v* (*local.adjoint I σ ν′*) *ODE ν* (*fst ν′*)) $ *i* = *fst ν* $ *i*
Inl i ∉ Inl ‘ ODE-vars I (*Osubst ODE σ*) *∧ Inl i ∉ Inr ‘ ODE-vars I* (*Osubst ODE σ*) ⟶
fst (*mk-v I* (*Osubst ODE σ*) *ν* (*fst ν′*)) $ *i* = *fst ν* $ *i*
Inl i ∉ Inl ‘ ODE-vars (*local.adjoint I σ ν′*) *ODE*
then show
fst (*mk-v I* (*Osubst ODE σ*) *ν* (*fst ν′*)) $ *i* = *fst* (*mk-v* (*local.adjoint I σ ν′*) *ODE ν* (*fst ν′*)) $ *i*
using *osubst-eq-ODE-vars*[*of I ODE σ ν′*] **by** *force*
qed
subgoal for *i*
apply(*erule allE*[**where** *x*=*i*])+
apply(*cases Inr i ∈ Inr ‘ ODE-vars* (*adjoint I σ ν′*) *ODE*)
using *osubst-eq-ODE-vars*[*of I ODE σ ν′*]
apply *force*
proof −
assume *ODE-sem I* (*Osubst ODE σ*) (*fst ν′*) = *ODE-sem* (*local.adjoint I σ ν′*) *ODE* (*fst ν′*)
Inr i ∉ Inl ‘ ODE-vars (*local.adjoint I σ ν′*) *ODE ∧ Inr i ∉ Inr ‘ ODE-vars* (*local.adjoint I σ ν′*) *ODE* ⟶
snd (*mk-v* (*local.adjoint I σ ν′*) *ODE ν* (*fst ν′*)) $ *i* = *snd ν* $ *i*
Inr i ∉ Inl ‘ ODE-vars I (*Osubst ODE σ*) *∧ Inr i ∉ Inr ‘ ODE-vars I* (*Osubst ODE σ*) ⟶
snd (*mk-v I* (*Osubst ODE σ*) *ν* (*fst ν′*)) $ *i* = *snd ν* $ *i*
Inr i ∉ Inr ‘ ODE-vars (*local.adjoint I σ ν′*) *ODE*
then show *snd* (*mk-v I* (*Osubst ODE σ*) *ν* (*fst ν′*)) $ *i* = *snd* (*mk-v* (*local.adjoint I σ ν′*) *ODE ν* (*fst ν′*)) $ *i*
using *osubst-eq-ODE-vars*[*of I ODE σ ν′*] **by** *force*
qed
done

lemma *subst-fml-hp*:
fixes *I*::(*′sf*, *′sc*, *′sz*) *interp*
assumes *good-interp*:*is-interp I*

shows
(*Padmit* σ α $\longrightarrow$
(*hpsafe* α $\longrightarrow$
ssafe σ $\longrightarrow$
($\forall$ ν ω. ((ν, ω) $\in$ *prog-sem I* (*Psubst* α σ)) = ((ν, ω) $\in$ *prog-sem* (*adjoint I* σ ν) α))))
$\wedge$
(*Fadmit* σ φ $\longrightarrow$
(*fsafe* φ $\longrightarrow$
ssafe σ $\longrightarrow$
($\forall$ ν. (ν $\in$ *fml-sem I* (*Fsubst* φ σ)) = (ν $\in$ *fml-sem* (*adjoint I* σ ν) φ))))
proof (*induction rule*: *Padmit-Fadmit.induct*)
case (*Padmit-Pvar* σ *a*) **then**
have *hpsafe* (\$$\alpha$ *a*) $\Longrightarrow$ *ssafe* σ $\Longrightarrow$ ($\bigwedge\nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*Psubst* (\$$\alpha$ *a*) σ)) = ((ν, ω) $\in$ *prog-sem* (*local.adjoint I* σ ν) (\$$\alpha$ *a*)))
apply (*cases SPrograms* σ *a*)
unfolding *adjoint-def* **by** *auto*
then show *?case* **by** *auto*
next
case (*Padmit-Sequence* σ *a b*) **then**
have *PUA*:*PUadmit* σ *b* (*BVP* (*Psubst a* σ))
and *PA*:*Padmit* σ *a*
and *IH1*:*hpsafe a* $\Longrightarrow$ *ssafe* σ $\Longrightarrow$ ($\bigwedge\nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*Psubst a* σ)) = ((ν, ω) $\in$ *prog-sem* (*local.adjoint I* σ ν) *a*))
and *IH2*:*hpsafe b* $\Longrightarrow$ *ssafe* σ $\Longrightarrow$ ($\bigwedge\nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*Psubst b* σ)) = ((ν, ω) $\in$ *prog-sem* (*local.adjoint I* σ ν) *b*))
and *substSafe*:*hpsafe* (*Psubst a* σ)
by *auto*
have *hpsafe* (*a* ;; *b*) $\Longrightarrow$ *ssafe* σ $\Longrightarrow$ ($\bigwedge\nu$ ω. ((ν, ω) $\in$ *prog-sem I* (*Psubst* (*a* ;; *b*) σ)) = ((ν, ω) $\in$ *prog-sem* (*local.adjoint I* σ ν) (*a* ;; *b*)))
proof −
assume *hpsafe*:*hpsafe* (*a* ;; *b*)
assume *ssafe*:*ssafe* σ
from *hpsafe* **have** *safe1*:*hpsafe a* **and** *safe2*:*hpsafe b* **by** (*auto dest*: *hpsafe.cases*)
fix ν ω
have *agree*:$\bigwedge\mu$. (ν, μ) $\in$ *prog-sem I* (*Psubst a* σ) $\Longrightarrow$ *Vagree* ν μ (−*BVP*(*Psubst a* σ))
subgoal for μ
using *bound-effect*[*OF good-interp*, *of* (*Psubst a* σ) ν, *OF substSafe*] **by** *auto*
done
have *sem-eq*:$\bigwedge\mu$. (ν, μ) $\in$ *prog-sem I* (*Psubst a* σ) $\Longrightarrow$
((μ, ω) $\in$ *prog-sem* (*local.adjoint I* σ ν) *b*) =
((μ, ω) $\in$ *prog-sem* (*local.adjoint I* σ μ) *b*)
subgoal for μ
proof −
assume *assm*:(ν, μ) $\in$ *prog-sem I* (*Psubst a* σ)
show ((μ, ω) $\in$ *prog-sem* (*local.adjoint I* σ ν) *b*) = ((μ, ω) $\in$ *prog-sem* (*local.adjoint I* σ μ) *b*)
using *uadmit-prog-adjoint*[*OF PUA agree*[*OF assm*] *safe2 ssafe good-interp*]

by *auto*
qed
done
have $((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ (a\ ;;\ b)\ \sigma)) = (\exists\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ a\ \sigma) \wedge (\mu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ b\ \sigma))$
by *auto*
moreover have $\ldots = (\exists\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ a\ \sigma) \wedge (\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{adjoint}\ I\ \sigma\ \mu)\ b)$
using *IH2*[*OF safe2 ssafe*] **by** *auto*
moreover have $\ldots = (\exists\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ a\ \sigma) \wedge (\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{adjoint}\ I\ \sigma\ \nu)\ b)$
using *sem-eq* **by** *auto*
moreover have $\ldots = (\exists\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ (\mathit{adjoint}\ I\ \sigma\ \nu)\ a \wedge (\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{adjoint}\ I\ \sigma\ \nu)\ b)$
using *IH1*[*OF safe1 ssafe*] **by** *auto*
ultimately
show $((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ (a\ ;;\ b)\ \sigma)) = ((\nu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{local.adjoint}\ I\ \sigma\ \nu)\ (a\ ;;\ b))$
by *auto*
qed
then show *?case* **by** *auto*
next
case (*Padmit-Loop* σ a) **then**
have *PA*:*Padmit* σ a
and *PUA*:*PUadmit* σ a $(\mathit{BVP}\ (\mathit{Psubst}\ a\ \sigma))$
and *IH*:*hpsafe* $a \Longrightarrow \mathit{ssafe}\ \sigma \Longrightarrow (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ a\ \sigma)) = ((\nu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{local.adjoint}\ I\ \sigma\ \nu)\ a))$
and *substSafe*:*hpsafe* $(\mathit{Psubst}\ a\ \sigma)$
by *auto*
have *hpsafe* $(a{*}{*}) \Longrightarrow \mathit{ssafe}\ \sigma \Longrightarrow (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ (a{*}{*})\ \sigma)) = ((\nu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{local.adjoint}\ I\ \sigma\ \nu)\ (a{*}{*})))$
proof −
assume *hpsafe* $(a{*}{*})$
then have *hpsafe*:*hpsafe* a **by** (*auto dest*: *hpsafe.cases*)
assume *ssafe*:*ssafe* σ
have *agree*:$\bigwedge \nu\ \mu.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ a\ \sigma) \Longrightarrow \mathit{Vagree}\ \nu\ \mu\ (-\mathit{BVP}(\mathit{Psubst}\ a\ \sigma))$
subgoal for ν μ
using *bound-effect*[*OF good-interp*, *of* $(\mathit{Psubst}\ a\ \sigma)$ ν, *OF substSafe*] **by** *auto*
done
have *sem-eq*:$\bigwedge \nu\ \mu\ \omega.\ (\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ a\ \sigma) \Longrightarrow$
$((\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{local.adjoint}\ I\ \sigma\ \nu)\ a) =$
$((\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{local.adjoint}\ I\ \sigma\ \mu)\ a)$
subgoal for ν μ ω
proof −
assume *assm*:$(\nu, \mu) \in \mathit{prog\text{-}sem}\ I\ (\mathit{Psubst}\ a\ \sigma)$
show $((\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{local.adjoint}\ I\ \sigma\ \nu)\ a) = ((\mu, \omega) \in \mathit{prog\text{-}sem}\ (\mathit{local.adjoint}\ I\ \sigma\ \mu)\ a)$
using *uadmit-prog-adjoint*[*OF PUA agree*[*OF assm*] *hpsafe ssafe good-interp*]

by *auto*
qed
done
fix ν ω
have *UN-rule*:$\bigwedge$ *a S S'*. ($\bigwedge$*n b*. (*a*,*b*) $\in$ *S n* $\longleftrightarrow$ (*a*,*b*) $\in$ *S' n*) $\Longrightarrow$ ($\bigwedge$*b*. (*a*,*b*) $\in$ ($\bigcup$ *n*. *S n*) $\longleftrightarrow$ (*a*,*b*) $\in$ ($\bigcup$ *n*. *S' n*))
by *auto*
have *eqL*:((*ν*, *ω*) $\in$ *prog-sem I* (*Psubst* (*a***) *σ*)) = ((*ν*, *ω*) $\in$ ($\bigcup$ *n*. (*prog-sem I* (*Psubst a σ*)) ^^ *n*))
using *rtrancl-is-UN-relpow* **by** *auto*
moreover have *eachEq*:$\bigwedge$*n*. (($\bigwedge$*ν ω*. ((*ν*, *ω*) $\in$ (*prog-sem I* (*Psubst a σ*)) ^^ *n*) = ((*ν*, *ω*) $\in$ (*prog-sem* (*adjoint I σ ν*) *a*) ^^ *n*)))
proof −
fix *n*
show (($\bigwedge$*ν ω*. ((*ν*, *ω*) $\in$ (*prog-sem I* (*Psubst a σ*)) ^^ *n*) = ((*ν*, *ω*) $\in$ (*prog-sem* (*adjoint I σ ν*) *a*) ^^ *n*)))
proof (*induct n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*) **then**
have *IH2*:$\bigwedge$*ν ω*. ((*ν*, *ω*) $\in$ *prog-sem I* (*Psubst a σ*) ^^ *n*) = ((*ν*, *ω*) $\in$ *prog-sem* (*local.adjoint I σ ν*) *a* ^^ *n*)
by *auto*
have *relpow*:$\bigwedge$*R n*. *R* ^^ *Suc n* = *R O R* ^^ *n*
using *relpow.simps*(*2*) *relpow-commute* **by** *metis*
show *?case*
apply (*simp only*: *relpow*[*of n prog-sem I* (*Psubst a σ*)] *relpow*[*of n prog-sem* (*adjoint I σ ν*) *a*])
apply(*unfold relcomp-unfold*)
apply *auto*
subgoal for *ab b*
apply(*rule exI*[**where** *x*=*ab*])
apply(*rule exI*[**where** *x*=*b*])
using *IH2 IH*[*OF hpsafe ssafe*] *sem-eq*[*of ν* (*ab*,*b*) *ω*] **apply** *auto*
using *uadmit-prog-adjoint*[*OF PUA agree hpsafe ssafe good-interp*] *IH*[*OF hpsafe ssafe*]
apply (*metis* (*no-types*, *lifting*))
using *uadmit-prog-adjoint*[*OF PUA agree hpsafe ssafe good-interp*] *IH*[*OF hpsafe ssafe*]
apply (*metis* (*no-types*, *lifting*))
done
subgoal for *ab b*
apply(*rule exI*[**where** *x*=*ab*])
apply(*rule exI*[**where** *x*=*b*])
using *IH2 IH*[*OF hpsafe ssafe*] *sem-eq*[*of ν* (*ab*,*b*) *ω*] **apply** *auto*
using *uadmit-prog-adjoint*[*OF PUA agree hpsafe ssafe good-interp*] *IH*[*OF hpsafe ssafe*]
apply (*metis* (*no-types*, *lifting*))

using *uadmit-prog-adjoint*[*OF PUA agree hpsafe ssafe good-interp*] *IH*[*OF hpsafe ssafe*]
apply (*metis* (*no-types*, *lifting*))
done
done
qed
qed
moreover have $((\nu, \omega) \in (\bigcup n.\ (prog\text{-}sem\ I\ (Psubst\ a\ \sigma))\ ^\frown\frown\ n)) = ((\nu, \omega) \in (\bigcup\ n.(prog\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ a)^\frown\frown\ n))$
apply(*rule UN-rule*)
using *eachEq* **by** *auto*
moreover have *eqR*:$((\nu, \omega) \in prog\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ (a{*}{*})) = ((\nu, \omega) \in (\bigcup n.\ (prog\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ a)\ ^\frown\frown\ n))$
using *rtrancl-is-UN-relpow* **by** *auto*
ultimately show $((\nu, \omega) \in prog\text{-}sem\ I\ (Psubst\ (a{*}{*})\ \sigma)) = ((\nu, \omega) \in prog\text{-}sem\ (local.adjoint\ I\ \sigma\ \nu)\ (a{*}{*}))$
by *auto*
qed
then show *?case* **by** *auto*
next
case (*Padmit-ODE* σ *ODE* φ) **then**
have *OA*:*Oadmit* σ *ODE* (*BVO ODE*)
and *FA*:*Fadmit* σ φ
and *FUA*:*FUadmit* σ φ (*BVO ODE*)
and *IH*:*fsafe* $\varphi \Longrightarrow$ *ssafe* $\sigma \Longrightarrow (\bigwedge\nu.\ (\nu \in fml\text{-}sem\ I\ (Fsubst\ \varphi\ \sigma)) = (\nu \in fml\text{-}sem\ (local.adjoint\ I\ \sigma\ \nu)\ \varphi))$
by *auto*
have *hpsafe* (*EvolveODE ODE* φ) $\Longrightarrow$
ssafe $\sigma \Longrightarrow (\bigwedge\nu\ \omega.\ ((\nu, \omega) \in prog\text{-}sem\ I\ (Psubst\ (EvolveODE\ ODE\ \varphi)\ \sigma)) = ((\nu, \omega) \in prog\text{-}sem\ (local.adjoint\ I\ \sigma\ \nu)\ (EvolveODE\ ODE\ \varphi)))$
proof (*auto*)
fix *aa ba bb*
and *sol* :: *real* $\Rightarrow$(*real*, *'sz*) *vec*
and *t* :: *real*
assume *ssafe*:*ssafe* σ
assume *osafe*:*osafe ODE*
assume *fsafe*:*fsafe* φ
assume *t*:$0 \leq t$
assume *eq*:$(aa,ba) = mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ t)$
assume *sol*:$(sol\ solves\text{-}ode\ (\lambda a.\ ODE\text{-}sem\ I\ (Osubst\ ODE\ \sigma)))\ \{0..t\}$
$\{x.\ mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ x \in fml\text{-}sem\ I\ (Fsubst\ \varphi\ \sigma)\}$
have *silly*:
$\bigwedge t.\ mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ t) = mk\text{-}v\ (local.adjoint\ I\ \sigma\ (sol\ t,\ bb))\ ODE\ (sol\ 0,\ bb)\ (sol\ t)$
subgoal for *t* **using** *subst-mkv*[*OF good-interp OA osafe ssafe*, *of* (*sol 0*, *bb*) (*sol t*, *bb*)] **by** *auto* **done**
have *hmmsubst*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow Vagree\ (sol\ 0,bb)\ (sol\ s,\ bb)\ (-(BVO\ (Osubst\ ODE\ \sigma)))$
subgoal for *s*

apply (*rule ODE-bound-effect*[*of s*])
apply *auto*[*1*]
by (*rule sol*)
done
have *sub*:(−(*BVO ODE*)) ⊆ (−(*BVO* (*Osubst ODE* σ)))
by(*induction ODE*, *auto*)
have *hmm*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ *Vagree* (*sol 0*,*bb*) (*sol s*, *bb*) (−(*BVO ODE*))
subgoal for *s*
using *agree-sub*[*OF sub hmmsubst*[*of s*]] **by** *auto*
done
from *hmm* **have** *hmm'*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ *VSagree* (*sol 0*) (*sol s*) {*x*. *Inl x* ∈ (−(*BVO ODE*))}
unfolding *VSagree-def Vagree-def* **by** *auto*
note *hmmm* = *hmmsubst*
from *hmmm* **have** *hmmm'*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ *VSagree* (*sol 0*) (*sol s*) {*x*. *Inl x* ∈ (−(*BVO* (*Osubst ODE* σ)))}
unfolding *VSagree-def Vagree-def* **by** *auto*
have *Vagree-of-VSagree*:⋀ν*1* ν*2* ω*1* ω*2 S*. *VSagree* ν*1* ν*2* {*x*. *Inl x* ∈ *S*} ⟹ *VSagree* ω*1* ω*2* {*x*. *Inr x* ∈ *S*} ⟹ *Vagree* (ν*1*, ω*1*) (ν*2*, ω*2*) *S*
unfolding *VSagree-def Vagree-def* **by** *auto*
have *mkv*:⋀*s*. *s* ∈ {*0*..*t*} ⟹ *mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) = *mk-v* (*adjoint I* σ (*sol s*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol s*)
subgoal for *s* **by** (*rule silly*[*of s*])
done
have *lem*:⋀*ODE*. *Oadmit* σ *ODE* (*BVO ODE*) ⟹ (⋃ *i*∈{*i* |*i*. *Inl i* ∈ *SIGO ODE*}. *case SFunctions* σ *i of None* ⇒ {} | *Some x* ⇒ *FVT x*) ⊆ (−(*BVO ODE*))
subgoal for *ODE*
apply(*induction rule*: *Oadmit.induct*)
apply *auto*
subgoal for σ ϑ *U x xa*
apply (*cases SFunctions* σ *xa*)
apply *auto*
unfolding *TUadmit-def*
proof −
fix *a*
assume *un*:(⋃ *i*∈*SIGT* ϑ. *case SFunctions* σ *i of None* ⇒ {} | *Some x* ⇒ *FVT x*) ∩ *U* = {}
assume *sig*:*xa* ∈ *SIGT* ϑ
assume *some*:*SFunctions* σ *xa* = *Some a*
assume *fvt*:*x* ∈ *FVT a*
assume *xU*:*x* ∈ *U*
from *un sig* **have** (*case SFunctions* σ *xa of None* ⇒ {} | *Some x* ⇒ *FVT x*) ∩ *U* = {}
by *auto*
then have (*FVT a*) ∩ *U* = {}
using *some* **by** *auto*
then show *False* **using** *fvt xU* **by** *auto*
qed
done

done
have *FVT-sub*:$(\bigcup i{\in}\{i \mid i.\ Inl\ i \in SIGO\ ODE\}.\ case\ SFunctions\ \sigma\ i\ of\ None \Rightarrow \{\} \mid Some\ x \Rightarrow FVT\ x) \subseteq (-(BVO\ ODE))$
using *lem*[*OF OA*] **by** *auto*
have *agrees*: $\bigwedge s.\ s \in \{0..t\} \Longrightarrow Vagree\ (sol\ 0,bb)\ (sol\ s,\ bb)\ (\bigcup i{\in}\{i \mid i.\ Inl\ i \in SIGO\ ODE\}.\ case\ SFunctions\ \sigma\ i\ of\ None \Rightarrow \{\} \mid Some\ x \Rightarrow FVT\ x)$
subgoal for *s* **using** *agree-sub*[*OF FVT-sub hmm*[*of s*]] **by** *auto* **done**
have $\bigwedge s.\ s \in \{0..t\} \Longrightarrow mk\text{-}v\ (adjoint\ I\ \sigma\ (sol\ 0,\ bb))\ ODE = mk\text{-}v\ (adjoint\ I\ \sigma\ (sol\ s,\ bb))\ ODE$
subgoal for *s*
apply (*rule uadmit-mkv-adjoint*)
prefer *3*
subgoal using *agrees* **by** *auto*
using *OA hmm*[*of s*] **unfolding** *Vagree-def*
using *ssafe good-interp osafe* **by** *auto*
done
then have *mkva*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow mk\text{-}v\ (adjoint\ I\ \sigma\ (sol\ s,\ bb))\ ODE\ (sol\ 0,\ bb)\ (sol\ s) = mk\text{-}v\ (adjoint\ I\ \sigma\ (sol\ 0,\ bb))\ ODE\ (sol\ 0,\ bb)\ (sol\ s)$
by *presburger*
have *main-eq*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow\ mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s) = mk\text{-}v\ (adjoint\ I\ \sigma\ (sol\ 0,\ bb))\ ODE\ (sol\ 0,\ bb)\ (sol\ s)$
using *mkv mkva* **by** *auto*
note *mkvt* = *main-eq*[*of t*]
have *fml-eq1*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
$(mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s) \in fml\text{-}sem\ I\ (Fsubst\ \varphi\ \sigma))$
$= (mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s) \in fml\text{-}sem\ (adjoint\ I\ \sigma\ (mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s)))\ \varphi)$
using *IH*[*OF fsafe ssafe*] **by** *auto*
have *fml-vagree*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow Vagree\ (mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s))\ (sol\ 0,\ bb)\ (-\ semBV\ I\ (Osubst\ ODE\ \sigma))$
subgoal for *s*
using *mk-v-agree*[*of I Osubst ODE* σ (*sol 0*,*bb*) *sol s*] *osubst-eq-ODE-vars*[*of I ODE* σ]
unfolding *Vagree-def*
by *auto*
done
have *sembv-eq*:$semBV\ I\ (Osubst\ ODE\ \sigma) = semBV\ (adjoint\ I\ \sigma\ (sol\ 0,\ bb))\ ODE$
using *subst-semBV* **by** *auto*
have *fml-vagree'*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow Vagree\ (mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s))\ (sol\ 0,\ bb)\ (-\ semBV\ (adjoint\ I\ \sigma\ (sol\ 0,bb))\ ODE)$
using *sembv-eq fml-vagree* **by** *auto*
have *mysub*:$-BVO\ ODE \subseteq -(semBV\ I\ (Osubst\ ODE\ \sigma))$
by(*induction ODE*,*auto*)
have *fml-vagree*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow Vagree\ (mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s))\ (sol\ 0,\ bb)\ (-\ BVO\ ODE)$
subgoal for *s* **using** *agree-sub*[*OF mysub fml-vagree*[*of s*]] **by** *auto* **done**
have *fml-sem-eq*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow fml\text{-}sem\ (adjoint\ I\ \sigma\ (mk\text{-}v\ I\ (Osubst\ ODE\ \sigma)\ (sol\ 0,\ bb)\ (sol\ s)))\ \varphi = fml\text{-}sem\ (adjoint\ I\ \sigma\ (sol\ 0,\ bb))\ \varphi$

apply (*rule uadmit-fml-adjoint*)
using *FUA fsafe ssafe good-interp fml-vagree* **by** *auto*
have *fml-eq2*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
((*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*))) φ)
=(*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ))
using *fml-sem-eq* **by** *auto*
have *fml-eq3*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
(*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ) = (*mk-v* (*adjoint I* σ (*sol 0*,*bb*)) *ODE* (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ)
using *main-eq* **by** *auto*
have *fml-eq*: $\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
(*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem I* (*Fsubst* φ σ))
= (*mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ)
using *fml-eq1 fml-eq2 fml-eq3* **by** *meson*
have *sem-eq*:$\bigwedge t.$ *ODE-sem I* (*Osubst ODE* σ) (*sol t*) = *ODE-sem* (*adjoint I* σ (*sol t*, *bb*)) *ODE* (*sol t*)
subgoal for *t*
using *subst-ode*[*OF good-interp osafe ssafe OA*, *of* (*sol t*,*bb*)] **by** *auto*
done
have *sem-fact*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *ODE-sem I* (*Osubst ODE* σ) (*sol s*) = *ODE-sem* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol s*)
subgoal for *s*
using *subst-ode*[*OF good-interp osafe ssafe OA*, *of* (*sol s*, *bb*)]
uadmit-ode-adjoint'[*OF ssafe good-interp agrees*[*of s*] *osafe*]
by *auto*
done
have *sol'*:(*sol solves-ode* (λ-. *ODE-sem* (*adjoint I* σ (*sol 0*, *bb*)) *ODE*)) {*0..t*} {*x*. *mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) *x* $\in$ *fml-sem I* (*Fsubst* φ σ)}
apply (*rule solves-ode-congI*)
apply (*rule sol*)
subgoal for *ta* **by** *auto*
subgoal for *ta* **by** (*rule sem-fact*[*of ta*])
subgoal by (*rule refl*)
subgoal by (*rule refl*)
done
have *sub*:$\bigwedge s.\ s \in \{0..t\}$
$\Longrightarrow$ *sol s* $\in$ {*x*. (*mk-v* (*adjoint I* σ (*sol 0*,*bb*)) *ODE* (*sol 0*, *bb*) *x* $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ)}
using *fml-eq rangeI t sol solves-ode-domainD* **by** *fastforce*
have *sol''*:(*sol solves-ode* (λ*c*. *ODE-sem* (*adjoint I* σ (*sol 0*, *bb*)) *ODE*)) {*0..t*} {*x*. *mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) *x* $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ}
apply (*rule solves-odeI*)
subgoal using *sol' solves-ode-vderivD* **by** *blast*
using *sub* **by** *auto*

show $\exists$ *sola. sol 0 = sola 0* $\wedge$
(∃ *ta. mk-v I* (*Osubst ODE σ*) (*sol 0*, *bb*) (*sol t*) = *mk-v* (*local.adjoint I σ* (*sol 0*, *bb*)) *ODE* (*sola 0*, *bb*) (*sola ta*) ∧
0 ≤ *ta* ∧
(*sola solves-ode* (λ*a. ODE-sem* (*local.adjoint I σ* (*sol 0*, *bb*)) *ODE*)) {*0..ta*}
{*x. mk-v* (*local.adjoint I σ* (*sol 0*, *bb*)) *ODE* (*sola 0*, *bb*) *x* ∈ *fml-sem* (*local.adjoint I σ* (*sol 0*, *bb*)) *φ*})
apply(*rule exI*[**where** *x*=*sol*])
apply(*rule conjI*)
subgoal by (*rule refl*)
apply(*rule exI*[**where** *x*=*t*])
apply(*rule conjI*)
subgoal using *mkvt t* **by** *auto*
apply(*rule conjI*)
subgoal by (*rule t*)
subgoal by (*rule sol''*)
done
next
fix *aa ba bb*
and *sol*::*real* ⇒ (*real*, *'sz*) *vec*
and *t*::*real*
assume *ssafe*:*ssafe σ*
assume *osafe*:*osafe ODE*
assume *fsafe*:*fsafe φ*

assume *eq*:(*aa*,*ba*) = *mk-v* (*adjoint I σ* (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol t*)
assume *t*:*0* ≤ *t*
assume *sol*:(*sol solves-ode* (λ*a. ODE-sem* (*adjoint I σ* (*sol 0*, *bb*)) *ODE*)) {*0..t*}
{*x. mk-v* (*adjoint I σ* (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) *x* ∈ *fml-sem* (*adjoint I σ* (*sol 0*, *bb*)) *φ*}
have *silly*:
⋀*t. mk-v I* (*Osubst ODE σ*) (*sol 0*, *bb*) (*sol t*) = *mk-v* (*local.adjoint I σ* (*sol t*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol t*)
subgoal for *t* **using** *subst-mkv*[*OF good-interp OA osafe ssafe*, *of* (*sol 0*, *bb*) (*sol t*, *bb*)] **by** *auto* **done**
have *hmm*:⋀*s. s* ∈ {*0..t*} ⟹ *Vagree* (*sol 0*,*bb*) (*sol s*, *bb*) (−(*BVO ODE*))
subgoal for *s*
apply (*rule ODE-bound-effect*[*of s*])
apply *auto*[*1*]
by (*rule sol*)
done
from *hmm* **have** *hmm'*:⋀*s. s* ∈ {*0..t*} ⟹ *VSagree* (*sol 0*) (*sol s*) {*x. Inl x* ∈ (−(*BVO ODE*))}
unfolding *VSagree-def Vagree-def* **by** *auto*
have *Vagree-of-VSagree*:⋀*ν1 ν2 ω1 ω2 S. VSagree ν1 ν2* {*x. Inl x* ∈ *S*} ⟹ *VSagree ω1 ω2* {*x. Inr x* ∈ *S*} ⟹ *Vagree* (*ν1*, *ω1*) (*ν2*, *ω2*) *S*
unfolding *VSagree-def Vagree-def* **by** *auto*

have *mkv*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) = *mk-v* (*adjoint I* σ (*sol s*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol s*)
subgoal for *s* **by** (*rule silly*[*of s*])
done
have *lem*:$\bigwedge$*ODE. Oadmit* σ *ODE* (*BVO ODE*) $\Longrightarrow$ ($\bigcup i \in \{i \mid i.\ Inl\ i \in SIGO\ ODE\}$. *case SFunctions* σ *i of None* $\Rightarrow$ {} | *Some x* $\Rightarrow$ *FVT x*) $\subseteq$ ($-$(*BVO ODE*))
subgoal for *ODE*
apply(*induction rule*: *Oadmit.induct*)
apply *auto*
subgoal for σ ϑ *U x xa*
apply (*cases SFunctions* σ *xa*)
apply *auto*
unfolding *TUadmit-def*
proof $-$
fix *a*
assume *un*:($\bigcup i \in SIGT\ \vartheta$. *case SFunctions* σ *i of None* $\Rightarrow$ {} | *Some x* $\Rightarrow$ *FVT x*) $\cap$ *U* = {}
assume *sig*:*xa* $\in$ *SIGT* ϑ
assume *some*:*SFunctions* σ *xa* = *Some a*
assume *fvt*:*x* $\in$ *FVT a*
assume *xU*:*x* $\in$ *U*
from *un sig* **have** (*case SFunctions* σ *xa of None* $\Rightarrow$ {} | *Some x* $\Rightarrow$ *FVT x*) $\cap$ *U* = {}
by *auto*
then have (*FVT a*) $\cap$ *U* = {}
using *some* **by** *auto*
then show *False* **using** *fvt xU* **by** *auto*
qed
done
done
have *FVT-sub*:($\bigcup i \in \{i \mid i.\ Inl\ i \in SIGO\ ODE\}$. *case SFunctions* σ *i of None* $\Rightarrow$ {} | *Some x* $\Rightarrow$ *FVT x*) $\subseteq$ ($-$(*BVO ODE*))
using *lem*[*OF OA*] **by** *auto*
have *agrees*: $\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *Vagree* (*sol 0*,*bb*) (*sol s*, *bb*) ($\bigcup i \in \{i \mid i.\ Inl\ i \in SIGO\ ODE\}$. *case SFunctions* σ *i of None* $\Rightarrow$ {} | *Some x* $\Rightarrow$ *FVT x*)
subgoal for *s* **using** *agree-sub*[*OF FVT-sub hmm*[*of s*]] **by** *auto* **done**
have $\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* = *mk-v* (*adjoint I* σ (*sol s*, *bb*)) *ODE*
subgoal for *s*
apply (*rule uadmit-mkv-adjoint*)
prefer *3*
subgoal using *agrees* **by** *auto*
using *OA hmm*[*of s*] **unfolding** *Vagree-def*
using *ssafe good-interp osafe* **by** *auto*
done
then have *mkva*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *mk-v* (*adjoint I* σ (*sol s*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol s*) = *mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol s*)
by *presburger*
have *main-eq*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*)

$=$ *mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol s*)
using *mkv mkva* **by** *auto*
note *mkvt* = *main-eq*[*of t*]
have *fml-eq1*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
(*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem I* (*Fsubst* φ σ))
$=$ (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*))) φ)
using *IH*[*OF fsafe ssafe*] **by** *auto*
have *fml-vagree*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *Vagree* (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*)) (*sol 0*, *bb*) ($-$ *semBV I* (*Osubst ODE* σ))
subgoal for *s*
using *mk-v-agree*[*of I Osubst ODE* σ (*sol 0*,*bb*) *sol s*] *osubst-eq-ODE-vars*[*of I ODE* σ]
unfolding *Vagree-def*
by *auto*
done
have *sembv-eq*:*semBV I* (*Osubst ODE* σ) $=$ *semBV* (*adjoint I* σ (*sol 0*, *bb*)) *ODE*
using *subst-semBV* **by** *auto*
have *fml-vagree'*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *Vagree* (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*)) (*sol 0*, *bb*) ($-$ *semBV* (*adjoint I* σ (*sol 0*,*bb*)) *ODE*)
using *sembv-eq fml-vagree* **by** *auto*
have *mysub*:$-$*BVO ODE* $\subseteq -$(*semBV I* (*Osubst ODE* σ))
by(*induction ODE*,*auto*)
have *fml-vagree*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *Vagree* (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*)) (*sol 0*, *bb*) ($-$ *BVO ODE*)
subgoal for *s* **using** *agree-sub*[*OF mysub fml-vagree*[*of s*]] **by** *auto* **done**
have *fml-sem-eq*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *fml-sem* (*adjoint I* σ (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*))) φ $=$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ
apply (*rule uadmit-fml-adjoint*)
using *FUA fsafe ssafe good-interp fml-vagree* **by** *auto*
have *fml-eq2*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
((*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*))) φ)
$=$(*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ))
using *fml-sem-eq* **by** *auto*
have *fml-eq3*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
(*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ) $=$ (*mk-v* (*adjoint I* σ (*sol 0*,*bb*)) *ODE* (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ)
using *main-eq* **by** *auto*
have *fml-eq*: $\bigwedge s.\ s \in \{0..t\} \Longrightarrow$
(*mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem I* (*Fsubst* φ σ))
$=$ (*mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ)
using *fml-eq1 fml-eq2 fml-eq3* **by** *meson*
have *sem-eq*:$\bigwedge t.$ *ODE-sem I* (*Osubst ODE* σ) (*sol t*) $=$ *ODE-sem* (*adjoint I* σ (*sol t*, *bb*)) *ODE* (*sol t*)

subgoal for t
using *subst-ode*[*OF good-interp osafe ssafe OA*, *of* (*sol t*,*bb*)] **by** *auto*
done
have *sem-fact*:$\bigwedge s.\ s \in \{0..t\} \Longrightarrow$ *ODE-sem I* (*Osubst ODE* σ) (*sol s*) = *ODE-sem* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol s*)
subgoal for s
using *subst-ode*[*OF good-interp osafe ssafe OA*, *of* (*sol s*, *bb*)]
uadmit-ode-adjoint'[*OF ssafe good-interp agrees*[*of s*] *osafe*]
by *auto*
done
have *sub*:$\bigwedge s.\ s \in \{0..t\}$
$\Longrightarrow$ *sol s* $\in$ {*x. mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) (*sol s*) $\in$ *fml-sem I* (*Fsubst* φ σ)}
using *fml-eq rangeI t sol solves-ode-domainD* **by** *fastforce*
have *sol'*:(*sol solves-ode* ($\lambda a.$ *ODE-sem I* (*Osubst ODE* σ))) {*0..t*} {*x. mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) *x* $\in$ *fml-sem* (*adjoint I* σ (*sol 0*, *bb*)) φ}
apply (*rule solves-ode-congI*)
apply (*rule sol*)
subgoal for *ta* **by** *auto*
subgoal for *ta* **using** *sem-fact*[*of ta*] **by** *auto*
subgoal by (*rule refl*)
subgoal by (*rule refl*)
done
have *sol''*:(*sol solves-ode* ($\lambda a.$ *ODE-sem I* (*Osubst ODE* σ))) {*0..t*} {*x. mk-v I* (*Osubst ODE* σ) (*sol 0*, *bb*) *x* $\in$ *fml-sem I* (*Fsubst* φ σ)}
apply (*rule solves-odeI*)
subgoal using *sol' solves-ode-vderivD* **by** *blast*
subgoal for *ta* **using** *sub*[*of ta*] **apply** *auto*
by (*meson empty-iff*)
done
show $\exists$ *sola. sol 0 = sola 0* $\wedge$
($\exists$ *ta. mk-v* (*adjoint I* σ (*sol 0*, *bb*)) *ODE* (*sol 0*, *bb*) (*sol t*) = *mk-v I* (*Osubst ODE* σ) (*sola 0*, *bb*) (*sola ta*) $\wedge$
0 $\leq$ *ta* $\wedge$
(*sola solves-ode* ($\lambda a.$ *ODE-sem I* (*Osubst ODE* σ))) {*0..ta*} {*x. mk-v I* (*Osubst ODE* σ) (*sola 0*, *bb*) *x* $\in$ *fml-sem I* (*Fsubst* φ σ)})
apply(*rule exI*[**where** *x*=*sol*])
apply(*rule conjI*)
subgoal by (*rule refl*)
apply(*rule exI*[**where** *x*=*t*])
apply(*rule conjI*)
subgoal using *mkvt t* **by** *auto*
apply(*rule conjI*)
subgoal by (*rule t*)
subgoal using *sol''* **by** *auto*
done
qed
then show *?case* **by** *auto*
next

case (*Padmit-Choice* σ a b) **then**
have *IH1*:*hpsafe* $a \Longrightarrow$ *ssafe* $\sigma \Longrightarrow (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in$ *prog-sem* I (*Psubst* $a\ \sigma$)) = $((\nu, \omega) \in$ *prog-sem* (*local.adjoint* $I\ \sigma\ \nu$) a))
and *IH2*:*hpsafe* $b \Longrightarrow$ *ssafe* $\sigma \Longrightarrow (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in$ *prog-sem* I (*Psubst* $b\ \sigma$)) = $((\nu, \omega) \in$ *prog-sem* (*local.adjoint* $I\ \sigma\ \nu$) b))
by *blast*+
have *hpsafe1*:*hpsafe* $(a \cup\cup b) \Longrightarrow$ *hpsafe* a
and *hpsafe2*:*hpsafe* $(a \cup\cup b) \Longrightarrow$ *hpsafe* b
by (*auto dest*: *hpsafe.cases*)
show *?case* **using** *IH1*[*OF hpsafe1*] *IH2*[*OF hpsafe2*] **by** *auto*
next
case (*Padmit-Assign* $\sigma\ \vartheta\ x$) **then**
have *TA*:*Tadmit* $\sigma\ \vartheta$ **by** *auto*
have *hpsafe* (*Assign* $x\ \vartheta$) $\Longrightarrow$ *ssafe* $\sigma \Longrightarrow\ (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in$ *prog-sem* I (*Psubst* (*Assign* $x\ \vartheta$) σ)) = $((\nu, \omega) \in$ *prog-sem* (*adjoint* $I\ \sigma\ \nu$) (*Assign* $x\ \vartheta$)))
proof −
assume *hpsafe*:*hpsafe* (*Assign* $x\ \vartheta$)
assume *ssafe*:*ssafe* σ
from *ssafe* **have** *ssafes*:$(\bigwedge i\ f'.$ *SFunctions* $\sigma\ i =$ *Some* $f' \Longrightarrow$ *dfree* $f')$
$(\bigwedge f\ f'.$ *SPredicates* $\sigma\ f =$ *Some* $f' \Longrightarrow$ *fsafe* $f')$
unfolding *ssafe-def* **by** *auto*
from *hpsafe* **have** *dsafe*:*dsafe* ϑ **by** (*auto elim*: *hpsafe.cases*)
fix $\nu\ \omega$
show *?thesis* $\nu\ \omega$
using *subst-dterm*[*OF good-interp TA dsafe ssafes*]
by *auto*
qed
then show *?case* **by** *auto*
next
case (*Padmit-DiffAssign* $\sigma\ \vartheta\ x$) **then**
have *TA*:*Tadmit* $\sigma\ \vartheta$ **by** *auto*
have *hpsafe* (*DiffAssign* $x\ \vartheta$) $\Longrightarrow$ *ssafe* $\sigma \Longrightarrow\ (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in$ *prog-sem* I (*Psubst* (*DiffAssign* $x\ \vartheta$) σ)) = $((\nu, \omega) \in$ *prog-sem* (*adjoint* $I\ \sigma\ \nu$) (*DiffAssign* x ϑ)))
proof −
assume *hpsafe*:*hpsafe* (*DiffAssign* $x\ \vartheta$)
assume *ssafe*:*ssafe* σ
from *ssafe* **have** *ssafes*:$(\bigwedge i\ f'.$ *SFunctions* $\sigma\ i =$ *Some* $f' \Longrightarrow$ *dfree* $f')$
$(\bigwedge f\ f'.$ *SPredicates* $\sigma\ f =$ *Some* $f' \Longrightarrow$ *fsafe* $f')$
unfolding *ssafe-def* **by** *auto*
from *hpsafe* **have** *dsafe*:*dsafe* ϑ **by** (*auto elim*: *hpsafe.cases*)
fix $\nu\ \omega$
show *?thesis* $\nu\ \omega$
using *subst-dterm*[*OF good-interp TA dsafe ssafes*]
by *auto*
qed
then show *?case* **by** *auto*
next
case (*Padmit-Test* $\sigma\ \varphi$) **then**

```
  have IH:fsafe φ ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst φ σ)) = (ν ∈
fml-sem (local.adjoint I σ ν) φ))
    by auto
  have hpsafe (? φ) ⟹ ssafe σ ⟹ (⋀ν ω. ((ν, ω) ∈ prog-sem I (Psubst (? φ)
σ)) = ((ν, ω) ∈ prog-sem (local.adjoint I σ ν) (? φ)))
  proof −
    assume hpsafe:hpsafe (? φ)
    from hpsafe have fsafe:fsafe φ by (auto dest: hpsafe.cases)
    assume ssafe:ssafe σ
    fix ν ω
    show ?thesis ν ω using IH[OF fsafe ssafe] by auto
  qed
  then show ?case by auto
next
  case (Fadmit-Geq σ ϑ1 ϑ2) then
  have TA1:Tadmit σ ϑ1 and TA2:Tadmit σ ϑ2
    by auto
  have fsafe (Geq ϑ1 ϑ2) ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst (Geq ϑ1
ϑ2) σ)) = (ν ∈ fml-sem (local.adjoint I σ ν) (Geq ϑ1 ϑ2)))
  proof −
    assume fsafe:fsafe (Geq ϑ1 ϑ2)
    assume ssafe:ssafe σ
    from fsafe have dsafe1:dsafe ϑ1 and dsafe2:dsafe ϑ2
      by (auto dest: fsafe.cases)
    from ssafe have ssafes:(⋀i f'. SFunctions σ i = Some f' ⟹ dfree f')
      (⋀f f'. SPredicates σ f = Some f' ⟹ fsafe f')
      unfolding ssafe-def by auto
    fix ν
    show ?thesis ν using
      subst-dterm[OF good-interp TA1 dsafe1 ssafes]
      subst-dterm[OF good-interp TA2 dsafe2 ssafes]
      by auto
  qed
  then show ?case by auto
next
  case (Fadmit-Prop1 σ args p p') then
    have TA:⋀i. Tadmit σ (args i)
    and some:SPredicates σ p = Some p'
    and NFA:NFadmit (λi. Tsubst (args i) σ) p'
    and substSafes:⋀i. dsafe (Tsubst (args i) σ)
      by auto
    have fsafe ($φ p args) ⟹
          ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst ($φ p args) σ)) = (ν ∈ fml-sem
(local.adjoint I σ ν) ($φ p args)))
    proof −
      assume fsafe:fsafe ($φ p args)
      assume ssafe:ssafe σ
      from ssafe have ssafes:(⋀i f'. SFunctions σ i = Some f' ⟹ dfree f')
      (⋀f f'. SPredicates σ f = Some f' ⟹ fsafe f')
```

```
    unfolding ssafe-def by auto
    fix ν
    from fsafe have safes:⋀i. dsafe (args i) using dfree-is-dsafe by auto
    have IH:(⋀ν'. ⋀i. dsafe (args i) ⟹
      dterm-sem I (Tsubst (args i) σ) ν = dterm-sem (adjoint I σ ν) (args i) ν)
      using subst-dterm[OF good-interp TA safes ssafes] by auto
    have eqs:⋀i ν'. dterm-sem I (Tsubst (args i) σ) ν = dterm-sem (adjoint I σ
ν) (args i) ν
      by (auto simp add: IH safes)
    let ?sub = (λ i. Tsubst (args i) σ)
    have freef:fsafe p' using ssafe some unfolding ssafe-def by auto
    have IH2:(ν ∈ fml-sem I (FsubstFO p' ?sub)) = (ν ∈ fml-sem (adjointFO I
?sub ν) p')
      using nsubst-fml good-interp NFA freef substSafes
      by blast
      have vec:(χ i. dterm-sem I (Tsubst (args i) σ) ν) = (χ i. dterm-sem
(local.adjoint I σ ν) (args i) ν)
      apply(auto simp add: vec-eq-iff)
      subgoal for i
        using IH[of i, OF safes[of i]]
        by auto
      done
    show ?thesis ν
      using IH safes eqs apply (auto simp add: IH2 some good-interp)
      using some unfolding adjoint-def adjointFO-def by auto
  qed
 then show ?case by auto
next
 case (Fadmit-Prop2 σ args p)
 note TA = Fadmit-Prop2.hyps(1)
  and none = Fadmit-Prop2.hyps(2)
 have fsafe (Prop p args) ⟹ ssafe σ ⟹ (⋀ν.(ν ∈ fml-sem I (Fsubst ($φ p args)
σ)) = (ν ∈ fml-sem (local.adjoint I σ ν) ($φ p args)))
 proof −
  assume safe:fsafe (Prop p args) and ssafe:ssafe σ
  from ssafe have ssafes:(⋀i f'. SFunctions σ i = Some f' ⟹ dfree f')
    (⋀f f'. SPredicates σ f = Some f' ⟹ fsafe f')
    unfolding ssafe-def by auto
  fix ν
  from safe have safes:⋀i. dsafe (args i) using dfree-is-dsafe by auto
  have IH:(⋀ν'. ⋀i. dsafe (args i) ⟹
    dterm-sem I (Tsubst (args i) σ) ν = dterm-sem (adjoint I σ ν) (args i) ν)
  using subst-dterm[OF good-interp TA safes ssafes] by auto
  have Ieq:Predicates I p = Predicates (adjoint I σ ν) p
    using none unfolding adjoint-def by auto
  have vec:(χ i. dterm-sem I (Tsubst (args i) σ) ν) = (χ i. dterm-sem (adjoint
I σ ν) (args i) ν)
    apply(auto simp add: vec-eq-iff)
    subgoal for i using IH[of i, OF safes[of i]] by auto
```

```
    done
   show ?thesis ν using none IH Ieq vec by auto
  qed
  then show ?case by auto
next
  case (Fadmit-Not σ φ) then
  have IH:fsafe φ ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst φ σ)) = (ν ∈
fml-sem (local.adjoint I σ ν) φ))
    by blast
  have fsafe:fsafe (Not φ) ⟹ fsafe φ
    by (auto dest: fsafe.cases)
  show ?case using IH[OF fsafe] by auto
next
  case (Fadmit-And σ φ ψ) then
    have IH1:fsafe φ ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst φ σ)) = (ν ∈
fml-sem (local.adjoint I σ ν) φ))
     and IH2:fsafe ψ ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst ψ σ)) = (ν ∈
fml-sem (local.adjoint I σ ν) ψ))
      by (blast)+
    have fsafe1:fsafe (φ && ψ) ⟹ fsafe φ and fsafe2:fsafe (φ && ψ) ⟹ fsafe
ψ
      by (auto dest: fsafe.cases)
    show ?case using IH1[OF fsafe1] IH2[OF fsafe2] by auto
next
  case (Fadmit-Exists σ φ x)
  then have IH:fsafe φ ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst φ σ)) = (ν
∈ fml-sem (local.adjoint I σ ν) φ))
    and FUA:FUadmit σ φ {Inl x}
    by blast+
  have fsafe:fsafe (Exists x φ) ⟹ fsafe φ
    by (auto dest: fsafe.cases)
  have eq:fsafe (Exists x φ) ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst  (Exists
x φ) σ)) = (ν ∈ fml-sem (local.adjoint I σ ν)  (Exists x φ)))
  proof −
    assume fsafe:fsafe (Exists x φ)
    from fsafe have fsafe′:fsafe φ by (auto dest: fsafe.cases)
    assume ssafe:ssafe σ
    fix ν
    have agree:⋀r. Vagree ν (repv ν x r) (− {Inl x})
      unfolding Vagree-def by auto
    have sem-eq:⋀r. ((repv ν x r) ∈ fml-sem (local.adjoint I σ (repv ν x r)) φ) =
                    ((repv ν x r) ∈ fml-sem (local.adjoint I σ ν) φ)
      using uadmit-fml-adjoint[OF FUA agree fsafe′ ssafe good-interp] by auto
    have (ν ∈ fml-sem I (Fsubst  (Exists x φ) σ)) = (∃ r. (repv ν x r) ∈ fml-sem
I (Fsubst φ σ))
      by auto
    moreover have ... = (∃ r. (repv ν x r) ∈ fml-sem (local.adjoint I σ (repv ν x
r)) φ)
      using IH[OF fsafe′ ssafe] by auto
```

moreover have ... = ($\exists\, r.\ (repv\ \nu\ x\ r) \in fml\text{-}sem\ (local.adjoint\ I\ \sigma\ \nu)\ \varphi$)
using *sem-eq* **by** *auto*
moreover have ... = ($\nu \in fml\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ (Exists\ x\ \varphi)$)
by *auto*
ultimately show ($\nu \in fml\text{-}sem\ I\ (Fsubst\ \ (Exists\ x\ \varphi)\ \sigma)$) = ($\nu \in fml\text{-}sem\ (local.adjoint\ I\ \sigma\ \nu)\ \ (Exists\ x\ \varphi)$)
by *auto*
qed
then show *?case* **by** *auto*
next
case (*Fadmit-Diamond* $\sigma\ \varphi\ a$) **then**
have *PA*:*Padmit* $\sigma\ a$
and *FUA*:*FUadmit* $\sigma\ \varphi\ (BVP\ (Psubst\ a\ \sigma))$
and *IH1*:*fsafe* $\varphi \Longrightarrow ssafe\ \sigma \Longrightarrow (\bigwedge\nu.\ (\nu \in fml\text{-}sem\ I\ (Fsubst\ \varphi\ \sigma)) = (\nu \in fml\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ \varphi))$
and *IH2*:*hpsafe* $a \Longrightarrow ssafe\ \sigma \Longrightarrow (\bigwedge\nu\ \omega.\ ((\nu, \omega) \in prog\text{-}sem\ I\ (Psubst\ a\ \sigma)) = ((\nu, \omega) \in prog\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ a))$
and *substSafe*:*hpsafe* $(Psubst\ a\ \sigma)$
by *auto*
have *fsafe* $(\langle\ a\ \rangle\ \varphi) \Longrightarrow ssafe\ \sigma \Longrightarrow (\bigwedge\nu.\ (\nu \in fml\text{-}sem\ I\ (Fsubst\ (\langle\ a\ \rangle\ \varphi)\ \sigma)) = (\nu \in fml\text{-}sem\ (local.adjoint\ I\ \sigma\ \nu)\ (\langle\ a\ \rangle\ \varphi)))$
proof −
assume *fsafe*:*fsafe* $(\langle\ a\ \rangle\ \varphi)$
assume *ssafe*:*ssafe* σ
from *fsafe* **have** *fsafe'*:*fsafe* φ **and** *hpsafe*:*hpsafe* a **by** (*auto dest*: *fsafe.cases*)
fix ν
have *agree*:$\bigwedge\omega.\ (\nu, \omega) \in prog\text{-}sem\ I\ (Psubst\ a\ \sigma) \Longrightarrow Vagree\ \nu\ \omega\ (-BVP(Psubst\ a\ \sigma))$
using *bound-effect*[*OF good-interp*, *of* $(Psubst\ a\ \sigma)\ \nu$, *OF substSafe*] **by** *auto*
have *sem-eq*:$\bigwedge\omega.\ (\nu, \omega) \in prog\text{-}sem\ I\ (Psubst\ a\ \sigma) \Longrightarrow$
$(\omega \in fml\text{-}sem\ (local.adjoint\ I\ \sigma\ \nu)\ \varphi) =$
$(\omega \in fml\text{-}sem\ (local.adjoint\ I\ \sigma\ \omega)\ \varphi)$
using *uadmit-fml-adjoint*[*OF FUA agree fsafe' ssafe good-interp*] **by** *auto*
have ($\nu \in fml\text{-}sem\ I\ (Fsubst\ (\langle\ a\ \rangle\ \varphi)\ \sigma)) = (\exists\ \omega.\ (\nu, \omega) \in prog\text{-}sem\ I\ (Psubst\ a\ \sigma) \wedge \omega \in fml\text{-}sem\ I\ (Fsubst\ \varphi\ \sigma))$
by *auto*
moreover have ... = ($\exists\ \omega.\ (\nu, \omega) \in prog\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ a \wedge \omega \in fml\text{-}sem\ (adjoint\ I\ \sigma\ \omega)\ \varphi$)
using *IH1*[*OF fsafe' ssafe*] *IH2*[*OF hpsafe ssafe*, *of* ν] **by** *auto*
moreover have ... = ($\exists\ \omega.\ (\nu, \omega) \in prog\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ a \wedge \omega \in fml\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ \varphi$)
using *sem-eq IH2 hpsafe ssafe* **by** *blast*
moreover have ... = ($\nu \in fml\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ (\langle\ a\ \rangle\ \varphi)$)
by *auto*
ultimately show *?thesis* ν **by** *auto*
qed
then show *?case* **by** *auto*
next
case (*Fadmit-Prop1* σ *args* $p\ p'$)

have *fsafe*(*Prop p args*) $\Longrightarrow$ *ssafe* $\sigma \Longrightarrow$ ($\bigwedge\nu$.($\nu \in$ *fml-sem I* (*Fsubst* (\$$\varphi$ *p args*) σ)) = ($\nu \in$ *fml-sem* (*local.adjoint I* σ ν) (\$$\varphi$ *p args*)))
proof −
assume *fsafe*:*fsafe* (*Prop p args*)
and *ssafe*:*ssafe* σ
from *ssafe* **have** *ssafes*:($\bigwedge i\ f'$. *SFunctions* σ *i* = *Some* $f' \Longrightarrow$ *dfree* f')
($\bigwedge f\ f'$. *SPredicates* σ *f* = *Some* $f' \Longrightarrow$ *fsafe* f')
unfolding *ssafe-def* **by** *auto*
fix ν
note *TA* = *Fadmit-Prop1.hyps*(*1*)
and *some* = *Fadmit-Prop1.hyps*(*2*) **and** *NFA* = *Fadmit-Prop1.hyps*(*3*)
from *fsafe* **have** *safes*:$\bigwedge i$. *dsafe* (*args i*) **using** *dfree-is-dsafe* **by** *auto*
have *IH*:($\bigwedge\nu'$. $\bigwedge i$. *dsafe* (*args i*) $\Longrightarrow$
dterm-sem I (*Tsubst* (*args i*) σ) ν = *dterm-sem* (*adjoint I* σ ν) (*args i*) ν)
using *subst-dterm*[*OF good-interp TA safes ssafes*] **by** *auto*
have *eqs*:$\bigwedge i\ \nu'$. *dterm-sem I* (*Tsubst* (*args i*) σ) ν = *dterm-sem* (*adjoint I* σ ν) (*args i*) ν
by (*auto simp add*: *IH safes*)
let *?sub* = (λ *i*. *Tsubst* (*args i*) σ)
have *subSafe*:($\forall$ *i*. *dsafe* (*?sub i*))
by (*simp add*: *safes ssafes tsubst-preserves-safe*)
have *freef*:*fsafe* p' **using** *ssafe some* **unfolding** *ssafe-def* **by** *auto*
have *IH2*:($\nu \in$ *fml-sem I* (*FsubstFO* p' *?sub*)) = ($\nu \in$ *fml-sem* (*adjointFO I ?sub* ν) p')
by (*simp add*: *nsubst-fml* [*OF good-interp NFA freef subSafe*])
have *vec*:(χ *i*. *dterm-sem I* (*Tsubst* (*args i*) σ) ν) = (χ *i*. *dterm-sem* (*local.adjoint I* σ ν) (*args i*) ν)
apply(*auto simp add*: *vec-eq-iff*)
subgoal for *i*
using *IH*[*of i*, *OF safes*[*of i*]]
by *auto*
done
show *?thesis* ν
using *IH safes eqs* **apply** (*auto simp add*: *IH2 some good-interp*)
using *some* **unfolding** *adjoint-def adjointFO-def* **by** *auto*
qed
next
case (*Fadmit-Context1* σ φ *C C'*) **then**
have *FA*:*Fadmit* σ φ
and *FUA*:*FUadmit* σ φ *UNIV*
and *some*:*SContexts* σ *C* = *Some C'*
and *PFA*:*PFadmit* (λ-. *Fsubst* φ σ) *C'*
and *IH*:*fsafe* $\varphi \Longrightarrow$ *ssafe* $\sigma \Longrightarrow$ ($\bigwedge\nu$. ($\nu \in$ *fml-sem I* (*Fsubst* φ σ)) = ($\nu \in$ *fml-sem* (*local.adjoint I* σ ν) φ))
and *substSafe*:*fsafe*(*Fsubst* φ σ)
by *auto*
have *fsafe* (*InContext C* φ) $\Longrightarrow$ *ssafe* $\sigma \Longrightarrow$ ($\bigwedge\nu$. ($\nu \in$ *fml-sem I* (*Fsubst* (*InContext C* φ) σ)) = ($\nu \in$ *fml-sem* (*local.adjoint I* σ ν) (*InContext C* φ)))
proof −

assume *safe*:*fsafe* (*InContext C* φ)
from *safe* **have** *fsafe*:*fsafe* φ **by** (*auto dest*: *fsafe.cases*)
assume *ssafe*:*ssafe* σ
fix ν :: *'sz state*
have *agree*:$\bigwedge\omega$. *Vagree* ν ω (−*UNIV*) **unfolding** *Vagree-def* **by** *auto*
have *adj-eq*:$\bigwedge\omega$. *fml-sem* (*adjoint I* σ ν) φ = *fml-sem* (*adjoint I* σ ω) φ
using *uadmit-fml-adjoint*[*OF FUA agree fsafe ssafe good-interp*] **by** *auto*
have *eq*:(ν ∈ *fml-sem I* (*Fsubst* φ σ)) = (ν ∈ *fml-sem* (*local.adjoint I* σ ν) φ)
using *adj-eq IH*[*OF fsafe ssafe*] **by** *auto*
let *?sub* = (λ-. *Fsubst* φ σ)
let *?R1* = *fml-sem I* (*Fsubst* φ σ)
let *?R2* = *fml-sem* (*adjoint I* σ ν) φ
have *eq'*:*?R1* = *?R2*
using *adj-eq IH*[*OF fsafe ssafe*] **by** *auto*
have *freef*:*fsafe C'* **using** *ssafe some* **unfolding** *ssafe-def* **by** *auto*
have *IH2*:(ν ∈ *fml-sem I* (*PFsubst C' ?sub*)) = (ν ∈ *fml-sem* (*PFadjoint I ?sub*) *C'*)
using *psubst-fml good-interp PFA fsafe substSafe freef* **by** *blast*
have *IH'*:(⋀ν. (ν ∈ *fml-sem I* (*Fsubst* φ σ)) = (ν ∈ *fml-sem* (*adjoint I* σ ν) φ))
using *IH*[*OF fsafe ssafe*] **by** *auto*
then have *IH*:*fml-sem I* (*Fsubst* φ σ) = *fml-sem* (*adjoint I* σ ν) φ
using *eq'* **by** *blast*
have *duh*: (λ*f'* -. *fml-sem I* (*case* () *of* () ⇒ *Fsubst* φ σ)) = (λ *x* (). *fml-sem* (*local.adjoint I* σ ν) φ)
by (*simp add*: *case-unit-Unity eq' ext*)
have *extend-PF*:(*PFadjoint I ?sub*) = (*extendc I ?R2*)
unfolding *PFadjoint-def* **using** *IH* **apply** (*simp*)
by (*metis old.unit.case old.unit.exhaust*)
have (ν ∈ *fml-sem I* (*Fsubst* (*InContext C* φ) σ)) = (ν ∈ *fml-sem I* (*PFsubst C'* (λ-. *Fsubst* φ σ)))
using *some* **by** *simp*
moreover have ... = (ν ∈ *fml-sem* (*PFadjoint I ?sub*) *C'*)
using *IH2* **by** *auto*
moreover have ... = (ν ∈ *fml-sem* (*extendc I ?R2*) *C'*)
using *extend-PF* **by** *simp*
moreover have ... = (ν ∈ *fml-sem* (*extendc I ?R1*) *C'*)
using *eq'* **by** *auto*
moreover have ... = (ν ∈ *Contexts* (*adjoint I* σ ν) *C* (*fml-sem* (*adjoint I* σ ν) φ))
using *some* **unfolding** *adjoint-def* **apply** *auto*
apply (*simp add*: *eq' local.adjoint-def*)
by (*simp add*: *eq' local.adjoint-def*)
moreover have ... = (ν ∈ *fml-sem* (*adjoint I* σ ν) (*InContext C* φ))
by *auto*
ultimately
show (ν ∈ *fml-sem I* (*Fsubst* (*InContext C* φ) σ)) = (ν ∈ *fml-sem* (*local.adjoint I* σ ν) (*InContext C* φ))
by *blast*

```
  qed
  then show ?case by auto
next
  case (Fadmit-Context2 σ φ C) then
  have FA:Fadmit σ φ
  and FUA:FUadmit σ φ UNIV
  and none:SContexts σ C = None
  and IH:fsafe φ ⟹ ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst φ σ)) = (ν ∈
fml-sem (local.adjoint I σ ν) φ))
    by auto
  have fsafe (InContext C φ) ⟹
          ssafe σ ⟹ (⋀ν. (ν ∈ fml-sem I (Fsubst (InContext C φ) σ)) = (ν ∈
fml-sem (local.adjoint I σ ν) (InContext C φ)))
  proof −
    assume safe:fsafe (InContext C φ)
    then have fsafe:fsafe φ by (auto dest: fsafe.cases)
    assume ssafe:ssafe σ
    fix ν
    have Ieq: Contexts (local.adjoint I σ ν) C = Contexts I C
      using none unfolding adjoint-def by auto
    have IH':⋀ν. (ν ∈ fml-sem I (Fsubst φ σ)) = (ν ∈ fml-sem (local.adjoint I σ
ν) φ)
      using IH[OF fsafe ssafe] by auto
    have agree:⋀ω. Vagree ν ω (−UNIV) unfolding Vagree-def by auto
    have adj-eq:⋀ω. fml-sem (adjoint I σ ν) φ = fml-sem (adjoint I σ ω) φ
      using uadmit-fml-adjoint[OF FUA agree fsafe ssafe good-interp] by auto
    then have sem:fml-sem I (Fsubst φ σ) =  fml-sem (local.adjoint I σ ν) φ
      using IH' agree adj-eq by auto
    show ?thesis ν  using none Ieq sem by auto
  qed
  then show ?case by auto
qed

lemma subst-fml:
  fixes I::('sf, 'sc, 'sz) interp and ν::'sz state
  assumes good-interp:is-interp I
  assumes Fadmit:Fadmit σ φ
  assumes fsafe:fsafe φ
  assumes ssafe:ssafe σ
  shows (ν ∈ fml-sem I (Fsubst φ σ)) = (ν ∈ fml-sem (adjoint I σ ν) φ)
     using subst-fml-hp[OF good-interp] Fadmit fsafe ssafe by blast

lemma subst-fml-valid:
  fixes I::('sf, 'sc, 'sz) interp and ν::'sz state
  assumes Fadmit:Fadmit σ φ
  assumes fsafe:fsafe φ
  assumes ssafe:ssafe σ
  assumes valid:valid φ
  shows valid (Fsubst φ σ)
```

proof −
have *sub-sem*:$\bigwedge I\ \nu$. *is-interp* $I \Longrightarrow \nu \in$ *fml-sem* I (*Fsubst* $\varphi\ \sigma$)
proof −
fix I::('*sf*,'*sc*,'*sz*) *interp* **and** ν::'*sz state*
assume *good-interp*:*is-interp* I
have *good-adj*:*is-interp* (*adjoint* $I\ \sigma\ \nu$)
apply(*rule adjoint-safe*[*OF good-interp*])
using *ssafe* **unfolding** *ssafe-def* **by** *auto*
have φ*sem*:$\nu \in$ *fml-sem* (*adjoint* $I\ \sigma\ \nu$) φ **using** *valid* **using** *good-adj* **unfolding** *valid-def* **by** *blast*
then show *?thesis* $I\ \nu$
using *subst-fml*[*OF good-interp Fadmit fsafe ssafe*]
by *auto*
qed
then show *?thesis* **unfolding** *valid-def* **by** *blast*
qed

lemma *subst-sequent*:
fixes I::('*sf*, '*sc*, '*sz*) *interp* **and** ν::'*sz state*
assumes *good-interp*:*is-interp* I
assumes *Sadmit*:*Sadmit* σ (Γ,Δ)
assumes *Ssafe*:*Ssafe* (Γ,Δ)
assumes *ssafe*:*ssafe* σ
shows ($\nu \in$ *seq-sem* I (*Ssubst* (Γ,Δ) σ)) = ($\nu \in$ *seq-sem* (*adjoint* $I\ \sigma\ \nu$) (Γ,Δ))
proof −
let *?f* = (*seq2fml* (Γ, Δ))
have *subst-eqG*:*Fsubst* (*foldr* (&&) Γ *TT*) σ = *foldr* (&&) (*map* ($\lambda\varphi$. *Fsubst* φ σ) Γ) *TT*
by(*induction* Γ, *auto simp add*: *TT-def*)
have *subst-eqD*:*Fsubst* (*foldr* (||) Δ *FF*) σ = *foldr* (||) (*map* ($\lambda\varphi$. *Fsubst* φ σ) Δ) *FF*
by(*induction* Δ, *auto simp add*: *FF-def Or-def*)
have *subst-eq*:*Fsubst ?f* σ = (*seq2fml* (*Ssubst* (Γ, Δ) σ))
using *subst-eqG subst-eqD*
by (*auto simp add*: *Implies-def Or-def*)
have *fsafeG*:*fsafe* (*foldr* (&&) Γ *TT*)
using *Ssafe* **apply**(*induction* Γ, *auto simp add*: *Ssafe-def TT-def*)
by *fastforce*
have *fsafeD*:*fsafe* (*foldr* (||) Δ *FF*)
using *Ssafe Or-def* **apply**(*induction* Δ, *auto simp add*: *Ssafe-def FF-def Or-def*)
by *fastforce*
have *fsafe*:*fsafe ?f*
using *fsafeD fsafeG* **by** (*auto simp add*: *Implies-def Or-def*)
have *FadmitG*:*Fadmit* σ (*foldr* (&&) Γ *TT*)
using *Sadmit Or-def* **apply**(*induction* Γ, *auto simp add*: *Sadmit-def TT-def Or-def*)
by *fastforce*
have *FadmitD*:*Fadmit* σ (*foldr* (||) Δ *FF*)

using *Sadmit Or-def* **apply**(*induction* Δ, *auto simp add*: *Sadmit-def FF-def Or-def*)
by *fastforce*
have *Fadmit*:*Fadmit* σ *?f*
using *FadmitG FadmitD* **unfolding** *Implies-def*
by (*simp add*: *Implies-def Or-def*)
have ($\nu \in$ *fml-sem I* (*Fsubst ?f* σ))
$=$($\nu \in$ *fml-sem* (*adjoint I* σ ν) (*seq2fml* (Γ, Δ)))
using *subst-fml*[*OF good-interp Fadmit fsafe ssafe*]
by *auto*
then show *?thesis*
using *subst-eq* **by** *auto*
qed

11.6 Soundness of substitution rule

theorem *subst-rule*:
assumes *sound*:*sound R*
assumes *Radmit*:*Radmit* σ *R*
assumes *FVS*:*FVS* $\sigma = \{\}$
assumes *Rsafe*:*Rsafe R*
assumes *ssafe*:*ssafe* σ
shows *sound* (*Rsubst R* σ)
proof $-$
obtain *SG* **and** *C* **where** *Rdef*:$R = (SG, C)$ **by** (*cases R*, *auto*)
obtain SG' **and** C' **where** *Rdef'*:*Rsubst R* $\sigma = (SG', C')$ **by** (*cases R*, *auto*)
obtain ΓC **and** ΔC **where** *Cdef*:$C = (\Gamma C, \Delta C)$ **by** (*cases C*, *auto*)
obtain $\Gamma C'$ **and** $\Delta C'$ **where** C'*def*:$C' = (\Gamma C', \Delta C')$ **by** (*cases* C', *auto*)
have CC':(*Ssubst* (ΓC, ΔC) σ) $= (\Gamma C', \Delta C')$
using *Rdef Rdef' Cdef C'def* **by** *auto*
have $\bigwedge I\ \nu$. *is-interp I* $\Longrightarrow$ ($\bigwedge \Gamma\ \Delta\ \omega$. *List.member* SG' (Γ, Δ) $\Longrightarrow$ $\omega \in$ *seq-sem I* (Γ, Δ)) $\Longrightarrow$ $\nu \in$ *seq-sem I* C'
proof $-$
fix I::($'sf$,$'sc$,$'sz$) *interp* **and** ν::$'sz$ *state*
assume *good-interp*:*is-interp I*
assume *prems*:($\bigwedge \Gamma\ \Delta\ \omega$. *List.member* SG' (Γ, Δ) $\Longrightarrow$ $\omega \in$ *seq-sem I* (Γ, Δ))
have *good-interp'*:$\bigwedge \omega$. *is-interp* (*adjoint I* σ ω)
using *adjoint-safe*[*OF good-interp*] *ssafe*[*unfolded ssafe-def*] **by** *auto*
have *sound*:$\bigwedge \omega$. ($\bigwedge \varphi\ \nu$. *List.member SG* φ $\Longrightarrow$ $\nu \in$ *seq-sem* (*adjoint I* σ ω) φ) $\Longrightarrow$ $\omega \in$ *seq-sem* (*adjoint I* σ ω) (ΓC, ΔC)
using *soundD-memv*[*of SG C*] *sound good-interp' Rdef Cdef* **by** *auto*
have *SadmitC*:*Sadmit* σ (ΓC, ΔC)
using *Radmit* **unfolding** *Radmit-def Rdef Cdef* **by** *auto*
have *SsafeC*:*Ssafe* (ΓC, ΔC)
using *Rsafe* **unfolding** *Rsafe-def Rdef Cdef* **by** *auto*
have *seq-sem*:$\nu \in$ *seq-sem* (*adjoint I* σ ν) (ΓC, ΔC)
proof(*rule sound*)
fix S :: ($'sf$,$'sc$,$'sz$) *sequent* **and** ν'
assume *mem*:*List.member SG S*

```
    obtain ΓS ΔS where Sdef:S = (ΓS, ΔS) by (cases S, auto)
    from mem obtain di where di:di < length SG ∧ SG ! di = S
    by (meson in-set-conv-nth in-set-member)
    have SadmitS:Sadmit σ (ΓS, ΔS)
      using Rdef Sdef di Radmit Radmit-def by auto
    have SsafeS:Ssafe (ΓS, ΔS)
      using Rsafe unfolding Rsafe-def Rdef Cdef using Sdef mem di by auto
    have map-mem:⋀f L x. List.member L x ⟹ List.member (map f L) (f x)
      subgoal for f L x
        by (induction L, auto simp add: member-rec)
      done
    let ?S' = (Ssubst (ΓS, ΔS) σ)
    have eq:Ssubst S σ = (map (λφ. Fsubst φ σ) ΓS, map (λφ. Fsubst φ σ) ΔS)

      using Sdef by auto
    from Sdef have mem':List.member SG' (fst ?S', snd ?S')
      using mem Rdef Rdef' eq map-mem[of SG S (λx. Ssubst x σ)] by auto
    have ν' ∈ seq-sem I (fst ?S', snd ?S') by (rule prems[OF mem', of ν'])
    then have ν' ∈ seq-sem (adjoint I σ ν') S
      using subst-sequent[OF good-interp SadmitS SsafeS ssafe, of ν']
      Sdef by auto
    have VA:Vagree ν ν' (FVS σ) using FVS unfolding Vagree-def by auto
    show ν' ∈ seq-sem (local.adjoint I σ ν) S
      using adjoint-consequence VA ssafe[unfolded ssafe-def]
      by (metis ‹ν' ∈ seq-sem (local.adjoint I σ ν') S› dfree-is-dsafe)
    qed
  have ν ∈ seq-sem I (ΓC', ΔC')
   using subst-sequent[OF good-interp SadmitC SsafeC ssafe, of ν] seq-sem Cdef
C'def CC'
    by auto
  then show  ν ∈ seq-sem I C' using C'def by auto
  qed
 then show ?thesis
  apply(rule soundI-memv')
    using Rdef' by auto
qed

end end
theory Uniform-Renaming
imports
  Ordinary-Differential-Equations.ODE-Analysis
  Ids
  Lib
  Syntax
  Denotational-Semantics
  Frechet-Correctness
  Static-Semantics
  Coincidence
  Bound-Effect
```

begin context *ids* **begin**

12 Uniform and Bound Renaming

Definitions and soundness proofs for the renaming rules Uniform Renaming and Bound Renaming. Renaming in dL swaps the names of two variables x and y, as in the swap operator of Nominal Logic.

fun *swap* ::*'sz* ⇒ *'sz* ⇒ *'sz* ⇒ *'sz*
where *swap x y z* = (*if z* = *x then y else if z* = *y then x else z*)

12.1 Uniform Renaming Definitions

primrec *TUrename* :: *'sz* ⇒ *'sz* ⇒ (*'sf*, *'sz*) *trm* ⇒ (*'sf*, *'sz*) *trm*
where
TUrename x y (*Var z*) = *Var* (*swap x y z*)
| *TUrename x y* (*DiffVar z*) = *DiffVar* (*swap x y z*)
| *TUrename x y* (*Const r*) = (*Const r*)
| *TUrename x y* (*Function f args*) = *Function f* (λ*i*. *TUrename x y* (*args i*))
| *TUrename x y* (*Plus ϑ1 ϑ2*) = *Plus* (*TUrename x y ϑ1*) (*TUrename x y ϑ2*)
| *TUrename x y* (*Times ϑ1 ϑ2*) = *Times* (*TUrename x y ϑ1*) (*TUrename x y ϑ2*)
| *TUrename x y* (*Differential ϑ*) = *Differential* (*TUrename x y ϑ*)

primrec *OUrename* :: *'sz* ⇒ *'sz* ⇒ (*'sf*, *'sz*) *ODE* ⇒ (*'sf*, *'sz*) *ODE*
where
OUrename x y (*OVar c*) = *undefined*
| *OUrename x y* (*OSing z ϑ*) = *OSing* (*swap x y z*) (*TUrename x y ϑ*)
| *OUrename x y* (*OProd ODE1 ODE2*) = *OProd* (*OUrename x y ODE1*) (*OUrename x y ODE2*)

inductive *ORadmit* :: (*'sf*, *'sz*) *ODE* ⇒ *bool*
where
ORadmit-Sing:*ORadmit* (*OSing x ϑ*)
| *ORadmit-Prod*:*ORadmit ODE1* ⟹ *ORadmit ODE2* ⟹ *ORadmit* (*OProd ODE1 ODE2*)

primrec *PUrename* :: *'sz* ⇒ *'sz* ⇒ (*'sf*, *'sc*, *'sz*) *hp* ⇒ (*'sf*, *'sc*, *'sz*) *hp*
and *FUrename* :: *'sz* ⇒ *'sz* ⇒ (*'sf*, *'sc*, *'sz*) *formula* ⇒ (*'sf*, *'sc*, *'sz*) *formula*
where
PUrename x y (*Pvar a*) = *undefined*
| *PUrename x y* (*Assign z ϑ*) = *Assign* (*swap x y z*) (*TUrename x y ϑ*)
| *PUrename x y* (*DiffAssign z ϑ*) = *DiffAssign* (*swap x y z*) (*TUrename x y ϑ*)
| *PUrename x y* (*Test φ*) = *Test* (*FUrename x y φ*)
| *PUrename x y* (*EvolveODE ODE φ*) = *EvolveODE* (*OUrename x y ODE*) (*FUrename x y φ*)
| *PUrename x y* (*Choice a b*) = *Choice* (*PUrename x y a*) (*PUrename x y b*)
| *PUrename x y* (*Sequence a b*) = *Sequence* (*PUrename x y a*) (*PUrename x y b*)
| *PUrename x y* (*Loop a*) = *Loop* (*PUrename x y a*)

| *FUrename x y (Geq ϑ1 ϑ2) = Geq (TUrename x y ϑ1) (TUrename x y ϑ2)*
| *FUrename x y (Prop p args) = Prop p (λi. TUrename x y (args i))*
| *FUrename x y (Not φ) = Not (FUrename x y φ)*
| *FUrename x y (And φ ψ) = And (FUrename x y φ) (FUrename x y ψ)*
| *FUrename x y (Exists z φ) = Exists (swap x y z) (FUrename x y φ)*
| *FUrename x y (Diamond α φ) = Diamond (PUrename x y α) (FUrename x y φ)*
| *FUrename x y (InContext C φ) = undefined*

12.2 Uniform Renaming Admissibility

inductive *PRadmit* :: (*'sf*, *'sc*, *'sz*) *hp* ⇒ *bool*
  and *FRadmit* ::(*'sf*, *'sc*, *'sz*) *formula* ⇒ *bool*
where
  PRadmit-Assign:*PRadmit* (*Assign x ϑ*)
| *PRadmit-DiffAssign*:*PRadmit* (*DiffAssign x ϑ*)
| *PRadmit-Test*:*FRadmit φ* ⟹ *PRadmit* (*Test φ*)
| *PRadmit-EvolveODE*:*ORadmit ODE* ⟹ *FRadmit φ* ⟹ *PRadmit* (*EvolveODE ODE φ*)
| *PRadmit-Choice*:*PRadmit a* ⟹ *PRadmit b* ⟹ *PRadmit* (*Choice a b*)
| *PRadmit-Sequence*:*PRadmit a* ⟹ *PRadmit b* ⟹ *PRadmit* (*Sequence a b*)
| *PRadmit-Loop*:*PRadmit a* ⟹ *PRadmit* (*Loop a*)

| *FRadmit-Geq*:*FRadmit* (*Geq ϑ1 ϑ2*)
| *FRadmit-Prop*:*FRadmit* (*Prop p args*)
| *FRadmit-Not*:*FRadmit φ* ⟹ *FRadmit* (*Not φ*)
| *FRadmit-And*:*FRadmit φ* ⟹ *FRadmit ψ* ⟹ *FRadmit* (*And φ ψ*)
| *FRadmit-Exists*:*FRadmit φ* ⟹ *FRadmit* (*Exists x φ*)
| *FRadmit-Diamond*:*PRadmit α* ⟹ *FRadmit φ* ⟹ *FRadmit* (*Diamond α φ*)

inductive-simps
  FRadmit-box-simps[*simp*]: *FRadmit* (*Box a f*)
and *PRadmit-box-simps*[*simp*]: *PRadmit* (*Assign x e*)

definition *RSadj* :: *'sz* ⇒ *'sz* ⇒ *'sz simple-state* ⇒ *'sz simple-state*
where *RSadj x y ν* = (*χ z. ν $ (swap x y z)*)

definition *Radj* :: *'sz* ⇒ *'sz* ⇒ *'sz state* ⇒ *'sz state*
where *Radj x y ν* = (*RSadj x y* (*fst ν*), *RSadj x y* (*snd ν*))

lemma *SUren*: *sterm-sem I* (*TUrename x y ϑ*) *ν* = *sterm-sem I ϑ* (*RSadj x y ν*)
  by (*induction ϑ*, *auto simp add*: *RSadj-def*)

lemma *ren-preserves-dfree*:*dfree ϑ* ⟹ *dfree* (*TUrename x y ϑ*)
  by(*induction rule*: *dfree.induct*, *auto intro*: *dfree.intros*)

12.3 Uniform Renaming Soundness Proof and Lemmas

lemma *TUren-frechet*:
  assumes *good-interp*:*is-interp I*

shows *dfree ϑ ⟹ frechet I (TUrename x y ϑ) ν ν′ = frechet I ϑ (RSadj x y ν) (RSadj x y ν′)*
proof (*induction rule*: *dfree.induct*)
— There's got to be a more elegant proof of this...
case (*dfree-Var i*)
then show *?case*
unfolding *RSadj-def* **apply** *auto*
subgoal by (*metis vec-lambda-eta*)
subgoal
proof (*auto simp add*: *axis-def*)
assume *yx*:*y ≠ x*
have *a*:(*χ z. ν′ $ (if z = x then y else if z = y then x else z)) $ y = ν′ $ x*
by *simp*
show *ν′ · (χ i. if i = x then 1 else 0)*
= (χ z. ν′ $ (if z = x then y else if z = y then x else z)) · (χ i. if i = y then 1 else 0)
by (*metis* (*no-types*) *a axis-def inner-axis*)
qed
subgoal
proof −
have *⋀v s. v · (χ sa. if sa = (s::′sz) then 1 else 0) = v $ s*
subgoal for *v s*
using *inner-axis*[*of v s 1*]
by (*auto simp add*: *axis-def*)
done
then show *?thesis*
by (*auto simp add*: *axis-def*)
qed
subgoal
proof −
assume *a1*: *i ≠ y*
assume *a2*: *i ≠ x*
have *⋀v s. v · (χ sa. if sa = (s::′sz) then 1 else 0) = v $ s*
by (*metis* (*no-types*) *inner-axis axis-def inner-prod-eq*)
then show *?thesis*
using *a2 a1* **by** (*auto simp add*: *axis-def*)
qed
done
qed (*auto simp add*: *SUren good-interp is-interp-def*)

lemma *RSadj-fst*:*RSadj x y* (*fst ν*) = *fst* (*Radj x y ν*)
unfolding *RSadj-def Radj-def* **by** *auto*

lemma *RSadj-snd*:*RSadj x y* (*snd ν*) = *snd* (*Radj x y ν*)
unfolding *RSadj-def Radj-def* **by** *auto*

lemma *TUren*:
assumes *good-interp*:*is-interp I*
shows *dsafe ϑ ⟹ dterm-sem I* (*TUrename x y ϑ*) *ν* = *dterm-sem I ϑ* (*Radj x*

```
y ν)
proof (induction rule: dsafe.induct)
  case (dsafe-Diff ϑ)
  assume free:dfree ϑ
  show ?case
    apply (auto simp add: directional-derivative-def)
    using TUren-frechet[OF good-interp free, of x y fst ν snd ν]
     by (auto simp add: RSadj-fst RSadj-snd)
qed (auto simp add: Radj-def RSadj-def)

lemma adj-sum:RSadj x y (ν1 + ν2) = (RSadj x y ν1) + (RSadj x y ν2)
  unfolding RSadj-def apply auto apply (rule vec-extensionality)
  subgoal for i
    apply (cases i = x)
     apply (cases i = y)
      by auto
  done

lemma OUren: ORadmit ODE ⟹ ODE-sem I (OUrename x y ODE) ν = RSadj
x y (ODE-sem I ODE (RSadj x y ν))
proof (induction rule: ORadmit.induct)
  case (ORadmit-Prod ODE1 ODE2)
  then show ?case
    using adj-sum by auto
next
  case (ORadmit-Sing z ϑ)
  then show ?case
    by (rule vec-extensionality | auto simp add: SUren RSadj-def)+
qed

lemma state-eq:
  fixes ν ν' :: 'sz state
  shows (⋀i. (fst ν) $ i = (fst ν') $ i) ⟹ (⋀i. (snd ν) $ i = (snd ν') $ i) ⟹
ν  = ν'
  apply (cases ν, cases ν', auto)
   by(rule vec-extensionality, auto)+

lemma Radj-repv1:
  fixes x y z ::'sz
  shows (Radj x y (repv ν y r)) = repv (Radj x y ν) x r
  apply(rule state-eq)
   subgoal for i
     apply(cases i = x) apply (cases i = y)
       unfolding Radj-def RSadj-def by auto
  subgoal for i
    apply(cases i = x) apply (cases i = y)
      unfolding Radj-def RSadj-def by auto
  done
```

lemma *Radj-repv2*:
fixes *x y z* ::'*sz*
shows (*Radj x y* (*repv* ν *x r*)) = *repv* (*Radj x y* ν) *y r*
apply(*rule state-eq*)
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
unfolding *Radj-def RSadj-def* **by** *auto*
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
unfolding *Radj-def RSadj-def* **by** *auto*
done

lemma *Radj-repv3*:
fixes *x y z* ::'*sz*
assumes *zx*:*z* ≠ *x* **and** *zy*:*z* ≠ *y*
shows (*Radj x y* (*repv* ν *z r*)) = *repv* (*Radj x y* ν) *z r*
apply(*rule state-eq*)
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
using *zx zy* **unfolding** *Radj-def RSadj-def* **by** *auto*
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
using *zx zy* **unfolding** *Radj-def RSadj-def* **by** *auto*
done

lemma *Radj-repd1*:
fixes *x y z* ::'*sz*
shows (*Radj x y* (*repd* ν *y r*)) = *repd* (*Radj x y* ν) *x r*
apply(*rule state-eq*)
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
unfolding *Radj-def RSadj-def* **by** *auto*
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
unfolding *Radj-def RSadj-def* **by** *auto*
done

lemma *Radj-repd2*:
fixes *x y z* ::'*sz*
shows (*Radj x y* (*repd* ν *x r*)) = *repd* (*Radj x y* ν) *y r*
apply(*rule state-eq*)
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
unfolding *Radj-def RSadj-def* **by** *auto*
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
unfolding *Radj-def RSadj-def* **by** *auto*
done

lemma *Radj-repd3*:
fixes *x y z* ::'*sz*
assumes *zx*:$z \neq x$ **and** *zy*:$z \neq y$
shows (*Radj x y* (*repd* ν *z r*)) = *repd* (*Radj x y* ν) *z r*
apply(*rule state-eq*)
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
using *zx zy* **unfolding** *Radj-def RSadj-def* **by** *auto*
subgoal for *i*
apply(*cases i* = *x*) **apply** (*cases i* = *y*)
using *zx zy* **unfolding** *Radj-def RSadj-def* **by** *auto*
done

— i.e. shows *Radj x y* is a bijection for all *x y*
lemma *Radj-eq-iff*:(*a* = *b*) = ((*Radj x y a*) = (*Radj x y b*))
unfolding *Radj-def RSadj-def* **apply** *auto*
apply (*rule state-eq*)
apply (*smt* (*verit*))+
done

lemma *RSadj-cancel*:*RSadj x y* (*RSadj x y* ν) = ν
unfolding *RSadj-def* **apply** *auto*
apply(*rule vec-extensionality*)
by(*auto*)

lemma *Radj-cancel*:*Radj x y* (*Radj x y* ν) = ν
unfolding *Radj-def RSadj-def* **apply** *auto*
apply(*rule state-eq*)
subgoal for *i* **by**(*cases i* = *x*, *cases i* = *y*, *auto*)
subgoal for *i* **by**(*cases i* = *x*, *cases i* = *y*, *auto*)
done

lemma *OUrename-preserves-ODE-vars*:*ORadmit ODE* $\Longrightarrow$ {*z*. (*swap x y z*) $\in$ *ODE-vars I ODE*} = *ODE-vars I* (*OUrename x y ODE*)
apply(*induction rule*: *ORadmit.induct*)
subgoal for *xa* ϑ **by** *auto*
subgoal for *ODE1 ODE2*
proof −
assume *IH1*:{*z*. *swap x y z* $\in$ *ODE-vars I ODE1*} = *ODE-vars I* (*OUrename x y ODE1*)
assume *IH2*:{*z*. *swap x y z* $\in$ *ODE-vars I ODE2*} = *ODE-vars I* (*OUrename x y ODE2*)
have {*z*. *swap x y z* $\in$ *ODE-vars I* (*OProd ODE1 ODE2*)} =
{*z*. *swap x y z* $\in$ (*ODE-vars I ODE1* $\cup$ *ODE-vars I ODE2*)} **by** *auto*
moreover have ... = {*z*. *swap x y z* $\in$ (*ODE-vars I ODE1*)} $\cup$ {*z*. *swap x y z* $\in$ (*ODE-vars I ODE2*)} **by** *auto*
moreover have ... = *ODE-vars I* (*OUrename x y ODE1*) $\cup$ *ODE-vars I* (*OUrename x y ODE2*) **using** *IH1 IH2* **by** *auto*
moreover have ... = *ODE-vars I* (*OUrename x y* (*OProd ODE1 ODE2*)) **by**

auto
ultimately show {*z*. *swap x y z* ∈ *ODE-vars I* (*OProd ODE1 ODE2*)} = *ODE-vars I* (*OUrename x y* (*OProd ODE1 ODE2*))
by *blast*
qed
done

lemma *ren-proj*:(*RSadj x y a*) $ *z* = *a* $ (*swap x y z*)
unfolding *RSadj-def* **by** *simp*

lemma *swap-cancel*:*swap x y* (*swap x y z*) = *z*
by *auto*

lemma *mkv-lemma*:
assumes *ORA*:*ORadmit ODE*
shows *Radj x y* (*mk-v I* (*OUrename x y ODE*) (*a*, *b*) *c*) = *mk-v I ODE* (*RSadj x y a*, *RSadj x y b*) (*RSadj x y c*)
proof −
have *inner1*:(*mk-v I* (*OUrename x y ODE*) (*a*, *b*) *c*) = ((χ *i*. (*if i* ∈ *ODE-vars I* (*OUrename x y ODE*) *then c else a*) $ *i*), (χ *i*. (*if i* ∈ *ODE-vars I* (*OUrename x y ODE*) *then ODE-sem I* (*OUrename x y ODE*) *c else b*) $ *i*))
using *mk-v-concrete*[*of I OUrename x y ODE* (*a*,*b*) *c*] **by** *auto*
have *inner2*:(((χ *i*. (*if i* ∈ *ODE-vars I* (*OUrename x y ODE*) *then c else a*) $ *i*), (χ *i*. (*if i* ∈ *ODE-vars I* (*OUrename x y ODE*) *then ODE-sem I* (*OUrename x y ODE*) *c else b*) $ *i*)))
= (((χ *i*. (*if* (*swap x y i*) ∈ *ODE-vars I ODE then c else a*) $ *i*), (χ *i*. (*if* (*swap x y i*) ∈ *ODE-vars I ODE then ODE-sem I* (*OUrename x y ODE*) *c else b*) $ *i*)))
by (*force simp*: *OUrename-preserves-ODE-vars*[*OF ORA*, *symmetric*])
have *Radj x y* (*mk-v I* (*OUrename x y ODE*) (*a*, *b*) *c*) =
Radj x y (((χ *i*. (*if i* ∈ *ODE-vars I* (*OUrename x y ODE*) *then c else a*) $ *i*), (χ *i*. (*if i* ∈ *ODE-vars I* (*OUrename x y ODE*) *then ODE-sem I* (*OUrename x y ODE*) *c else b*) $ *i*)))
using *inner1* **by** *auto*
moreover have ... = *Radj x y* (((χ *i*. (*if* (*swap x y i*) ∈ *ODE-vars I ODE then c else a*) $ *i*),
(χ *i*. (*if* (*swap x y i*) ∈ *ODE-vars I ODE then ODE-sem I* (*OUrename x y ODE*) *c else b*) $ *i*)))
using *inner2* **by** *auto*
moreover have ... = (((χ *i*. (*if* (*swap x y* (*swap x y i*)) ∈ *ODE-vars I ODE then c else a*) $ (*swap x y i*))),
(χ *i*. (*if* (*swap x y* (*swap x y i*)) ∈ *ODE-vars I ODE then ODE-sem I* (*OUrename x y ODE*) *c else b*) $ (*swap x y i*)))
unfolding *Radj-def RSadj-def* **by** *auto*
moreover have ... = (((χ *i*. (*if i* ∈ *ODE-vars I ODE then c else a*) $ (*swap x y i*))),
(χ *i*. (*if i* ∈ *ODE-vars I ODE then ODE-sem I* (*OUrename x y ODE*) *c else b*) $ (*swap x y i*)))
using *swap-cancel* **by** *auto*

moreover have ... = (((χ *i*. (*if i* ∈ *ODE-vars I ODE then RSadj x y c else RSadj x y a*) $ *i*)),
(χ *i*. (*if i* ∈ *ODE-vars I ODE then RSadj x y* (*ODE-sem I* (*OUrename x y ODE*) *c*) *else RSadj x y b*) $ *i*))
by(*auto simp add*: *ren-proj*)
moreover have ... = (((χ *i*. (*if i* ∈ *ODE-vars I ODE then RSadj x y c else RSadj x y a*) $ *i*)),
(χ *i*. (*if i* ∈ *ODE-vars I ODE then RSadj x y* (*RSadj x y* (*ODE-sem I ODE* (*RSadj x y c*))) *else RSadj x y b*) $ *i*))
using *OUren*[*OF ORA*, *of I x y c*] **by** *auto*
moreover have ... = (((χ *i*. (*if i* ∈ *ODE-vars I ODE then RSadj x y c else RSadj x y a*) $ *i*)),
(χ *i*. (*if i* ∈ *ODE-vars I ODE then* (*ODE-sem I ODE* (*RSadj x y c*)) *else RSadj x y b*) $ *i*))
by(*auto simp add*: *RSadj-cancel*)
moreover have ... = *mk-v I ODE* (*RSadj x y a*, *RSadj x y b*) (*RSadj x y c*)
using *mk-v-concrete*[*of I ODE* (*RSadj x y a*, *RSadj x y b*) *RSadj x y c*]
by *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *sol-lemma*:
assumes *ORA*:*ORadmit ODE*
assumes *t*:*0* ≤ *t*
assumes *fml*:⋀ν. (ν ∈ *fml-sem I* (*FUrename x y φ*)) = (*Radj x y* ν ∈ *fml-sem I φ*)
assumes *sol*:(*sol solves-ode* (λ*a*. *ODE-sem I* (*OUrename x y ODE*))) {*0*..*t*} {*xa*. *mk-v I* (*OUrename x y ODE*) (*sol 0*, *b*) *xa* ∈ *fml-sem I* (*FUrename x y φ*)}
shows ((λ*t*. *RSadj x y* (*sol t*)) *solves-ode* (λ*a*. *ODE-sem I ODE*)) {*0*..*t*} {*xa*. *mk-v I ODE* (*RSadj x y* (*sol 0*), *RSadj x y b*) *xa* ∈ *fml-sem I φ*}
apply(*unfold solves-ode-def*)
apply(*rule conjI*)
defer
subgoal
apply *auto*
proof −
fix *s*
assume *t*:*0* ≤ *s s* ≤ *t*
have *ivl*:*s* ∈ {*0*..*t*} **using** *t* **by** *auto*
have *mk-v I* (*OUrename x y ODE*) (*sol 0*,*b*) (*sol s*) ∈ *fml-sem I* (*FUrename x y φ*)
using *solves-odeD*(*2*)[*OF sol ivl*] **by** *auto*
then have *Radj x y* (*mk-v I* (*OUrename x y ODE*) (*sol 0*, *b*) (*sol s*)) ∈ *fml-sem I φ*
using *fml*[*of mk-v I* (*OUrename x y ODE*) (*sol 0*, *b*) (*sol s*)] **by** *auto*
then show *mk-v I ODE* (*RSadj x y* (*sol 0*), *RSadj x y b*) (*RSadj x y* (*sol s*)) ∈ *fml-sem I φ*
using *mkv-lemma*[*OF ORA*, *of x y I sol 0 b sol s*] **by** *auto*
qed

```
  apply (unfold has-vderiv-on-def has-vector-derivative-def)
  proof −
    have ⋀s. s∈{0..t} ⟹ ((λt. RSadj x y (sol t)) has-derivative (λxb. xb *R
ODE-sem I ODE (RSadj x y (sol s)))) (at s within {0..t})
    proof −
      fix s
      assume s:s ∈{0..t}
      let ?g = RSadj x y
      let ?g' = RSadj x y
      let ?f = sol
      let ?f' = (λt'. t' *R ODE-sem I (OUrename x y ODE) (sol s))
      let ?h = ?g ∘ ?f

      have fun-eq:(λt'. t' *R ODE-sem I (OUrename x y ODE) (sol s)) = (λt'. t'
*R (RSadj x y (ODE-sem I ODE (RSadj x y (sol s)))))
        apply(rule ext)
        using OUren[OF ORA, of I x y] by simp
      have fun-eq1:(λν. (χ i. RSadj x y ν $ i)) = RSadj x y
        by(rule ext, rule vec-extensionality, simp)
      have s ∈ {0..t} ⟹ (sol has-derivative (λt'. t' *R ODE-sem I (OUrename
x y ODE) (sol s))) (at s within {0..t})
      using solves-odeD(1)[OF sol] unfolding has-vderiv-on-def has-vector-derivative-def
by auto
      then have fderiv:s ∈ {0..t} ⟹ (?f has-derivative ?f') (at s within {0..t})
        using fun-eq by auto
      have ((λν. (χ i. RSadj x y ν $ i)) has-derivative (λν'. (χ i . RSadj x y ν' $
i))) (at (?f s) within ?f ‘ {0..t})
        apply(rule has-derivative-vec)
        apply(auto simp add: RSadj-def intro:derivative-eq-intros)
          by (simp add: has-derivative-at-withinI has-derivative-proj')+
      then have gderiv:(RSadj x y has-derivative (RSadj x y)) (at (?f s) within ?f
‘ {0..t})
        using fun-eq1 by auto
      have hderiv:(?h has-derivative (?g' ∘ ?f')) (at s within {0..t})
        by (rule diff-chain-within[OF fderiv gderiv], rule s)
      have heq:(λt. RSadj x y (sol t)) = ?h
        unfolding comp-def by simp
      have RSadj-scale:⋀c a. RSadj x y (c *R RSadj x y a) = c *R a
        subgoal for c a
          unfolding RSadj-def
          apply auto
          apply(rule vec-extensionality)
          by(auto)
        done
      have heq':(λxb. xb *R ODE-sem I ODE (RSadj x y (sol s))) = (?g' ∘ ?f')
       unfolding comp-def apply(rule ext) using OUren[OF ORA, of I x y sol s]
        apply auto
        subgoal for c
           using RSadj-scale[of c ODE-sem I ODE (RSadj x y (sol s))] by auto
```

done
show $((\lambda t.\ RSadj\ x\ y\ (sol\ t))\ has\text{-}derivative\ (\lambda xb.\ xb *_R ODE\text{-}sem\ I\ ODE\ (RSadj\ x\ y\ (sol\ s))))\ (at\ s\ within\ \{0..t\})$
using *heq heq' hderiv* **by** *auto*
qed
then show $\forall xa{\in}\{0..t\}.\ ((\lambda t.\ RSadj\ x\ y\ (sol\ t))\ has\text{-}derivative\ (\lambda xb.\ xb *_R ODE\text{-}sem\ I\ ODE\ (RSadj\ x\ y\ (sol\ xa))))\ (at\ xa\ within\ \{0..t\})$
by *auto*
qed

lemma *sol-lemma2*:
assumes *ORA*:*ORadmit ODE*
assumes $t{:}0 \le t$
assumes *fml*:$\bigwedge\nu.\ (\nu \in fml\text{-}sem\ I\ (FUrename\ x\ y\ \varphi)) = (Radj\ x\ y\ \nu \in fml\text{-}sem\ I\ \varphi)$
assumes *sol*:$(sol\ solves\text{-}ode\ (\lambda a.\ ODE\text{-}sem\ I\ ODE))\ \{0..t\}\ \{x.\ mk\text{-}v\ I\ ODE\ (sol\ 0,\ b)\ x \in fml\text{-}sem\ I\ \varphi\}$
shows $((\lambda t.\ RSadj\ x\ y\ (sol\ t))\ solves\text{-}ode\ (\lambda a.\ ODE\text{-}sem\ I\ (OUrename\ x\ y\ ODE)))\ \{0..t\}$
$\{xa.\ mk\text{-}v\ I\ (OUrename\ x\ y\ ODE)\ (RSadj\ x\ y\ (sol\ 0),\ RSadj\ x\ y\ b)\ xa \in fml\text{-}sem\ I\ (FUrename\ x\ y\ \varphi)\}$
apply(*unfold solves-ode-def*)
apply(*rule conjI*)
defer
subgoal
apply *auto*
proof −
fix *s*
assume $t{:}0 \le s\ s \le t$
have *ivl*:$s \in \{0..t\}$ **using** *t* **by** *auto*
have *mk-v I ODE* $(sol\ 0,b)\ (sol\ s) \in fml\text{-}sem\ I\ \varphi$
using *solves-odeD*(2)[*OF sol ivl*] **by** *auto*
then have $Radj\ x\ y\ (mk\text{-}v\ I\ ODE\ (sol\ 0,\ b)\ (sol\ s)) \in fml\text{-}sem\ I\ (FUrename\ x\ y\ \varphi)$
using *Radj-cancel*[*of x y* (*mk-v I ODE* (*sol 0*, *b*) (*sol s*))]
by (*simp add*: *fml*)
then show $mk\text{-}v\ I\ (OUrename\ x\ y\ ODE)\ (RSadj\ x\ y\ (sol\ 0),\ RSadj\ x\ y\ b)\ (RSadj\ x\ y\ (sol\ s)) \in fml\text{-}sem\ I\ (FUrename\ x\ y\ \varphi)$
using *mkv-lemma*[*OF ORA*, *of x y I RSadj x y* (*sol 0*) *RSadj x y b RSadj x y* (*sol s*)]
by (*simp add*: *RSadj-cancel* ‹*mk-v I ODE* (*sol 0*, *b*) (*sol s*) ∈ *fml-sem I* φ› *fml*)
qed
apply (*unfold has-vderiv-on-def has-vector-derivative-def*)
proof −
have $\bigwedge s.\ s{\in}\{0..t\} \Longrightarrow\ ((\lambda t.\ RSadj\ x\ y\ (sol\ t))\ has\text{-}derivative\ (\lambda xb.\ xb *_R ODE\text{-}sem\ I\ (OUrename\ x\ y\ ODE)\ (RSadj\ x\ y\ (sol\ s))))\ (at\ s\ within\ \{0..t\})$
proof −

fix s
assume s:$s \in \{0..t\}$
let $?g = RSadj\ x\ y$
let $?g' = RSadj\ x\ y$
let $?f = sol$
let $?f' = (\lambda xb.\ xb *_R RSadj\ x\ y\ (ODE\text{-}sem\ I\ (OUrename\ x\ y\ ODE)\ (RSadj\ x\ y\ (sol\ s))))$
let $?h = ?g \circ ?f$
have *fun-eq*:$(\lambda t'.\ t' *_R ODE\text{-}sem\ I\ ODE\ (sol\ s)) = (\lambda xb.\ xb *_R RSadj\ x\ y\ (ODE\text{-}sem\ I\ (OUrename\ x\ y\ ODE)\ (RSadj\ x\ y\ (sol\ s))))$
apply(*rule ext*)
using *OUren*[*OF ORA*, *of I x y*, *of RSadj x y* (*sol s*)] *RSadj-cancel* **by** *simp*
have *fun-eq1*:$(\lambda \nu.\ (\chi\ i.\ RSadj\ x\ y\ \nu\ \$\ i)) = RSadj\ x\ y$
by(*rule ext*, *rule vec-extensionality*, *simp*)
have $s \in \{0..t\} \Longrightarrow (sol\ has\text{-}derivative\ (\lambda t'.\ t' *_R ODE\text{-}sem\ I\ ODE\ (sol\ s)))\ (at\ s\ within\ \{0..t\})$
using *solves-odeD*(*1*)[*OF sol*] **unfolding** *has-vderiv-on-def has-vector-derivative-def* **by** *auto*
then have *fderiv*:$s \in \{0..t\} \Longrightarrow (?f\ has\text{-}derivative\ ?f')\ (at\ s\ within\ \{0..t\})$
using *fun-eq* **by** *auto*
have $((\lambda \nu.\ (\chi\ i.\ RSadj\ x\ y\ \nu\ \$\ i))\ has\text{-}derivative\ (\lambda \nu'.\ (\chi\ i\ .\ RSadj\ x\ y\ \nu'\ \$\ i)))\ (at\ (?f\ s)\ within\ ?f\ `\ \{0..t\})$
apply(*rule has-derivative-vec*)
apply(*auto simp add*: *RSadj-def intro*:*derivative-eq-intros*)
by (*simp add*: *has-derivative-at-withinI has-derivative-proj'*)+
then have *gderiv*:$(RSadj\ x\ y\ has\text{-}derivative\ (RSadj\ x\ y))\ (at\ (?f\ s)\ within\ ?f\ `\ \{0..t\})$
using *fun-eq1* **by** *auto*
have *hderiv*:$(?h\ has\text{-}derivative\ (?g' \circ ?f'))\ (at\ s\ within\ \{0..t\})$
by (*rule diff-chain-within*[*OF fderiv gderiv*], *rule s*)
have *heq*:$(\lambda t.\ RSadj\ x\ y\ (sol\ t)) = ?h$
unfolding *comp-def* **by** *simp*
have *RSadj-scale*:$\bigwedge c\ a.\ RSadj\ x\ y\ (c *_R RSadj\ x\ y\ a) = c *_R a$
subgoal for *c a*
unfolding *RSadj-def*
apply *auto*
apply(*rule vec-extensionality*)
by(*auto*)
done
have *heq'*:$(\lambda xb.\ xb *_R ODE\text{-}sem\ I\ (OUrename\ x\ y\ ODE)\ (RSadj\ x\ y\ (sol\ s))) = (?g' \circ ?f')$
unfolding *comp-def* **apply**(*rule ext*) **using** *OUren*[*OF ORA*, *of I x y RSadj x y* (*sol s*)]
apply *auto*
subgoal for *c*
using *RSadj-scale*[*of c ODE-sem I* (*OUrename x y ODE*) (*RSadj x y* (*sol s*))] *RSadj-cancel*[*of x y sol s*]
RSadj-cancel[*of x y ODE-sem I ODE* (*sol s*)] **by** *auto*
done

show ((λt. *RSadj x y* (*sol t*)) *has-derivative* (λxb. *xb* $*_R$ *ODE-sem I* (*OUrename x y ODE*) (*RSadj x y* (*sol s*)))) (*at s within* {*0..t*})
using *heq heq′ hderiv* **by** *auto*
qed
then show ∀ *xa*∈{*0..t*}. ((λt. *RSadj x y* (*sol t*)) *has-derivative* (λxb. *xb* $*_R$ *ODE-sem I* (*OUrename x y ODE*) (*RSadj x y* (*sol xa*)))) (*at xa within* {*0..t*})
by *blast*
qed

lemma *PUren-FUren*:
assumes *good-interp*:*is-interp I*
shows
(*PRadmit* α ⟶ *hpsafe* α ⟶ (∀ ν ω. (ν, ω) ∈ *prog-sem I* (*PUrename x y* α) ⟷ (*Radj x y* ν, *Radj x y* ω) ∈ *prog-sem I* α))
∧ (*FRadmit* φ ⟶ *fsafe* φ ⟶ (∀ ν. ν ∈ *fml-sem I* (*FUrename x y* φ) ⟷ (*Radj x y* ν) ∈ *fml-sem I* φ))
proof(*induction rule*: *PRadmit-FRadmit.induct*)
case (*FRadmit-Geq* ϑ1 ϑ2)
then show *?case* **using** *TUren*[*OF good-interp*] **by** *auto*
next
case (*FRadmit-Exists* φ *z*) **then have**
FRA:*FRadmit* φ
and *IH*:*fsafe* φ ⟹ (⋀ν. (ν ∈ *fml-sem I* (*FUrename x y* φ)) = (*Radj x y* ν ∈ *fml-sem I* φ))
by *auto*
have *fsafe* (*Exists z* φ) ⟹ (⋀ν. (ν ∈ *fml-sem I* (*FUrename x y* (*Exists z* φ))) = (*Radj x y* ν ∈ *fml-sem I* (*Exists z* φ)))
apply (*cases z* = *x*)
subgoal for ν
proof −
assume *fsafe*:*fsafe* (*Exists z* φ)
assume *zx*:*z* = *x*
from *fsafe* **have** *fsafe′*:*fsafe* φ **by** *auto*
have *IH′*:(⋀ν. (ν ∈ *fml-sem I* (*FUrename x y* φ)) = (*Radj x y* ν ∈ *fml-sem I* φ))
by (*rule IH*[*OF fsafe′*])
have (ν ∈ *fml-sem I* (*FUrename x y* (*Exists z* φ))) = (ν ∈ *fml-sem I* (*Exists y* (*FUrename x y* φ))) **using** *zx* **by** *auto*
moreover have ... = (∃ *r*. (*repv* ν *y r*) ∈ *fml-sem I* (*FUrename x y* φ)) **by** *auto*
moreover have ... = (∃ *r*. (*Radj x y* (*repv* ν *y r*)) ∈ *fml-sem I* φ) **using** *IH′* **by** *auto*
moreover have ... = (∃ *r*. (*repv* (*Radj x y* ν) *x r*) ∈ *fml-sem I* φ) **using** *Radj-repv1*[*of x y* ν] **by** *auto*
moreover have ... = (*Radj x y* ν ∈ *fml-sem I* (*Exists z* φ)) **using** *zx* **by** *auto*
ultimately
show (ν ∈ *fml-sem I* (*FUrename x y* (*Exists z* φ))) = (*Radj x y* ν ∈ *fml-sem I* (*Exists z* φ))

by *auto*
qed
apply (*cases* $z = y$)
subgoal for ν
proof −
assume *fsafe*:*fsafe* (*Exists* z φ)
assume *zx*:$z = y$
from *fsafe* **have** *fsafe'*:*fsafe* φ **by** *auto*
have *IH'*:($\bigwedge\nu$. ($\nu \in$ *fml-sem* I (*FUrename* x y φ)) = (*Radj* x y $\nu \in$ *fml-sem* I φ))
by (*rule IH*[*OF fsafe'*])
have ($\nu \in$ *fml-sem* I (*FUrename* x y (*Exists* z φ))) = ($\nu \in$ *fml-sem* I (*Exists* x (*FUrename* x y φ))) **using** *zx* **by** *auto*
moreover have ... = ($\exists\, r$. (*repv* ν x r) $\in$ *fml-sem* I (*FUrename* x y φ)) **by** *auto*
moreover have ... = ($\exists\, r$. (*Radj* x y (*repv* ν x r)) $\in$ *fml-sem* I φ) **using** *IH'* **by** *auto*
moreover have ... = ($\exists\, r$. (*repv* (*Radj* x y ν) y r) $\in$ *fml-sem* I φ) **using** *Radj-repv2*[*of* x y ν] **by** *auto*
moreover have ... = (*Radj* x y $\nu \in$ *fml-sem* I (*Exists* z φ)) **using** *zx* **by** *auto*
ultimately
show ($\nu \in$ *fml-sem* I (*FUrename* x y (*Exists* z φ))) = (*Radj* x y $\nu \in$ *fml-sem* I (*Exists* z φ))
by *auto*
qed
subgoal for ν
proof −
assume *fsafe*:*fsafe* (*Exists* z φ)
assume *zx*:$z \neq x$ **and** *zy*:$z \neq y$
from *fsafe* **have** *fsafe'*:*fsafe* φ **by** *auto*
have *IH'*:($\bigwedge\nu$. ($\nu \in$ *fml-sem* I (*FUrename* x y φ)) = (*Radj* x y $\nu \in$ *fml-sem* I φ))
by (*rule IH*[*OF fsafe'*])
have ($\nu \in$ *fml-sem* I (*FUrename* x y (*Exists* z φ))) = ($\nu \in$ *fml-sem* I (*Exists* z (*FUrename* x y φ))) **using** *zx zy* **by** *auto*
moreover have ... = ($\exists\, r$. (*repv* ν z r) $\in$ *fml-sem* I (*FUrename* x y φ)) **by** *auto*
moreover have ... = ($\exists\, r$. (*Radj* x y (*repv* ν z r)) $\in$ *fml-sem* I φ) **using** *IH'* **by** *auto*
moreover have ... = ($\exists\, r$. (*repv* (*Radj* x y ν) z r) $\in$ *fml-sem* I φ) **using** *Radj-repv3*[*of* z x y ν, *OF zx zy*] **by** *auto*
moreover have ... = (*Radj* x y $\nu \in$ *fml-sem* I (*Exists* z φ)) **using** *zx* **by** *auto*
ultimately
show ($\nu \in$ *fml-sem* I (*FUrename* x y (*Exists* z φ))) = (*Radj* x y $\nu \in$ *fml-sem* I (*Exists* z φ))
by *auto*
qed
done

then show *?case* **by** *auto*
next
case (*PRadmit-Assign z ϑ*)
have *hpsafe* (*Assign z ϑ*) ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*Assign z ϑ*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*Assign z ϑ*)))
apply (*cases z* = *x*)
subgoal for *ν ω*
proof −
assume *fsafe*:*hpsafe* (*Assign z ϑ*)
assume *zx*:*z* = *x*
from *fsafe* **have** *dsafe*:*dsafe ϑ* **by** *auto*
have *IH′*:(⋀*ν*. *dterm-sem I* (*TUrename x y ϑ*) *ν* = *dterm-sem I ϑ* (*Radj x y ν*))
subgoal for *ν* **using** *TUren*[*OF good-interp dsafe* , *of x y ν*] **by** *auto* **done**
have ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*Assign z ϑ*))) = ((*ν*, *ω*) ∈ *prog-sem I* (*Assign y* (*TUrename x y ϑ*))) **using** *zx* **by** *auto*
moreover have ... = (*ω* = *repv ν y* (*dterm-sem I* (*TUrename x y ϑ*) *ν*)) **by** *auto*
moreover have ... = (*ω* = *repv ν y* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *IH′* **by** *auto*
moreover have ... = (*Radj x y ω* = *Radj x y* (*repv ν y* (*dterm-sem I ϑ* (*Radj x y ν*)))) **using** *Radj-eq-iff* **by** *auto*
moreover have ... = (*Radj x y ω* = *repv* (*Radj x y ν*) *x* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *Radj-repv1* **by** *auto*
moreover have ... = (*Radj x y ω* = *repv* (*Radj x y ν*) *z* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *zx* **by** *auto*
moreover have ... = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*Assign z ϑ*)) **by** *auto*
ultimately
show ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*Assign z ϑ*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*Assign z ϑ*))
by *auto*
qed
apply (*cases z* = *y*)
subgoal for *ν ω*
proof −
assume *fsafe*:*hpsafe* (*Assign z ϑ*)
assume *zy*:*z* = *y*
from *fsafe* **have** *dsafe*:*dsafe ϑ* **by** *auto*
have *IH′*:(⋀*ν*. *dterm-sem I* (*TUrename x y ϑ*) *ν* = *dterm-sem I ϑ* (*Radj x y ν*))
subgoal for *ν* **using** *TUren*[*OF good-interp dsafe* , *of x y ν*] **by** *auto* **done**
have ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*Assign z ϑ*))) = ((*ν*, *ω*) ∈ *prog-sem I* (*Assign x* (*TUrename x y ϑ*))) **using** *zy* **by** *auto*
moreover have ... = (*ω* = *repv ν x* (*dterm-sem I* (*TUrename x y ϑ*) *ν*)) **by** *auto*
moreover have ... = (*ω* = *repv ν x* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *IH′* **by** *auto*
moreover have ... = (*Radj x y ω* = *Radj x y* (*repv ν x* (*dterm-sem I ϑ*

(*Radj x y ν*)))) **using** *Radj-eq-iff* **by** *auto*
moreover have ... = (*Radj x y ω* = *repv* (*Radj x y ν*) *y* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *Radj-repv2* **by** *auto*
moreover have ... = (*Radj x y ω* = *repv* (*Radj x y ν*) *z* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *zy* **by** *auto*
moreover have ... = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*Assign z ϑ*)) **by** *auto*
ultimately
show ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*Assign z ϑ*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*Assign z ϑ*))
by *auto*
qed
subgoal for *ν ω*
proof −
assume *fsafe*:*hpsafe* (*Assign z ϑ*)
assume *zx*:*z* ≠ *x* **and** *zy*:*z* ≠ *y*
from *fsafe* **have** *dsafe*:*dsafe ϑ* **by** *auto*
have *IH′*:(⋀*ν*. *dterm-sem I* (*TUrename x y ϑ*) *ν* = *dterm-sem I ϑ* (*Radj x y ν*))
subgoal for *ν* **using** *TUren*[*OF good-interp dsafe , of x y ν*] **by** *auto* **done**
have ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*Assign z ϑ*))) = ((*ν*, *ω*) ∈ *prog-sem I* (*Assign z* (*TUrename x y ϑ*))) **using** *zx zy* **by** *auto*
moreover have ... = (*ω* = *repv ν z* (*dterm-sem I* (*TUrename x y ϑ*) *ν*)) **by** *auto*
moreover have ... = (*ω* = *repv ν z* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *IH′* **by** *auto*
moreover have ... = (*Radj x y ω* = *Radj x y* (*repv ν z* (*dterm-sem I ϑ* (*Radj x y ν*)))) **using** *Radj-eq-iff* **by** *auto*
moreover have ... = (*Radj x y ω* = *repv* (*Radj x y ν*) *z* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *Radj-repv3*[*OF zx zy*] **by** *auto*
moreover have ... = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*Assign z ϑ*)) **by** *auto*
ultimately
show ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*Assign z ϑ*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*Assign z ϑ*))
by *auto*
qed
done
then show *?case* **by** *auto*
next
case (*PRadmit-DiffAssign z ϑ*)
have *hpsafe* (*DiffAssign z ϑ*) ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*DiffAssign z ϑ*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*DiffAssign z ϑ*)))
apply (*cases z* = *x*)
subgoal for *ν ω*
proof −
assume *fsafe*:*hpsafe* (*DiffAssign z ϑ*)
assume *zx*:*z* = *x*
from *fsafe* **have** *dsafe*:*dsafe ϑ* **by** *auto*

have *IH'*:($\bigwedge\nu$. *dterm-sem I* (*TUrename x y ϑ*) *ν* = *dterm-sem I ϑ* (*Radj x y ν*))
subgoal for *ν* **using** *TUren*[*OF good-interp dsafe , of x y ν*] **by** *auto* **done**
have ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*DiffAssign z ϑ*))) = ((*ν*, *ω*) ∈ *prog-sem I* (*DiffAssign y* (*TUrename x y ϑ*))) **using** *zx* **by** *auto*
moreover have ... = (*ω* = *repd ν y* (*dterm-sem I* (*TUrename x y ϑ*) *ν*)) **by** *auto*
moreover have ... = (*ω* = *repd ν y* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *IH'* **by** *auto*
moreover have ... = (*Radj x y ω* = *Radj x y* (*repd ν y* (*dterm-sem I ϑ* (*Radj x y ν*)))) **using** *Radj-eq-iff* **by** *auto*
moreover have ... = (*Radj x y ω* = *repd* (*Radj x y ν*) *x* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *Radj-repd1* **by** *auto*
moreover have ... = (*Radj x y ω* = *repd* (*Radj x y ν*) *z* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *zx* **by** *auto*
moreover have ... = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*DiffAssign z ϑ*)) **by** *auto*
ultimately
show ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*DiffAssign z ϑ*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*DiffAssign z ϑ*))
by *auto*
qed
apply (*cases z* = *y*)
subgoal for *ν ω*
proof −
assume *fsafe*:*hpsafe* (*DiffAssign z ϑ*)
assume *zy*:*z* = *y*
from *fsafe* **have** *dsafe*:*dsafe ϑ* **by** *auto*
have *IH'*:($\bigwedge\nu$. *dterm-sem I* (*TUrename x y ϑ*) *ν* = *dterm-sem I ϑ* (*Radj x y ν*))
subgoal for *ν* **using** *TUren*[*OF good-interp dsafe , of x y ν*] **by** *auto* **done**
have ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*DiffAssign z ϑ*))) = ((*ν*, *ω*) ∈ *prog-sem I* (*DiffAssign x* (*TUrename x y ϑ*))) **using** *zy* **by** *auto*
moreover have ... = (*ω* = *repd ν x* (*dterm-sem I* (*TUrename x y ϑ*) *ν*)) **by** *auto*
moreover have ... = (*ω* = *repd ν x* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *IH'* **by** *auto*
moreover have ... = (*Radj x y ω* = *Radj x y* (*repd ν x* (*dterm-sem I ϑ* (*Radj x y ν*)))) **using** *Radj-eq-iff* **by** *auto*
moreover have ... = (*Radj x y ω* = *repd* (*Radj x y ν*) *y* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *Radj-repd2* **by** *auto*
moreover have ... = (*Radj x y ω* = *repd* (*Radj x y ν*) *z* (*dterm-sem I ϑ* (*Radj x y ν*))) **using** *zy* **by** *auto*
moreover have ... = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*DiffAssign z ϑ*)) **by** *auto*
ultimately
show ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*DiffAssign z ϑ*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*DiffAssign z ϑ*))
by *auto*

qed
subgoal for ν ω
proof −
assume *fsafe*:*hpsafe* (*DiffAssign z* ϑ)
assume *zx*:$z \neq x$ **and** *zy*:$z \neq y$
from *fsafe* **have** *dsafe*:*dsafe* ϑ **by** *auto*
have *IH'*:($\bigwedge\nu$. *dterm-sem I* (*TUrename x y* ϑ) ν = *dterm-sem I* ϑ (*Radj x y* ν))
subgoal for ν **using** *TUren*[*OF good-interp dsafe , of x y* ν] **by** *auto* **done**
have ((ν, ω) ∈ *prog-sem I* (*PUrename x y* (*DiffAssign z* ϑ))) = ((ν, ω) ∈ *prog-sem I* (*DiffAssign z* (*TUrename x y* ϑ))) **using** *zx zy* **by** *auto*
moreover have ... = (ω = *repd* ν *z* (*dterm-sem I* (*TUrename x y* ϑ) ν)) **by** *auto*
moreover have ... = (ω = *repd* ν *z* (*dterm-sem I* ϑ (*Radj x y* ν))) **using** *IH'* **by** *auto*
moreover have ... = (*Radj x y* ω = *Radj x y* (*repd* ν *z* (*dterm-sem I* ϑ (*Radj x y* ν)))) **using** *Radj-eq-iff* **by** *auto*
moreover have ... = (*Radj x y* ω = *repd* (*Radj x y* ν) *z* (*dterm-sem I* ϑ (*Radj x y* ν))) **using** *Radj-repd3*[*OF zx zy*] **by** *auto*
moreover have ... = ((*Radj x y* ν, *Radj x y* ω) ∈ *prog-sem I* (*DiffAssign z* ϑ)) **by** *auto*
ultimately
show ((ν, ω) ∈ *prog-sem I* (*PUrename x y* (*DiffAssign z* ϑ))) = ((*Radj x y* ν, *Radj x y* ω) ∈ *prog-sem I* (*DiffAssign z* ϑ))
by *auto*
qed
done
then show *?case* **by** *auto*
next
case (*PRadmit-Test* φ) **then**
have *FRA*:*FRadmit* φ
and *IH*:*fsafe* φ ⟹ ($\bigwedge\nu$. (ν ∈ *fml-sem I* (*FUrename x y* φ)) = (*Radj x y* ν ∈ *fml-sem I* φ))
by *auto*
have *hpsafe* (*?* φ) ⟹ ($\bigwedge\nu$ ω. ((ν, ω) ∈ *prog-sem I* (*PUrename x y* (*?* φ))) = ((*Radj x y* ν, *Radj x y* ω) ∈ *prog-sem I* (*?* φ)))
proof −
assume *hpsafe*:*hpsafe* (*?* φ)
fix ν ω
from *hpsafe* **have** *fsafe*:*fsafe* φ **by** *auto*
have *IH'*:$\bigwedge\nu$. (ν ∈ *fml-sem I* (*FUrename x y* φ)) = (*Radj x y* ν ∈ *fml-sem I* φ)
by (*rule IH*[*OF fsafe*])
have ((ν, ω) ∈ *prog-sem I* (*PUrename x y* (*?* φ))) = (ν = ω ∧ (ω ∈ *fml-sem I* (*FUrename x y* φ))) **by** (*cases* ω, *auto*)
moreover have ... = (ν = ω ∧ (*Radj x y* ω) ∈ *fml-sem I* φ) **using** *IH'* **by** *auto*
moreover have ... = (*Radj x y* ν = *Radj x y* ω ∧ (*Radj x y* ω) ∈ *fml-sem I* φ) **using** *Radj-eq-iff* **by** *auto*

moreover have ... = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*? φ*)) **by** (*cases Radj x y ω*, *auto*)
ultimately show *?thesis ν ω* **by** *auto*
qed
then show *?case* **by** *auto*
next
case (*FRadmit-Prop p args*) **then**
have *fsafe* (*Prop p args*) ⟹ (⋀*ν*. (*ν* ∈ *fml-sem I* (*FUrename x y* (*Prop p args*))) = ((*Radj x y ν*) ∈ *fml-sem I* (*Prop p args*)))
proof −
assume *fsafe*:*fsafe* (*Prop p args*)
fix *ν*
from *fsafe* **have** *dsafes*:⋀*i*. *dsafe* (*args i*) **using** *dfree-is-dsafe* **by** *auto*
have *IH*:⋀*i ν*. *dterm-sem I* (*TUrename x y* (*args i*)) *ν* = *dterm-sem I* (*args i*) (*Radj x y ν*)
using *TUren*[*OF good-interp dsafes*] **by** *auto*
have (*ν* ∈ *fml-sem I* (*FUrename x y* (*Prop p args*))) = (*ν* ∈ *fml-sem I* (*Prop p* (*λi* . *TUrename x y* (*args i*)))) **by** *auto*
moreover have ... = (*Predicates I p* (*χ i*. *dterm-sem I* (*TUrename x y* (*args i*)) *ν*)) **by** *auto*
moreover have ... = (*Predicates I p* (*χ i*. *dterm-sem I* (*args i*) (*Radj x y ν*))) **using** *IH* **by** *auto*
moreover have ... = ((*Radj x y ν*) ∈ *fml-sem I* (*Prop p args*)) **by** *auto*
ultimately show *?thesis ν* **by** *blast*
qed
then show *?case* **by** *auto*
next
case (*PRadmit-Sequence a b*) **then**
have *IH1*:*hpsafe a* ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y a*)) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I a*))
and *IH2*:*hpsafe b* ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y b*)) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I b*))
by *auto*
have *hpsafe* (*a* ;; *b*) ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*a* ;;*b*))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*a* ;; *b*)))
proof −
assume *hpsafe*:*hpsafe* (*a* ;; *b*)
fix *ν ω*
from *hpsafe* **have** *safe1*:*hpsafe a* **and** *safe2*:*hpsafe b* **by** *auto*
have *IH1*:(⋀*μ*. ((*ν*, *μ*) ∈ *prog-sem I* (*PUrename x y a*)) = ((*Radj x y ν*, *Radj x y μ*) ∈ *prog-sem I a*))
using *IH1*[*OF safe1*] **by** *auto*
have *IH2*:(⋀*μ*. ((*μ*, *ω*) ∈ *prog-sem I* (*PUrename x y b*)) = ((*Radj x y μ*, *Radj x y ω*) ∈ *prog-sem I b*))
using *IH2*[*OF safe2*] **by** *auto*
have ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*a* ;;*b*))) = ((*ν*, *ω*) ∈ *prog-sem I* ((*PUrename x y a*) ;;(*PUrename x y b*))) **by** *auto*
moreover have ... = (∃*μ*. (*ν*, *μ*) ∈ *prog-sem I* (*PUrename x y a*) ∧ (*μ*, *ω*) ∈ *prog-sem I* (*PUrename x y b*)) **by** *auto*

moreover have ... = ($\exists \mu$. (*Radj x y* ν, *Radj x y* μ) $\in$ *prog-sem I a* $\wedge$ (*Radj x y* μ, *Radj x y* ω) $\in$ *prog-sem I b*) **using** *IH1 IH2* **by** *auto*
moreover have ... = ($\exists \mu$. (*Radj x y* ν, μ) $\in$ *prog-sem I a* $\wedge$ (μ, *Radj x y* ω) $\in$ *prog-sem I b*)
apply *auto*
subgoal for *aa ba*
apply(*rule exI*[**where** *x*=*fst*(*Radj x y* (*aa*,*ba*))])
apply(*rule exI*[**where** *x*=*snd*(*Radj x y* (*aa*,*ba*))])
by *auto*
subgoal for *aa ba*
apply(*rule exI*[**where** *x*=*fst*(*Radj x y* (*aa*,*ba*))])
apply(*rule exI*[**where** *x*=*snd*(*Radj x y* (*aa*,*ba*))])
using *Radj-cancel* **by** *auto*
done
moreover have ... = ((*Radj x y* ν, *Radj x y* ω) $\in$ *prog-sem I* (*a* ;;*b*)) **by** (*auto*,*blast*)
ultimately show *?thesis* ν ω **by** *auto*
qed
then show *?case* **by** *auto*
next
case (*FRadmit-Diamond* α φ) **then**
have *IH1*:*hpsafe* $\alpha \Longrightarrow (\bigwedge \nu\ \omega.\ ((\nu, \omega) \in$ *prog-sem I* (*PUrename x y* α)) = ((*Radj x y* ν, *Radj x y* ω) $\in$ *prog-sem I* α))
and *IH2*:*fsafe* $\varphi \Longrightarrow (\bigwedge \nu.\ (\nu \in$ *fml-sem I* (*FUrename x y* φ)) = (*Radj x y* $\nu \in$ *fml-sem I* φ))
by *auto*
have *fsafe* ($\langle\alpha\rangle\varphi$) $\Longrightarrow (\bigwedge \nu.\ (\nu \in$ *fml-sem I* (*FUrename x y* ($\langle\alpha\rangle\varphi$))) = (*Radj x y* $\nu \in$ *fml-sem I* ($\langle\alpha\rangle\varphi$)))
proof −
assume *safe*:*fsafe* ($\langle\alpha\rangle\varphi$)
fix ν
from *safe* **have** *safe1*:*hpsafe* α **and** *safe2*:*fsafe* φ **by** *auto*
have *IH1*:$\bigwedge\omega$. ((ν, ω) $\in$ *prog-sem I* (*PUrename x y* α)) = ((*Radj x y* ν, *Radj x y* ω) $\in$ *prog-sem I* α)
using *IH1*[*OF safe1*] **by** *auto*
have *IH2*:$\bigwedge\nu$. ($\nu \in$ *fml-sem I* (*FUrename x y* φ)) = (*Radj x y* $\nu \in$ *fml-sem I* φ)
by (*rule IH2*[*OF safe2*])
have ($\nu \in$ *fml-sem I* (*FUrename x y* ($\langle\alpha\rangle\varphi$))) = ($\nu \in$ *fml-sem I* ($\langle$*PUrename x y* $\alpha\rangle$*FUrename x y* φ)) **by** *auto*
moreover have ... = ($\exists\ \omega.\ (\nu, \omega) \in$ *prog-sem I* (*PUrename x y* α) $\wedge\ \omega \in$ *fml-sem I* (*FUrename x y* φ)) **by** *auto*
moreover have ... = ($\exists\ \omega.$ (*Radj x y* ν, *Radj x y* ω) $\in$ *prog-sem I* $\alpha \wedge$ (*Radj x y* ω) $\in$ *fml-sem I* φ)
using *IH1 IH2* **by** *auto*
moreover have ... = ($\exists\ \omega.$ (*Radj x y* ν, ω) $\in$ *prog-sem I* $\alpha \wedge \omega \in$ *fml-sem I* φ)
apply *auto*
subgoal for *aa ba*

apply(*rule exI*[**where** *x*=*fst*(*Radj x y* (*aa*,*ba*))])
apply(*rule exI*[**where** *x*=*snd*(*Radj x y* (*aa*,*ba*))])
by *auto*
subgoal for *aa ba*
apply(*rule exI*[**where** *x*=*fst*(*Radj x y* (*aa*,*ba*))])
apply(*rule exI*[**where** *x*=*snd*(*Radj x y* (*aa*,*ba*))])
using *Radj-cancel* **by** *auto*
done
moreover have ... = (*Radj x y ν* ∈ *fml-sem I* (⟨*α*⟩*φ*)) **by** *auto*
ultimately show *?thesis ν* **by** *auto*
qed
then show *?case* **by** *auto*
next
case (*PRadmit-Loop a*) **then**
have *IH*: *hpsafe a* ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y a*)) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I a*))
by *auto*
have *hpsafe* (*a***) ⟹ (⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y* (*a***))) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I* (*a***)))
proof −
assume *safe*:*hpsafe* (*a***)
fix *ν ω*
from *safe* **have** *safe*:*hpsafe a* **by** *auto*
have *IH1*:(⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y a*)) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I a*))
by (*rule IH*[*OF safe*])
have *relpow-iff*:⋀*ν ω R n*. ((*ν*, *ω*) ∈ *R* ^^ *Suc n*) = (∃ *μ*. (*ν*, *μ*) ∈ *R* ∧ (*μ*, *ω*) ∈ *R* ^^ *n*)
apply *auto*
subgoal for *R n x y z* **by** (*auto simp add*: *relpow-Suc-D2'*)
subgoal for *ν ω R n μ* **using** *relpow-Suc-I2* **by** *fastforce*
done
have *rtrancl-iff-relpow*:⋀*ν ω R*. ((*ν*, *ω*) ∈ *R**) = (∃ *n*. (*ν*, *ω*) ∈ *R* ^^ *n*)
using *rtrancl-imp-relpow relpow-imp-rtrancl* **by** *blast*
have *lem*:⋀*n*. (∀ *ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y a*)^^*n*) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I a*^^*n*))
subgoal for *n*
proof(*induction n*)
case *0*
then show *?case* **using** *Radj-eq-iff* **by** *auto*
next
case (*Suc n*) **then**
have *IH2*:⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y a*) ^^ *n*) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I a* ^^ *n*)
by *auto*
have ⋀*ν ω*. ((*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y a*) ^^ *Suc n*) = ((*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I a* ^^ *Suc n*)
proof −
fix *ν ω*

```
        have ((ν, ω) ∈ prog-sem I (PUrename x y a) ^^ Suc n)
          = (∃ μ. (ν, μ) ∈ prog-sem I (PUrename x y a) ∧ (μ, ω) ∈ prog-sem I
(PUrename x y a) ^^ n)
          using relpow-iff[of ν ω n prog-sem I (PUrename x y a)] by auto
         moreover have ... = (∃ μ. (Radj x y ν, Radj x y μ) ∈ prog-sem I a ∧
(Radj x y μ, Radj x y ω) ∈ prog-sem I a ^^ n)
          using IH1 IH2 by blast
        moreover have ... = (∃ μ. (Radj x y ν, μ) ∈ prog-sem I a ∧ (μ, Radj x y
ω) ∈ prog-sem I a ^^ n)
          apply auto
           subgoal for aa ba
             apply(rule exI[where x=fst(Radj x y (aa,ba))])
             apply(rule exI[where x=snd(Radj x y (aa,ba))])
             by auto
          subgoal for aa ba
            apply(rule exI[where x=fst(Radj x y (aa,ba))])
            apply(rule exI[where x=snd(Radj x y (aa,ba))])
            using Radj-cancel by auto
          done
        moreover have ... = ((Radj x y ν, Radj x y ω) ∈ prog-sem I a ^^ Suc n)
          using relpow-iff[of Radj x y ν Radj x y ω  n prog-sem I a] by auto
         ultimately show ?thesis ν ω by auto
       qed
       then show ?case by auto
     qed
     done
   have ((ν, ω) ∈ prog-sem I (PUrename x y (a** ))) = ((ν, ω) ∈ (prog-sem I
(PUrename x y a))*) by auto
   moreover have ... = (∃ n. (ν, ω) ∈ (prog-sem I (PUrename x y a)) ^^ n)
     using rtrancl-iff-relpow[of ν ω prog-sem I (PUrename x y a)] by auto
   moreover have ... = (∃ n. (Radj x y ν, Radj x y ω) ∈ (prog-sem I a) ^^ n)
     using lem by blast
   moreover have ... = ((Radj x y ν, Radj x y ω) ∈ (prog-sem I a)*)
     using rtrancl-iff-relpow[of Radj x y ν Radj x y ω prog-sem I a] by auto
   moreover have ... = ((Radj x y ν, Radj x y ω) ∈ prog-sem I (a** )) by auto
   ultimately show ?thesis ν ω by blast
 qed
 then show ?case by auto
next
 case (PRadmit-EvolveODE ODE φ) then
 have ORA:ORadmit ODE
   and IH:fsafe φ ⟹ (⋀ν. (ν ∈ fml-sem I (FUrename x y φ)) = (Radj x y ν ∈
fml-sem I φ))
   by auto
 have hpsafe (EvolveODE ODE φ) ⟹ (⋀ν ω. ((ν, ω) ∈ prog-sem I (PUrename
x y (EvolveODE ODE φ))) = ((Radj x y ν, Radj x y ω) ∈ prog-sem I (EvolveODE
ODE φ)))
 proof −
   assume safe:hpsafe (EvolveODE ODE φ)
```

fix ν ω
from *safe* **have** *osafe*:*osafe ODE* **and** *fsafe*:*fsafe* φ **by** *auto*
have *IH1*:$\bigwedge\nu$. ($\nu \in$ *fml-sem I* (*FUrename x y* φ) = (*Radj x y* $\nu \in$ *fml-sem I* φ)) **by** (*rule IH*[*OF fsafe*])
have *IH2*:$\bigwedge\nu$. *ODE-sem I* (*OUrename x y ODE*) ν = *RSadj x y* (*ODE-sem I ODE* (*RSadj x y* ν))
using *OUren*[*OF ORA*] **by** *auto*
have *RSadj-Radj*:$\bigwedge a\ b$. (*RSadj x y a*, *RSadj x y b*) = *Radj x y* (*a*,*b*)
unfolding *RSadj-def Radj-def* **by** *auto*
have *Radj-swap*:$\bigwedge a\ b$. *Radj x y a* = *b* $\Longrightarrow$ *a* = *Radj x y b*
using *Radj-cancel Radj-eq-iff* **by** *metis*
have *mkv*:$\bigwedge t$ *sol b*. *Radj x y* (*mk-v I* (*OUrename x y ODE*) (*sol 0*, *b*) (*sol t*)) = *mk-v I ODE* (*RSadj x y* (*sol 0*), *RSadj x y b*) (*RSadj x y* (*sol t*))
using *mkv-lemma*[*OF ORA*] **by** *blast*
have *mkv2*:$\bigwedge t$ *sol b*. *Radj x y* ω = *mk-v I ODE* (*sol 0*, *b*) (*sol t*) $\Longrightarrow$
ω = *mk-v I* (*OUrename x y ODE*) (*RSadj x y* (*sol 0*), *RSadj x y b*) (*RSadj x y* (*sol t*))
using *mkv-lemma*[*OF ORA*] **by** (*metis RSadj-cancel Radj-cancel*)
have *sol*:$\bigwedge t$ *sol b*. 0 $\leq$ *t* $\Longrightarrow$
(*sol solves-ode* (λa. *ODE-sem I* (*OUrename x y ODE*))) {*0..t*} {*xa*. *mk-v I* (*OUrename x y ODE*) (*sol 0*, *b*) *xa* $\in$ *fml-sem I* (*FUrename x y* φ)} $\Longrightarrow$
((*λt*. *RSadj x y* (*sol t*)) *solves-ode* (λa. *ODE-sem I ODE*)) {*0..t*} {*xa*. *mk-v I ODE* (*RSadj x y* (*sol 0*), *RSadj x y b*) *xa* $\in$ *fml-sem I* φ}
using *sol-lemma IH1 IH2 ORA* **by** *blast*
have *sol2*:$\bigwedge t$ *sol b*. 0 $\leq$ *t* $\Longrightarrow$
(*sol solves-ode* (λa. *ODE-sem I ODE*)) {*0..t*} {*x*. *mk-v I ODE* (*sol 0*, *b*) *x* $\in$ *fml-sem I* φ} $\Longrightarrow$
((*λt*. *RSadj x y* (*sol t*)) *solves-ode* (λa. *ODE-sem I* (*OUrename x y ODE*))) {*0..t*} {*xa*. *mk-v I* (*OUrename x y ODE*) (*RSadj x y* (*sol 0*), *RSadj x y b*) *xa* $\in$ *fml-sem I* (*FUrename x y* φ)}
using *sol-lemma2 IH1 IH2 ORA* **by** *blast*
show *?thesis* ν ω
apply *auto*
subgoal for *b sol t*
apply(*rule exI*[**where** *x*= *RSadj x y b*])
apply(*rule exI*[**where** *x*= (λt. *RSadj x y* (*sol t*))])
apply(*rule conjI*)
subgoal using *RSadj-Radj*[*of sol 0 b*] **by** *auto*
apply(*rule exI*[**where** *x* =*t*])
apply(*rule conjI*)
subgoal by (*rule mkv*)
apply(*rule conjI*)
subgoal by *assumption*
by (*rule sol*)
subgoal for *b sol t*
apply(*rule exI*[**where** *x*= *RSadj x y b*])
apply(*rule exI*[**where** *x*= (λt. *RSadj x y* (*sol t*))])
apply(*rule conjI*)
subgoal using *RSadj-Radj*[*of sol 0 b*] *Radj-swap*[*of* ν (*sol 0*,*b*)] **by** *auto*

apply(*rule exI*[**where** $x = t$])
apply(*rule conjI*)
subgoal by (*rule mkv2*)
apply(*rule conjI*)
subgoal by *assumption*
by (*rule sol2*)
done
qed
then show *?case* **by** *auto*
qed (*auto simp add*: *Radj-def*)

lemma *FUren*:*is-interp I* $\Longrightarrow$ *FRadmit* $\varphi \Longrightarrow$ *fsafe* $\varphi \Longrightarrow (\bigwedge \nu.\ (\nu \in \textit{fml-sem}\ I\ (\textit{FUrename}\ x\ y\ \varphi)) = (\textit{Radj}\ x\ y\ \nu \in \textit{fml-sem}\ I\ \varphi))$
using *PUren-FUren* **by** *blast*

12.4 Uniform Renaming Rule Soundness

lemma *URename-sound*:*FRadmit* $\varphi \Longrightarrow$ *fsafe* $\varphi \Longrightarrow$ *valid* $\varphi \Longrightarrow$ *valid* (*FUrename* $x\ y\ \varphi$)
unfolding *valid-def* **using** *FUren* **by** *blast*

12.5 Bound Renaming Rule Soundness

lemma *BRename-sound*:
assumes *FRA*:*FRadmit*($[[\textit{Assign}\ x\ \vartheta]]\varphi$)
assumes *fsafe*:*fsafe* ($[[\textit{Assign}\ x\ \vartheta]]\varphi$)
assumes *valid*:*valid* ($[[\textit{Assign}\ x\ \vartheta]]\varphi$)
assumes *FVF*:$\{\textit{Inl}\ y, \textit{Inr}\ y, \textit{Inr}\ x\} \cap \textit{FVF}\ \varphi = \{\}$
shows *valid*($[[\textit{Assign}\ y\ \vartheta]]\textit{FUrename}\ x\ y\ \varphi$)
proof −
have *FRA'*:*FRadmit* φ **using** *FRA*
by (*metis* (*no-types*, *lifting*) *Box-def FRadmit.cases formula.distinct*(*15*) *formula.distinct*(*21*) *formula.distinct*(*27*) *formula.distinct*(*29*) *formula.distinct*(*3*) *formula.distinct*(*31*) *formula.distinct formula.distinct*(*9*) *formula.inject*(*3*) *formula.inject*(*6*))
have *fsafe'*:*fsafe* φ **using** *fsafe* **by** (*simp add*: *Box-def*)
have *dsafe*:*dsafe* ϑ **using** *fsafe* **by** (*simp add*: *Box-def*)
have $\bigwedge I\ \nu.$ *is-interp I* $\Longrightarrow \nu \in$ *fml-sem I* ($[[y := \vartheta]]\textit{FUrename}\ x\ y\ \varphi$)
proof −
fix I::$('sf,'sc,'sz)$ *interp* **and** ν::$'sz$ *state*
assume *good-interp*:*is-interp I*
from *FVF* **have** *sub*:*FVF* $\varphi \subseteq -\{\textit{Inl}\ y, \textit{Inr}\ y, \textit{Inr}\ x\}$ **by** *auto*
have *Vagree* (*repv* (*Radj x y* ν) *x* (*dterm-sem I* ϑ ν)) (*repv* ν *x* (*dterm-sem I* ϑ ν)) ($-\{\textit{Inl}\ y, \textit{Inr}\ y, \textit{Inr}\ x\}$)
unfolding *Vagree-def* **using** *FVF* **unfolding** *Radj-def RSadj-def* **by** *auto*
then have *agree*:*Vagree* (*repv* (*Radj x y* ν) *x* (*dterm-sem I* ϑ ν)) (*repv* ν *x* (*dterm-sem I* ϑ ν)) (*FVF* φ)
using *agree-sub*[*OF sub*] **by** *auto*
have *fml-sem-eq*:(*repv* (*Radj x y* ν) *x* (*dterm-sem I* ϑ ν) $\in$ *fml-sem I* φ) = (*repv* ν *x* (*dterm-sem I* ϑ ν) $\in$ *fml-sem I* φ)
using *coincidence-formula*[*OF fsafe' Iagree-refl agree*] **by** *auto*

have ($\nu \in$ *fml-sem I* ([[$y := \vartheta$]]*FUrename x y* φ)) = (*repv* ν *y* (*dterm-sem I* ϑ ν) $\in$ *fml-sem I* (*FUrename x y* φ))
by *auto*
moreover have ... = (*Radj x y* (*repv* ν *y* (*dterm-sem I* ϑ ν)) $\in$ *fml-sem I* φ)
using *FUren*[*OF good-interp FRA' fsafe'*] **by** *auto*
moreover have ... = (*repv* (*Radj x y* ν) *x* (*dterm-sem I* ϑ ν) $\in$ *fml-sem I* φ)
using *Radj-repv1* **by** *auto*
moreover have ... = ($\nu \in$ *fml-sem I* ([[$x := \vartheta$]]φ))
using *fml-sem-eq* **by** *auto*
moreover have ... = *True*
using *valid* **unfolding** *valid-def* **using** *good-interp* **by** *blast*
ultimately
show $\nu \in$ *fml-sem I* ([[$y := \vartheta$]]*FUrename x y* φ)
by *blast*
qed
then
show *?thesis* **unfolding** *valid-def* **by** *auto*
qed

end end
theory *Pretty-Printer*
imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
begin
context *ids* **begin**

13 Syntax Pretty-Printer

The deeply-embedded syntax is difficult to read for large formulas. This pretty-printer produces a more human-friendly syntax, which can be helpful if you want to produce a proof term by hand for the proof checker (not recommended for most users).

fun *join* :: *string* ⇒ *char list list* ⇒ *char list*
where *join S* [] = []
| *join S* [*S'*] = *S'*
| *join S* (*S'* # *SS*) = *S'* @ *S* @ (*join S SS*)

fun *vid-to-string*::'*sz* ⇒ *char list*
where *vid-to-string vid* = (*if vid = vid1 then* ''*x*'' *else if vid = vid2 then* ''*y*'' *else if vid = vid3 then* ''*z*'' *else* ''*w*'')

fun *oid-to-string*::'*sz* ⇒ *char list*

where *oid-to-string vid* = (*if vid* = *vid1 then* ''*c*'' *else if vid* = *vid2 then* ''*c2*'' *else if vid* = *vid3 then* ''*c3*'' *else* ''*c4*'')

fun *cid-to-string*::'*sc* ⇒ *char list*
where *cid-to-string vid* = (*if vid* = *pid1 then* ''*C*'' *else if vid* = *pid2 then* ''*C2*'' *else if vid* = *pid3 then* ''*C3*'' *else* ''*C4*'')

fun *ppid-to-string*::'*sc* ⇒ *char list*
where *ppid-to-string vid* = (*if vid* = *pid1 then* ''*P*'' *else if vid* = *pid2 then* ''*Q*'' *else if vid* = *pid3 then* ''*R*'' *else* ''*H*'')

fun *hpid-to-string*::'*sz* ⇒ *char list*
where *hpid-to-string vid* = (*if vid* = *vid1 then* ''*a*'' *else if vid* = *vid2 then* ''*b*'' *else if vid* = *vid3 then* ''*a1*'' *else* ''*b1*'')

fun *fid-to-string*::'*sf* ⇒ *char list*
where *fid-to-string vid* = (*if vid* = *fid1 then* ''*f*'' *else if vid* = *fid2 then* ''*g*'' *else if vid* = *fid3 then* ''*h*'' *else* ''*j*'')

primrec *trm-to-string*::('*sf*,'*sz*) *trm* ⇒ *char list*
where
trm-to-string (*Var x*) = *vid-to-string x*
| *trm-to-string* (*Const r*) = ''*r*''
| *trm-to-string* (*Function f args*) = *fid-to-string f*
| *trm-to-string* (*Plus t1 t2*) = *trm-to-string t1* @ ''+'' @ *trm-to-string t2*
| *trm-to-string* (*Times t1 t2*) = *trm-to-string t1* @ ''∗'' @ *trm-to-string t2*
| *trm-to-string* (*DiffVar x*) = ''*Dv*{'' @ *vid-to-string x* @ ''}''
| *trm-to-string* (*Differential t*) = ''*D*{'' @ *trm-to-string t* @ ''}''

primrec *ode-to-string*::('*sf*,'*sz*) *ODE* ⇒ *char list*
where
ode-to-string (*OVar x*) = *oid-to-string x*
| *ode-to-string* (*OSing x t*) = ''*d*'' @ *vid-to-string x* @ ''='' @ *trm-to-string t*
| *ode-to-string* (*OProd ODE1 ODE2*) = *ode-to-string ODE1* @ '', '' @ *ode-to-string ODE2*

fun *fml-to-string* ::('*sf*, '*sc*, '*sz*) *formula* ⇒ *char list*
and *hp-to-string* ::('*sf*, '*sc*, '*sz*) *hp* ⇒ *char list*
where
fml-to-string (*Geq t1 t2*) = *trm-to-string t1* @ ''>='' @ *trm-to-string t2*
| *fml-to-string* (*Prop p args*) = []
| *fml-to-string* (*Not p*) =
(*case p of* (*And* (*Not q*) (*Not* (*Not p*))) ⇒ *fml-to-string p* @ ''−>'' @ *fml-to-string q*
| (*Exists x* (*Not p*)) ⇒ ''*A*''@ *vid-to-string x* @ ''.'' @ *fml-to-string p*
| (*Diamond a* (*Not p*)) ⇒ ''[''@ *hp-to-string a* @ '']'' @ *fml-to-string p*
| (*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p*′) (*Not q*′)))) ⇒
(*if* (*p* = *p*′ ∧ *q* = *q*′) *then fml-to-string p* @ ''<−>'' @ *fml-to-string q else* ''!'' @ *fml-to-string* (*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p*′) (*Not q*′)))))

```
            | - ⇒ ''!'' @ fml-to-string p)
  | fml-to-string (And p q) = fml-to-string p @ ''&'' @ fml-to-string q
  | fml-to-string (Exists x p) = ''E'' @ vid-to-string x @ '' . '' @ fml-to-string p
  | fml-to-string (Diamond a p) = ''<'' @ hp-to-string a @ ''>'' @ fml-to-string p
  | fml-to-string (InContext C p) =
      (case p of
        (Geq  - -) ⇒ ppid-to-string C
      | - ⇒ cid-to-string C @ ''('' @ fml-to-string p @ '')'')

  | hp-to-string (Pvar a) = hpid-to-string a
  | hp-to-string (Assign x e) = vid-to-string x @ '':='' @ trm-to-string e
  | hp-to-string (DiffAssign x e) = ''D{'' @ vid-to-string x @ ''}:='' @ trm-to-string
e
  | hp-to-string (Test p) = ''?'' @ fml-to-string p
   | hp-to-string (EvolveODE ODE p) = ''{'' @ ode-to-string ODE @ ''&'' @
fml-to-string p @ ''}''
  | hp-to-string (Choice a b) = hp-to-string a @ ''U'' @ hp-to-string b
  | hp-to-string (Sequence a b) = hp-to-string a @ '';'' @ hp-to-string b
  | hp-to-string (Loop a) = hp-to-string a @ ''*''

end end
theory Proof-Checker
imports
  Ordinary-Differential-Equations.ODE-Analysis
  Ids
  Lib
  Syntax
  Denotational-Semantics
  Axioms
  Differential-Axioms
  Frechet-Correctness
  Static-Semantics
  Coincidence
  Bound-Effect
  Uniform-Renaming
  USubst-Lemma
  Pretty-Printer

begin context ids begin
```

14 Proof Checker

This proof checker defines a datatype for proof terms in dL and a function for checking proof terms, with a soundness proof that any proof accepted by the checker is a proof of a sound rule or valid formula.

A simple concrete hybrid system and a differential invariant rule for conjunctions are provided as example proofs.

lemma *sound-weaken-gen*:$\bigwedge A\ B\ C.\ sublist\ A\ B \Longrightarrow sound\ (A,\ C) \Longrightarrow sound\ (B,C)$
proof (*rule soundI-mem*)
fix *A B*::(*'sf*,*'sc*,*'sz*) *sequent list*
and *C*::(*'sf*,*'sc*,*'sz*) *sequent*
and *I*::(*'sf*,*'sc*,*'sz*) *interp*
assume *sub*:*sublist A B*
assume *good*:*is-interp I*
assume *sound* (*A*, *C*)
then have *soundC*:$(\bigwedge\varphi.\ List.member\ A\ \varphi \Longrightarrow seq\text{-}sem\ I\ \varphi = UNIV) \Longrightarrow seq\text{-}sem\ I\ C = UNIV$
apply *simp*
apply(*drule soundD-mem*)
by (*auto simp add*: *good*)
assume *SG*:$(\bigwedge\varphi.\ List.member\ B\ \varphi \Longrightarrow seq\text{-}sem\ I\ \varphi = UNIV)$
show *seq-sem I C* = *UNIV*
using *soundC SG sub* **unfolding** *sublist-def* **by** *auto*
qed

lemma *sound-weaken*:$\bigwedge SG\ SGS\ C.\ sound\ (SGS,\ C) \Longrightarrow sound\ (SG\ \#\ SGS,\ C)$
subgoal for *SG SGS C*
apply(*induction SGS*)
subgoal unfolding *sound-def* **by** *auto*
subgoal for *SG2 SGS*
unfolding *sound-def*
by (*metis fst-conv le0 length-Cons not-less-eq nth-Cons-Suc snd-conv*)
done
done

lemma *member-filter*:$\bigwedge P.\ List.member\ (filter\ P\ L)\ x \Longrightarrow List.member\ L\ x$
apply(*induction L*, *auto*)
by(*metis* (*full-types*) *member-rec*(*1*))

lemma *nth-member*:$n < List.length\ L \Longrightarrow List.member\ L\ (List.nth\ L\ n)$
apply(*induction L*, *auto simp add*: *member-rec*)
by (*metis in-set-member length-Cons nth-mem set-ConsD*)

lemma *mem-appL*:*List.member A x* $\Longrightarrow$ *List.member* (*A* @ *B*) *x*
by(*induction A*, *auto simp add*: *member-rec*)

lemma *sound-weaken-appR*:$\bigwedge SG\ SGS\ C.\ sound\ (SG,\ C) \Longrightarrow sound\ (SG\ @\ SGS,\ C)$
subgoal for *SG SGS C*
apply(*rule sound-weaken-gen*)
apply(*auto*)
unfolding *sublist-def* **apply**(*rule allI*)
subgoal for *x*
using *mem-appL*[*of SG x SGS*] **by** *auto*
done

done

fun *start-proof*::('*sf*,'*sc*,'*sz*) *sequent* ⇒ ('*sf*,'*sc*,'*sz*) *rule*
where *start-proof S* = ([*S*], *S*)

lemma *start-proof-sound*:*sound* (*start-proof S*)
unfolding *sound-def* **by** *auto*

15 Proof Checker Implementation

datatype *axiom* =
AloopIter | *AI* | *Atest* | *Abox* | *Achoice* | *AK* | *AV* | *Aassign* | *Adassign*
| *AdConst* | *AdPlus* | *AdMult*
| *ADW* | *ADE* | *ADC* | *ADS* | *ADIGeq* | *ADIGr* | *ADG*

fun *get-axiom*:: *axiom* ⇒ ('*sf*,'*sc*,'*sz*) *formula*
where
get-axiom AloopIter = *loop-iterate-axiom*
| *get-axiom AI* = *Iaxiom*
| *get-axiom Atest* = *test-axiom*
| *get-axiom Abox* = *box-axiom*
| *get-axiom Achoice* = *choice-axiom*
| *get-axiom AK* = *Kaxiom*
| *get-axiom AV* = *Vaxiom*
| *get-axiom Aassign* = *assign-axiom*
| *get-axiom Adassign* = *diff-assign-axiom*
| *get-axiom AdConst* = *diff-const-axiom*
| *get-axiom AdPlus* = *diff-plus-axiom*
| *get-axiom AdMult* = *diff-times-axiom*
| *get-axiom ADW* = *DWaxiom*
| *get-axiom ADE* = *DEaxiom*
| *get-axiom ADC* = *DCaxiom*
| *get-axiom ADS* = *DSaxiom*
| *get-axiom ADIGeq* = *DIGeqaxiom*
| *get-axiom ADIGr* = *DIGraxiom*
| *get-axiom ADG* = *DGaxiom*

lemma *axiom-safe*:*fsafe* (*get-axiom a*)
by(*cases a*, *auto simp add*: *axiom-defs Box-def Or-def Equiv-def Implies-def empty-def Equals-def f1-def p1-def P-def f0-def expand-singleton Forall-def Greater-def id-simps*)

lemma *axiom-valid*:*valid* (*get-axiom a*)
proof (*cases a*)
case *AloopIter*
then show *?thesis* **by** (*simp add*: *loop-valid*)
next
case *AI*

```
  then show ?thesis by (simp add: I-valid)
next
  case Atest
  then show ?thesis by (simp add: test-valid)
next
  case Abox
  then show ?thesis by (simp add: box-valid)
next
  case Achoice
  then show ?thesis by (simp add: choice-valid)
next
  case AK
  then show ?thesis by (simp add: K-valid)
next
  case AV
  then show ?thesis by (simp add: V-valid)
next
  case Aassign
  then show ?thesis by (simp add: assign-valid)
next
  case Adassign
  then show ?thesis by (simp add: diff-assign-valid)
next
  case AdConst
  then show ?thesis by (simp add: diff-const-axiom-valid)
next
  case AdPlus
  then show ?thesis by (simp add: diff-plus-axiom-valid)
next
  case AdMult
  then show ?thesis by (simp add: diff-times-axiom-valid)
next
  case ADW
  then show ?thesis by (simp add: DW-valid)
next
  case ADE
  then show ?thesis by (simp add: DE-valid)
next
  case ADC
  then show ?thesis by (simp add: DC-valid)
next
  case ADS
  then show ?thesis by (simp add: DS-valid)
next
  case ADIGeq
  then show ?thesis by (simp add: DIGeq-valid)
next
  case ADIGr
  then show ?thesis by (simp add: DIGr-valid)
```

next
  case *ADG*
  then show *?thesis* **by** (*simp add*: *DG-valid*)
qed

datatype *rrule* = *ImplyR* | *AndR* | *CohideR* | *CohideRR* | *TrueR* | *EquivR*
datatype *lrule* = *ImplyL* | *AndL* | *EquivForwardL* | *EquivBackwardL*

datatype ('*a*, '*b*, '*c*) *step* =
  Axiom axiom
| *MP*
| *G*
| *CT*
| *CQ* ('*a*, '*c*) *trm* ('*a*, '*c*) *trm* ('*a*, '*b*, '*c*) *subst*
| *CE* ('*a*, '*b*, '*c*) *formula* ('*a*, '*b*, '*c*) *formula* ('*a*, '*b*, '*c*) *subst*
| *Skolem*
— Apply *Usubst* to some other (valid) formula
| *VSubst* ('*a*, '*b*, '*c*) *formula* ('*a*, '*b*, '*c*) *subst*
| *AxSubst axiom* ('*a*, '*b*, '*c*) *subst*
| *URename*
| *BRename*
| *Rrule rrule nat*
| *Lrule lrule nat*
| *CloseId nat nat*
| *Cut* ('*a*, '*b*, '*c*) *formula*
| *DEAxiomSchema* ('*a*,'*c*) *ODE* ('*a*, '*b*, '*c*) *subst*

type-synonym ('*a*, '*b*, '*c*) *derivation* = (*nat* ∗ ('*a*, '*b*, '*c*) *step*) *list*
type-synonym ('*a*, '*b*, '*c*) *pf* = ('*a*,'*b*,'*c*) *sequent* ∗ ('*a*, '*b*, '*c*) *derivation*

fun *seq-to-string* :: ('*sf*, '*sc*, '*sz*) *sequent* ⇒ *char list*
where *seq-to-string* (*A*,*S*) = *join* '', '' (*map fml-to-string A*) @ '' |− '' @ *join* '', '' (*map fml-to-string S*)

fun *rule-to-string* :: ('*sf*, '*sc*, '*sz*) *rule* ⇒ *char list*
where *rule-to-string* (*SG*, *C*) = (*join* '';; '' (*map seq-to-string SG*)) @ '' '' @ *seq-to-string C*

fun *close* :: '*a list* ⇒ '*a* ⇒'*a list*
where *close L x* = *filter* (λ*y*. *y* ≠ *x*) *L*

fun *closeI* ::'*a list* ⇒ *nat* ⇒'*a list*
where *closeI L i* = *close L* (*nth L i*)

lemma *close-sub*:*sublist* (*close* Γ φ) Γ
  apply (*auto simp add*: *sublist-def*)
  using *member-filter* **by** *fastforce*

lemma *close-app-comm*:*close* (*A* @ *B*) *x* = *close A x* @ *close B x*

by *auto*

lemma *close-provable-sound*:*sound* (*SG*, *C*) ⟹ *sound* (*close SG* φ, φ) ⟹ *sound* (*close SG* φ, *C*)
proof (*rule soundI-mem*)
fix *I*::('*sf*,'*sc*,'*sz*) *interp*
assume *S1*:*sound* (*SG*, *C*)
assume *S2*:*sound* (*close SG* φ, φ)
assume *good*:*is-interp I*
assume *SGCs*:(⋀φ'. *List.member* (*close SG* φ) φ' ⟹ *seq-sem I* φ' = *UNIV*)
have *Sφ*:*seq-sem I* φ = *UNIV*
using *S2* **apply** *simp*
apply(*drule soundD-mem*)
using *good* **apply** *auto*
using *SGCs UNIV-I* **by** *fastforce*
have *mem-close*:⋀*P*. *List.member SG P* ⟹ *P* ≠ φ ⟹ *List.member* (*close SG* φ) *P*
by(*induction SG*, *auto simp add*: *member-rec*)
have *SGs*:⋀*P*. *List.member SG P* ⟹ *seq-sem I P* = *UNIV*
subgoal for *P*
apply(*cases P* = φ)
subgoal using *Sφ* **by** *auto*
subgoal using *mem-close*[*of P*] *SGCs* **by** *auto*
done
done
show *seq-sem I C* = *UNIV*
using *S1* **apply** *simp*
apply(*drule soundD-mem*)
using *good* **apply** *auto*
using *SGs* **apply** *auto*
using *impl-sem* **by** *blast*
qed

fun *Lrule-result* :: *lrule* ⇒ *nat* ⇒ ('*sf*, '*sc*, '*sz*) *sequent* ⇒ ('*sf*, '*sc*, '*sz*) *sequent list*
where *Lrule-result AndL j* (*A*,*S*) = (*case* (*nth A j*) *of And p q* ⇒ [(*close* ([*p*, *q*] @ *A*) (*nth A j*), *S*)])
| *Lrule-result ImplyL j* (*A*,*S*) = (*case* (*nth A j*) *of Not* (*And* (*Not q*) (*Not* (*Not p*))) ⇒
[(*close* (*q* # *A*) (*nth A j*), *S*), (*close A* (*nth A j*), *p* # *S*)])
| *Lrule-result EquivForwardL j* (*A*,*S*) = (*case* (*nth A j*) *of Not*(*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p'*) (*Not q'*)))) ⇒
[(*close* (*q* # *A*) (*nth A j*), *S*), (*close A* (*nth A j*), *p* # *S*)])
| *Lrule-result EquivBackwardL j* (*A*,*S*) = (*case* (*nth A j*) *of Not*(*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p'*) (*Not q'*)))) ⇒
[(*close* (*p* # *A*) (*nth A j*), *S*), (*close A* (*nth A j*), *q* # *S*)])

— Note: Some of the pattern-matching here is... interesting. The reason for this is that we can only

— match on things in the base grammar, when we would quite like to check things in the derived grammar.
— So all the pattern-matches have the definitions expanded, sometimes in a silly way.

```
fun Rrule-result :: rrule ⇒ nat ⇒ ('sf, 'sc, 'sz) sequent ⇒ ('sf, 'sc, 'sz) sequent list
where
  Rstep-Imply:Rrule-result ImplyR j (A,S) = (case (nth S j) of Not (And (Not q) (Not (Not p))) ⇒ [(p # A, q # (closeI S j))] | - ⇒ undefined)
| Rstep-And:Rrule-result AndR j (A,S) = (case (nth S j) of (And p q) ⇒ [(A, p # (closeI S j )), (A, q # (closeI S j))])
| Rstep-EquivR:Rrule-result EquivR j (A,S) =
   (case (nth S j) of Not(And (Not (And p q)) (Not (And (Not p') (Not q')))) ⇒
                (if (p = p' ∧ q = q') then [(p # A, q # (closeI S j)), (q # A, p # (closeI S j))]
                else undefined))
| Rstep-CohideR:Rrule-result CohideR j (A,S) = [(A, [nth S j])]
| Rstep-CohideRR:Rrule-result CohideRR j (A,S) = [([], [nth S j])]
| Rstep-TrueR:Rrule-result TrueR j (A,S) = []

fun step-result :: ('sf, 'sc, 'sz) rule ⇒ (nat * ('sf, 'sc, 'sz) step) ⇒  ('sf, 'sc, 'sz) rule
where
  Step-axiom:step-result (SG,C) (i,Axiom a)   = (closeI SG i, C)
| Step-AxSubst:step-result (SG,C) (i,AxSubst a σ)   = (closeI SG i, C)
| Step-Lrule:step-result (SG,C) (i,Lrule L j) = (close (append SG (Lrule-result L j (nth SG i))) (nth SG i), C)
| Step-Rrule:step-result (SG,C) (i,Rrule L j) = (close (append SG (Rrule-result L j (nth SG i))) (nth SG i), C)
| Step-Cut:step-result (SG,C) (i,Cut φ) = (let (A,S) = nth SG i in ((φ # A, S) # ((A, φ # S) # (closeI SG i)), C))
| Step-Vsubst:step-result (SG,C) (i,VSubst φ σ) = (closeI SG i, C)
| Step-CloseId:step-result (SG,C) (i,CloseId j k) = (closeI SG i, C)
| Step-G:step-result (SG,C) (i,G) = (case nth SG i of (-, (Not (Diamond q (Not p))) # Nil) ⇒ (([], [p]) # closeI SG i, C))
| Step-DEAxiomSchema:step-result (SG,C) (i,DEAxiomSchema ODE σ) = (closeI SG i, C)
| Step-CE:step-result (SG,C) (i, CE φ ψ σ) =  (closeI SG i, C)
| Step-CQ:step-result (SG,C) (i, CQ ϑ1 ϑ2 σ) =  (closeI SG i, C)
| Step-default:step-result R (i,S) = R

fun deriv-result :: ('sf, 'sc, 'sz) rule ⇒ ('sf, 'sc, 'sz) derivation ⇒ ('sf, 'sc, 'sz) rule
where
  deriv-result R [] = R
| deriv-result R (s # ss) = deriv-result (step-result R s) (ss)

fun proof-result :: ('sf, 'sc, 'sz) pf ⇒ ('sf, 'sc, 'sz) rule
where proof-result (D,S) = deriv-result (start-proof D) S
```

inductive *lrule-ok* ::('*sf*,'*sc*,'*sz*) *sequent list* ⇒ ('*sf*,'*sc*,'*sz*) *sequent* ⇒ *nat* ⇒ *nat* ⇒ *lrule* ⇒ *bool*
where
Lrule-And:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* && *q*) ⟹ *lrule-ok SG C i j AndL*
| *Lrule-Imply*:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* → *q*) ⟹ *lrule-ok SG C i j ImplyL*
| *Lrule-EquivForward*:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* ↔ *q*) ⟹ *lrule-ok SG C i j EquivForwardL*
| *Lrule-EquivBackward*:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* ↔ *q*) ⟹ *lrule-ok SG C i j EquivBackwardL*

named-theorems *prover Simplification rules for checking validity of proof certificates*
lemmas [*prover*] = *axiom-defs Box-def Or-def Implies-def filter-append ssafe-def SDom-def FUadmit-def PFUadmit-def id-simps*

inductive-simps
Lrule-And[*prover*]: *lrule-ok SG C i j AndL*
and *Lrule-Imply*[*prover*]: *lrule-ok SG C i j ImplyL*
and *Lrule-Forward*[*prover*]: *lrule-ok SG C i j EquivForwardL*
and *Lrule-EquivBackward*[*prover*]: *lrule-ok SG C i j EquivBackwardL*

inductive *rrule-ok* ::('*sf*,'*sc*,'*sz*) *sequent list* ⇒ ('*sf*,'*sc*,'*sz*) *sequent* ⇒ *nat* ⇒ *nat* ⇒ *rrule* ⇒ *bool*
where
Rrule-And:⋀*p q*. *nth* (*snd* (*nth SG i*)) *j* = (*p* && *q*) ⟹ *rrule-ok SG C i j AndR*
| *Rrule-Imply*:⋀*p q*. *nth* (*snd* (*nth SG i*)) *j* = (*p* → *q*) ⟹ *rrule-ok SG C i j ImplyR*
| *Rrule-Equiv*:⋀*p q*. *nth* (*snd* (*nth SG i*)) *j* = (*p* ↔ *q*) ⟹ *rrule-ok SG C i j EquivR*
| *Rrule-Cohide*:*length* (*snd* (*nth SG i*)) > *j* ⟹ (⋀Γ *q*. (*nth SG i*) ≠ (Γ, [*q*])) ⟹ *rrule-ok SG C i j CohideR*
| *Rrule-CohideRR*:*length* (*snd* (*nth SG i*)) > *j* ⟹ (⋀*q*. (*nth SG i*) ≠ ([], [*q*])) ⟹ *rrule-ok SG C i j CohideRR*
| *Rrule-True*:*nth* (*snd* (*nth SG i*)) *j* = *TT* ⟹ *rrule-ok SG C i j TrueR*

inductive-simps
Rrule-And-simps[*prover*]: *rrule-ok SG C i j AndR*
and *Rrule-Imply-simps*[*prover*]: *rrule-ok SG C i j ImplyR*
and *Rrule-Equiv-simps*[*prover*]: *rrule-ok SG C i j EquivR*
and *Rrule-CohideR-simps*[*prover*]: *rrule-ok SG C i j CohideR*
and *Rrule-CohideRR-simps*[*prover*]: *rrule-ok SG C i j CohideRR*
and *Rrule-TrueR-simps*[*prover*]: *rrule-ok SG C i j TrueR*

inductive *step-ok* :: ('*sf*, '*sc*, '*sz*) *rule* ⇒ *nat* ⇒ ('*sf*, '*sc*, '*sz*) *step* ⇒ *bool*
where
Step-Axiom:(*nth SG i*) = ([], [*get-axiom a*]) ⟹ *step-ok* (*SG*,*C*) *i* (*Axiom a*)
| *Step-AxSubst*:(*nth SG i*) = ([], [*Fsubst* (*get-axiom a*) *σ*]) ⟹ *Fadmit σ* (*get-axiom a*) ⟹ *ssafe σ* ⟹ *step-ok* (*SG*,*C*) *i* (*AxSubst a σ*)
| *Step-Lrule*:*lrule-ok SG C i j L* ⟹ *j* < *length* (*fst* (*nth SG i*)) ⟹ *step-ok* (*SG*,*C*) *i* (*Lrule L j*)

| *Step-Rrule:rrule-ok SG C i j L* $\Longrightarrow$ *j* $<$ *length* (*snd* (*nth SG i*)) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*Rrule L j*)
| *Step-Cut:fsafe* φ $\Longrightarrow$ *i* $<$ *length SG* $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*Cut* φ)
| *Step-CloseId:nth* (*fst* (*nth SG i*)) *j* = *nth* (*snd* (*nth SG i*)) *k* $\Longrightarrow$ *j* $<$ *length* (*fst* (*nth SG i*)) $\Longrightarrow$ *k* $<$ *length* (*snd* (*nth SG i*)) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*CloseId j k*)
| *Step-G:*$\bigwedge$*a p. nth SG i* = ([], [([[*a*]]*p*)]) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i G*
| *Step-DEAxiom-schema*:
nth SG i =
([], [*Fsubst* ((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*)) *ODE*) (*p1 vid2 vid1*)]] (*P pid1*)) $\leftrightarrow$
([[*EvolveODE* ((*OProd* (*OSing vid1* (*f1 fid1 vid1*))) *ODE*) (*p1 vid2 vid1*)]]
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*))) σ])
$\Longrightarrow$ *ssafe* σ
$\Longrightarrow$ *osafe ODE*
$\Longrightarrow$ {*Inl vid1*, *Inr vid1*} $\cap$ *BVO ODE* = {}
$\Longrightarrow$ *Fadmit* σ ((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]] (*P pid1*)) $\leftrightarrow$
([[*EvolveODE* ((*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*)) (*p1 vid2 vid1*)]]
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*)))
$\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*DEAxiomSchema ODE* σ)
| *Step-CE:nth SG i* = ([], [*Fsubst* (*Equiv* (*InContext pid1* φ) (*InContext pid1* ψ)) σ])
$\Longrightarrow$ *valid* (*Equiv* φ ψ)
$\Longrightarrow$ *fsafe* φ
$\Longrightarrow$ *fsafe* ψ
$\Longrightarrow$ *ssafe* σ
$\Longrightarrow$ *Fadmit* σ (*Equiv* (*InContext pid1* φ) (*InContext pid1* ψ))
$\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*CE* φ ψ σ)
| *Step-CQ:nth SG i* = ([], [*Fsubst* (*Equiv* (*Prop p* (*singleton* ϑ)) (*Prop p* (*singleton* ϑ'))) σ])
$\Longrightarrow$ *valid* (*Equals* ϑ ϑ')
$\Longrightarrow$ *dsafe* ϑ
$\Longrightarrow$ *dsafe* ϑ'
$\Longrightarrow$ *ssafe* σ
$\Longrightarrow$ *Fadmit* σ (*Equiv* (*Prop p* (*singleton* ϑ)) (*Prop p* (*singleton* ϑ')))
$\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*CQ* ϑ ϑ' σ)

inductive-simps
Step-G-simps[*prover*]: *step-ok* (*SG*,*C*) *i G*
and *Step-CloseId-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*CloseId j k*)
and *Step-Cut-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Cut* φ)
and *Step-Rrule-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Rrule j L*)
and *Step-Lrule-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Lrule j L*)
and *Step-Axiom-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Axiom a*)
and *Step-AxSubst-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*AxSubst a* σ)
and *Step-DEAxiom-schema-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*DEAxiomSchema ODE* σ)
and *Step-CE-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*CE* φ ψ σ)
and *Step-CQ-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*CQ* ϑ ϑ' σ)

inductive *deriv-ok* :: (*'sf*, *'sc*, *'sz*) *rule* ⇒ (*'sf*, *'sc*, *'sz*) *derivation* ⇒ *bool*
where
Deriv-Nil:*deriv-ok R Nil*
| *Deriv-Cons*:*step-ok R i S* ⟹ *i* ≥ *0* ⟹ *i* < *length* (*fst R*) ⟹ *deriv-ok* (*step-result R* (*i*,*S*)) *SS* ⟹ *deriv-ok R* ((*i*,*S*) # *SS*)

inductive-simps
Deriv-nil-simps[*prover*]: *deriv-ok R Nil*
and *Deriv-cons-simps*[*prover*]: *deriv-ok R* ((*i*,*S*)#*SS*)

inductive *proof-ok* :: (*'sf*, *'sc*, *'sz*) *pf* ⇒ *bool*
where
Proof-ok:*deriv-ok* (*start-proof D*) *S* ⟹ *proof-ok* (*D*,*S*)

inductive-simps *Proof-ok-simps*[*prover*]: *proof-ok* (*D*,*S*)

15.1 Soundness

named-theorems *member-intros Prove that stuff is in lists*

lemma *mem-sing*[*member-intros*]:⋀*x*. *List.member* [*x*] *x*
by(*auto simp add*: *member-rec*)

lemma *mem-appR*[*member-intros*]:⋀*A B x*. *List.member B x* ⟹ *List.member* (*A* @ *B*) *x*
subgoal for *A* **by**(*induction A*, *auto simp add*: *member-rec*) **done**

lemma *mem-filter*[*member-intros*]:⋀*A P x*. *P x* ⟹ *List.member A x* ⟹ *List.member* (*filter P A*) *x*
subgoal for *A*
by(*induction A*, *auto simp add*: *member-rec*)
done

lemma *sound-weaken-appL*:⋀*SG SGS C*. *sound* (*SGS*, *C*) ⟹ *sound* (*SG* @ *SGS*, *C*)
subgoal for *SG SGS C*
apply(*rule sound-weaken-gen*)
apply(*auto*)
unfolding *sublist-def* **apply**(*rule allI*)
subgoal for *x*
using *mem-appR*[*of SGS x SG*] **by** *auto*
done
done

lemma *fml-seq-valid*:*valid* φ ⟹ *seq-valid* ([], [φ])
unfolding *seq-valid-def valid-def* **by** *auto*

lemma *closeI-provable-sound*:⋀*i*. *sound* (*SG*, *C*) ⟹ *sound* (*closeI SG i*, (*nth SG*

i)) ⟹ *sound* (*closeI SG i*, *C*)
using *close-provable-sound* **by** *auto*

lemma *valid-to-sound*:*seq-valid A* ⟹ *sound* (*B*, *A*)
unfolding *seq-valid-def sound-def* **by** *auto*

lemma *closeI-valid-sound*:⋀*i*. *sound* (*SG*, *C*) ⟹ *seq-valid* (*nth SG i*) ⟹ *sound* (*closeI SG i*, *C*)
using *valid-to-sound closeI-provable-sound* **by** *auto*

lemma *close-nonmember-eq*:¬(*List.member A a*) ⟹ *close A a* = *A*
by (*induction A*, *auto simp add*: *member-rec*)

lemma *close-noneq-nonempty*:*List.member A x* ⟹ *x* ≠ *a* ⟹ *close A a* ≠ []
by(*induction A*, *auto simp add*: *member-rec*)

lemma *close-app-neq*:*List.member A x* ⟹ *x* ≠ *a* ⟹ *close* (*A* @ *B*) *a* ≠ *B*
using *append-self-conv2*[*of close A a close B a*] *append-self-conv2*[*of close A a B*] *close-app-comm*[*of A B a*] *close-noneq-nonempty*[*of A x a*]
apply(*cases close B a* = *B*)
apply (*auto*)
by (*metis* (*no-types*, *lifting*) *filter-True filter-append mem-Collect-eq set-filter*)

lemma *member-singD*:⋀*x P*. *P x* ⟹ (⋀*y*. *List.member* [*x*] *y* ⟹ *P y*)
by (*metis member-rec*(*1*) *member-rec*(*2*))

lemma *fst-neq*:*A* ≠ *B* ⟹ (*A*,*C*) ≠ (*B*,*D*)
by *auto*

lemma *lrule-sound*: *lrule-ok SG C i j L* ⟹ *i* < *length SG* ⟹ *j* < *length* (*fst* (*SG* ! *i*)) ⟹ *sound* (*SG*,*C*) ⟹ *sound* (*close* (*append SG* (*Lrule-result L j* (*nth SG i*))) (*nth SG i*), *C*)
proof(*induction rule*: *lrule-ok.induct*)
case (*Lrule-And SG i j C p q*)
assume *eq*:*fst* (*SG* ! *i*) ! *j* = (*p* &&& *q*)
assume *sound*:*sound* (*SG*, *C*)
obtain *AI* **and** *SI* **where** *SG-dec*:(*AI*,*SI*) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have *AIjeq*:*AI* ! *j* = (*p* &&& *q*) **using** *SG-dec eq*
by (*metis fst-conv*)
have *sub*:*sublist* [(*close* ([*p*, *q*] @ *AI*) (*p* &&& *q*),*SI*)] ([*y*←*SG* . *y* ≠ (*AI*, *SI*)] @ [*y*← [(*close* (*p* # *q* # *AI*) (*p* &&& *q*), *SI*)] . *y* ≠ (*AI*, *SI*)])
apply (*rule sublistI*)
using *member-singD* [*of* λ*y*. *List.member* ([*y*←*SG* . *y* ≠ (*AI*, *SI*)] @ [*y*← [(*close* ([*p*, *q*] @ *AI*) (*p* &&& *q*), *SI*)] . *y* ≠ (*AI*, *SI*)]) *y* (*close* ([*p*, *q*] @ *AI*) (*p* &&& *q*),*SI*)]
using *close-app-neq*[*of* [*p*, *q*] *p p* &&& *q AI*]
by(*auto intro*: *member-intros fst-neq simp add*: *member-rec expr-diseq*)
have *cool*:*sound* ([*y*←*SG* . *y* ≠ (*AI*, *SI*)] @ [*y*← [(*close* (*p* # *q* # *AI*) (*p* &&

q), *SI*)] . *y* ≠ (*AI*, *SI*)], *AI*, *SI*)
apply(*rule sound-weaken-gen*[*OF sub*])
apply(*auto simp add*: *member-rec expr-diseq*)
unfolding *seq-valid-def*
proof (*rule soundI-mem*)
fix *I*::('*sf*,'*sc*,'*sz*) *interp*
assume *good*:*is-interp I*
assume *sgs*:(⋀φ. *List.member* [(*p* # *q* # [*y*←*AI* . *y* ≠ (*p* && *q*)], *SI*)] φ ⟹ *seq-sem I* φ = *UNIV*)
have *theSg*:*seq-sem I* (*p* # *q* # [*y*←*AI* . *y* ≠ (*p* && *q*)], *SI*) = *UNIV*
apply(*rule sgs*)
by(*auto intro*: *member-intros*)
then have *sgIn*:⋀ν. ν ∈ *seq-sem I* ((*p* && *q*) # [*y*←*AI* . *y* ≠ (*p* && *q*)], *SI*)
by *auto*
{ fix ν
assume *sem*:ν ∈ *seq-sem I* ((*p* && *q*) # [*y*←*AI* . *y* ≠ (*p* && *q*)], *SI*)
have *mem-eq*:⋀*x*. *List.member* ((*p* && *q*) # [*y*←*AI* . *y* ≠ (*p* && *q*)]) *x* = *List.member AI x*
by (*metis* (*mono-tags*, *lifting*) *Lrule-And.prems*(*2*) *SG-dec eq fst-conv local.member-filter mem-filter member-rec*(*1*) *nth-member*)
have *myeq*:ν ∈ *seq-sem I* ((*p* && *q*) # [*y*←*AI* . *y* ≠ (*p* && *q*)], *SI*) ⟹ ν ∈ *seq-sem I* (*AI*, *SI*)
using *and-foldl-sem and-foldl-sem-conv seq-semI Lrule-And.prems*(*2*) *SG-dec eq seq-MP seq-semI*′ *mem-eq*
by (*metis* (*no-types*, *lifting*))
have ν ∈ *seq-sem I* ((*p* && *q*) # [*y*←*AI* . *y* ≠ (*p* && *q*)], *SI*)
using *sem* **by** *auto*
then have ν ∈ *seq-sem I* ((*p* && *q*) # [*y*←*AI* . *y* ≠ (*p* && *q*)], *SI*)
by *blast*
then have ν ∈ *seq-sem I* (*AI*, *SI*)
using *myeq* **by** *auto***}**
then show *seq-sem I* (*AI*, *SI*) = *UNIV*
using *sgIn* **by** *blast*
qed
have *res-sound*:*sound* ([*y*←*SG* . *y* ≠ (*AI*,*SI*)] @ [*y*←*Lrule-result AndL j* (*AI*,*SI*) . *y* ≠ (*AI*,*SI*)],(*AI*,*SI*))
apply (*simp*)
using *cool AIjeq* **by** *auto*
show *?case*
apply(*rule close-provable-sound*)
apply(*rule sound-weaken-appR*)
apply(*rule sound*)
using *res-sound SG-dec* **by** *auto*
next
case (*Lrule-Imply SG i j C p q*)
have *implyL-simp*:⋀*AI SI SS p q*.
(*nth AI j*) = (*Not* (*And* (*Not q*) (*Not* (*Not p*)))) ⟹
(*AI*,*SI*) = *SS* ⟹
Lrule-result ImplyL j SS = [(*close* (*q* # *AI*) (*nth AI j*), *SI*), (*close AI* (*nth AI*

```
j), p # SI)]
    subgoal for AI SI SS p q apply(cases SS) by auto done
  assume eq:fst (SG ! i) ! j = (p → q)
  assume iL:i < length SG
  assume jL:j < length (fst (SG ! i))
  assume sound:sound (SG, C)
  obtain Γ and Δ where SG-dec:(Γ,Δ) = (SG ! i)
    by (metis seq2fml.cases)
  have res-eq:Lrule-result ImplyL j (SG ! i) =
    [(close (q # Γ) (nth Γ j), Δ),
     (close Γ (nth Γ j), p # Δ)]
    apply(rule implyL-simp)
    using SG-dec eq Implies-def Or-def
    by (metis fstI)+
  have AIjeq:Γ ! j = (p → q)
    using SG-dec eq unfolding Implies-def Or-def
    by (metis fst-conv)
  have big-sound:sound ([(close (q # Γ) (p → q), Δ), (close Γ (p → q), p # Δ)],
(Γ,Δ))
    apply(rule soundI')
    apply(rule seq-semI')
  proof −
    fix I::('sf,'sc,'sz) interp and ν::'sz state
    assume good:is-interp I
    assume sgs:(⋀i. 0 ≤ i ⟹
          i < length [(close (q # Γ) (p → q), Δ), (close Γ (p → q), p # Δ)] ⟹
          ν ∈ seq-sem I ([(close (q # Γ) (p → q), Δ), (close Γ (p → q), p # Δ)]
! i))
    have sg1:ν ∈ seq-sem I (close (q # Γ) (p → q), Δ) using sgs[of 0] by auto
    have sg2:ν ∈ seq-sem I (close Γ (p → q), p # Δ) using sgs[of Suc 0] by auto
    assume Γ:ν ∈ fml-sem I (foldr And Γ TT)
    have Γ-proj:⋀φ Γ. List.member Γ φ ⟹ ν ∈ fml-sem I (foldr And Γ TT) ⟹
ν ∈ fml-sem I φ
      apply(induction Γ, auto simp add: member-rec)
      using and-foldl-sem by blast
    have imp:ν ∈ fml-sem I (p → q)
      apply(rule Γ-proj[of Γ])
      using AIjeq jL SG-dec nth-member
      apply (metis fst-conv)
      by (rule Γ)
    have sub:sublist (close Γ (p → q)) Γ
      by (rule close-sub)
    have ΓC:ν ∈ fml-sem I (foldr And (close Γ (p → q)) TT)
      by (rule Γ-sub-sem[OF sub Γ])
    have ν ∈ fml-sem I (foldr (||) (p # Δ) FF)
      by(rule seq-MP[OF sg2 ΓC])
    then have disj:ν ∈ fml-sem I p ∨ ν ∈ fml-sem I (foldr (||) Δ FF)
      by auto
    { assume p:ν ∈ fml-sem I p
```

```
    have q:ν ∈ fml-sem I q using p imp by simp
    have res: ν ∈ fml-sem I (foldr (||) Δ FF)
     using disj Γ seq-semI
     proof −
      have ν ∈ fml-sem I (foldr (&&) (q # Γ) TT)
       using Γ q by auto
      then show ?thesis
       by (meson Γ-sub-sem close-sub seq-MP sg1)
     qed
    have conj:ν ∈ fml-sem I (foldr (&&) (q # Γ) TT)
     using q Γ by auto
    have conj:ν ∈ fml-sem I (foldr (&&) (close (q # Γ) (p → q)) TT)
     apply(rule Γ-sub-sem)
     defer
     apply(rule conj)
     by(rule close-sub)
    have Δ1:ν ∈ fml-sem I (foldr (||) Δ FF)
     by(rule seq-MP[OF sg1 conj])
    }
   then show ν ∈ fml-sem I (foldr (||) Δ FF)
    using disj by auto
   qed
   have neq1:close ([q] @ Γ) (p → q) ≠ Γ
    apply(rule close-app-neq)
     apply(rule mem-sing)
    by (auto simp add: expr-diseq)
   have neq2:p # Δ ≠ Δ
    by(induction p, auto)
    have close-eq:close [(close (q # Γ) (p → q), Δ), (close Γ (p → q), p # Δ)]
(Γ,Δ) = [(close (q # Γ) (p → q), Δ), (close Γ (p → q), p # Δ)]
    apply(rule close-nonmember-eq)
    apply auto
     using neq1 neq2
     apply (simp add: member-rec)
   proof −
    assume a1: q = (p → q)
    assume List.member [([y←Γ . y ≠ q], Δ), ([y←Γ . y ≠ q], p # Δ)] (Γ, Δ)
     then have [f←Γ . f ≠ q] = Γ
    by (simp add: member-rec)
    then show False
     using a1 neq1 by fastforce
   qed
  show ?case
   apply(rule close-provable-sound)
    apply(rule sound-weaken-appR)
    apply(rule sound)
   apply(unfold res-eq)
   apply(unfold AIjeq)
   unfolding close-app-comm
```

```
    apply (rule sound-weaken-appL)
    using close-eq big-sound SG-dec
    by simp
next
  case (Lrule-EquivBackward SG i j C p q)
  have equivLBackward-simp:⋀AI SI SS p q.
    (nth AI  j) = Not (And (Not (And p q)) (Not (And (Not p) (Not q)))) ⟹
    (AI,SI) = SS ⟹
    Lrule-result EquivBackwardL j SS = [(close (p # AI) (nth AI j), SI), (close AI
(nth AI j), q # SI)]
    subgoal for AI SI SS p q apply(cases SS) by auto done
  assume eq:fst (SG ! i) ! j = (p ↔ q)
  assume iL:i < length SG
  assume jL:j < length (fst (SG ! i))
  assume sound:sound (SG, C)
  obtain Γ and Δ where SG-dec:(Γ,Δ) = (SG ! i)
    by (metis seq2fml.cases)
  have res-eq:Lrule-result EquivBackwardL j (SG ! i) =
    [(close (p # Γ) (nth Γ j), Δ),
     (close Γ (nth Γ j), q # Δ)]
    apply(rule equivLBackward-simp)
     using SG-dec eq Equiv-def Or-def
     by (metis fstI)+
  have AIjeq:Γ ! j = (p ↔ q)
    using SG-dec eq unfolding Implies-def Or-def
    by (metis fst-conv)
  have big-sound:sound ([(close (p # Γ) (p ↔ q), Δ), (close Γ (p ↔ q), q # Δ)],
(Γ,Δ))
    apply(rule soundI')
    apply(rule seq-semI')
  proof −
    fix I::('sf,'sc,'sz) interp and ν::'sz state
    assume good:is-interp I
    assume sgs:(⋀i. 0 ≤ i ⟹
          i < length [(close (p # Γ) (p ↔ q), Δ), (close Γ (p ↔ q), q # Δ)] ⟹
          ν ∈ seq-sem I ([(close (p # Γ) (p ↔ q), Δ), (close Γ (p ↔ q), q # Δ)]
! i))
    have sg1:ν ∈ seq-sem I (close (p # Γ) (p ↔ q), Δ) using sgs[of 0] by auto
    have sg2:ν ∈ seq-sem I (close Γ (p ↔ q), q # Δ) using sgs[of Suc 0] by auto
    assume Γ:ν ∈ fml-sem I (foldr And Γ TT)
    have Γ-proj:⋀φ Γ. List.member Γ φ ⟹ ν ∈ fml-sem I (foldr And Γ TT) ⟹
ν ∈ fml-sem I φ
      apply(induction Γ, auto simp add: member-rec)
      using and-foldl-sem by blast
    have imp:ν ∈ fml-sem I (p ↔ q)
      apply(rule Γ-proj[of Γ])
      using AIjeq  jL SG-dec nth-member
      apply (metis fst-conv)
      by (rule Γ)
```

have *sub*:*sublist* (*close* Γ (*p* → *q*)) Γ
by (*rule close-sub*)
have Γ*C*:ν ∈ *fml-sem I* (*foldr And* (*close* Γ (*p* → *q*)) *TT*)
by (*rule* Γ-*sub-sem*[*OF sub* Γ])
have ν ∈ *fml-sem I* (*foldr* (||) (*p* # Δ) *FF*)
by (*metis* Γ Γ-*sub-sem close-sub iff-sem imp member-rec*(*1*) *or-foldl-sem or-foldl-sem-conv seq-MP sg2*)
then have *disj*:ν ∈ *fml-sem I p* ∨ ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
by *auto*
{ assume *p*:ν ∈ *fml-sem I p*
have *q*:ν ∈ *fml-sem I q* **using** *p imp* **by** *simp*
have *res*: ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
using *disj* Γ *seq-semI*
proof −
have ν ∈ *fml-sem I* (*foldr* (&&) (*q* # Γ) *TT*)
using Γ *q* **by** *auto*
then show *?thesis*
proof −
have ∀*fs p i*. (∃*f*. *List.member fs* (*f*::('*sf*, '*sc*, '*sz*) *formula*) ∧ *p* ∉ *fml-sem i f*) ∨ *p* ∈ *fml-sem i* (*foldr* (&&) *fs TT*)
using *and-foldl-sem-conv* **by** *blast*
then obtain *ff* :: ('*sf*, '*sc*, '*sz*) *formula list* ⇒ (*real*, '*sz*) *vec* × (*real*, '*sz*) *vec* ⇒ ('*sf*, '*sc*, '*sz*) *interp* ⇒ ('*sf*, '*sc*, '*sz*) *formula* **where**
f1: ∀*fs p i*. *List.member fs* (*ff fs p i*) ∧ *p* ∉ *fml-sem i* (*ff fs p i*) ∨ *p* ∈ *fml-sem i* (*foldr* (&&) *fs TT*)
by *metis*
have ⋀*f*. ν ∈ *fml-sem I f* ∨ ¬ *List.member* Γ *f*
by (*meson* ‹ν ∈ *fml-sem I* (*foldr* (&&) (*q* # Γ) *TT*)› *and-foldl-sem member-rec*(*1*))
then have ν ∈ *fml-sem I* (*foldr* (&&) (*close* (*p* # Γ) (*p* ↔ *q*)) *TT*)
using *f1* **by** (*metis* (*no-types*) *close-sub local.sublist-def member-rec*(*1*) *p*)
then show *?thesis*
using *seq-MP sg1* **by** *blast*
qed
qed
have *conj*:ν ∈ *fml-sem I* (*foldr* (&&) (*q* # Γ) *TT*)
using *q* Γ **by** *auto*
have *conj*:ν ∈ *fml-sem I* (*foldr* (&&) (*close* (*q* # Γ) (*p* → *q*)) *TT*)
apply(*rule* Γ-*sub-sem*)
defer
apply(*rule conj*)
by(*rule close-sub*)
have Δ*1*:ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
using *res* **by** *blast*
}
then show ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
using *disj* **by** *auto*
qed

have *neq1*:*close* ([*q*] @ Γ) (*p* ↔ *q*) ≠ Γ
apply(*rule close-app-neq*)
apply(*rule mem-sing*)
by (*auto simp add*: *expr-diseq*)
have *neq2*:*p* # Δ ≠ Δ
by(*induction p*, *auto*)
have *close-eq*:*close* [(*close* (*p* # Γ) (*p* ↔ *q*), Δ), (*close* Γ (*p* ↔ *q*), *q* # Δ)] (Γ,Δ)
= [(*close* (*p* # Γ) (*p* ↔ *q*), Δ), (*close* Γ (*p* ↔ *q*), *q* # Δ)]
apply(*rule close-nonmember-eq*)
apply *auto*
using *neq1 neq2*
apply (*simp add*: *member-rec*)
apply (*metis append-Cons append-Nil close.simps close-app-neq member-rec*(*1*))
proof −
assume *a1*:*p* = (*p* ↔ *q*)
then show *False*
by (*simp add*: *expr-diseq*)
qed
show *?case*
apply(*rule close-provable-sound*)
apply(*rule sound-weaken-appR*)
apply(*rule sound*)
apply(*unfold res-eq*)
apply(*unfold AIjeq*)
unfolding *close-app-comm*
apply (*rule sound-weaken-appL*)
using *close-eq big-sound SG-dec*
by *simp*
next
case (*Lrule-EquivForward SG i j C p q*)
have *equivLForward-simp*:⋀*AI SI SS p q*.
(*nth AI j*) = *Not* (*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p*) (*Not q*)))) ⟹
(*AI*,*SI*) = *SS* ⟹
Lrule-result EquivForwardL j SS = [(*close* (*q* # *AI*) (*nth AI j*), *SI*), (*close AI*
(*nth AI j*), *p* # *SI*)]
subgoal for *AI SI SS p q* **apply**(*cases SS*) **by** *auto* **done**
assume *eq*:*fst* (*SG* ! *i*) ! *j* = (*p* ↔ *q*)
assume *iL*:*i* < *length SG*
assume *jL*:*j* < *length* (*fst* (*SG* ! *i*))
assume *sound*:*sound* (*SG*, *C*)
obtain Γ **and** Δ **where** *SG-dec*:(Γ,Δ) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have *res-eq*:*Lrule-result EquivForwardL j* (*SG* ! *i*) =
[(*close* (*q* # Γ) (*nth* Γ *j*), Δ),
(*close* Γ (*nth* Γ *j*), *p* # Δ)]
apply(*rule equivLForward-simp*)
using *SG-dec eq Equiv-def Or-def*
by (*metis fstI*)+
have *AIjeq*:Γ ! *j* = (*p* ↔ *q*)

using *SG-dec eq* **unfolding** *Implies-def Or-def*
by (*metis fst-conv*)
have *big-sound*:*sound* ([(*close* (*q* # Γ) (*p* ↔ *q*), Δ), (*close* Γ (*p* ↔ *q*), *p* # Δ)], (Γ,Δ))
apply(*rule soundI'*)
apply(*rule seq-semI'*)
proof −
fix *I*::('*sf*,'*sc*,'*sz*) *interp* **and** ν::'*sz state*
assume *good*:*is-interp I*
assume *sgs*:(⋀*i*. *0* ≤ *i* ⟹
i < *length* [(*close* (*q* # Γ) (*p* ↔ *q*), Δ), (*close* Γ (*p* ↔ *q*), *p* # Δ)] ⟹
ν ∈ *seq-sem I* ([(*close* (*q* # Γ) (*p* ↔ *q*), Δ), (*close* Γ (*p* ↔ *q*), *p* # Δ)] ! *i*))
have *sg1*:ν ∈ *seq-sem I* (*close* (*q* # Γ) (*p* ↔ *q*), Δ) **using** *sgs*[*of 0*] **by** *auto*
have *sg2*:ν ∈ *seq-sem I* (*close* Γ (*p* ↔ *q*), *p* # Δ) **using** *sgs*[*of Suc 0*] **by** *auto*

assume Γ:ν ∈ *fml-sem I* (*foldr And* Γ *TT*)
have Γ-*proj*:⋀φ Γ. *List.member* Γ φ ⟹ ν ∈ *fml-sem I* (*foldr And* Γ *TT*) ⟹ ν ∈ *fml-sem I* φ
apply(*induction* Γ, *auto simp add*: *member-rec*)
using *and-foldl-sem* **by** *blast*
have *imp*:ν ∈ *fml-sem I* (*p* ↔ *q*)
apply(*rule* Γ-*proj*[*of* Γ])
using *AIjeq jL SG-dec nth-member*
apply (*metis fst-conv*)
by (*rule* Γ)
have *sub*:*sublist* (*close* Γ (*p* → *q*)) Γ
by (*rule close-sub*)
have Γ*C*:ν ∈ *fml-sem I* (*foldr And* (*close* Γ (*p* → *q*)) *TT*)
by (*rule* Γ-*sub-sem*[*OF sub* Γ])
have ν ∈ *fml-sem I* (*foldr* (||) (*p* # Δ) *FF*)
by (*metis* Γ Γ-*sub-sem close-sub iff-sem imp member-rec*(*1*) *or-foldl-sem or-foldl-sem-conv seq-MP sg2*)
then have *disj*:ν ∈ *fml-sem I p* ∨ ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
by *auto*
{ assume *p*:ν ∈ *fml-sem I p*
have *q*:ν ∈ *fml-sem I q* **using** *p imp* **by** *simp*
have *res*: ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
using *disj* Γ *seq-semI*
proof −
have ν ∈ *fml-sem I* (*foldr* (&&) (*q* # Γ) *TT*)
using Γ *q* **by** *auto*
then show *?thesis*
by (*meson* ‹ν ∈ *fml-sem I* (*foldr* (&&) (*q* # Γ) *TT*)› *and-foldl-sem and-foldl-sem-conv close-sub local.sublist-def seq-MP sg1*)
qed
have *conj*:ν ∈ *fml-sem I* (*foldr* (&&) (*q* # Γ) *TT*)
using *q* Γ **by** *auto*
have *conj*:ν ∈ *fml-sem I* (*foldr* (&&) (*close* (*q* # Γ) (*p* → *q*)) *TT*)

```
      apply(rule Γ-sub-sem)
      defer
      apply(rule conj)
      by(rule close-sub)
    have Δ1:ν ∈ fml-sem I (foldr (||) Δ FF)
     using res by blast
    }
  then show ν ∈ fml-sem I (foldr (||) Δ FF)
   using disj by auto
 qed
 have neq1:close ([q] @ Γ) (p ↔ q) ≠ Γ
  apply(rule close-app-neq)
  apply(rule mem-sing)
  by (auto simp add: expr-diseq)
 have neq2:p # Δ ≠ Δ
  by(induction p, auto)
 have close-eq:close [(close (q # Γ) (p ↔ q), Δ), (close Γ (p ↔ q), p # Δ)] (Γ,Δ)
= [(close (q # Γ) (p ↔ q), Δ), (close Γ (p ↔ q), p # Δ)]
  apply(rule close-nonmember-eq)
  apply auto
   using neq1 neq2
   apply (simp add: member-rec)
 proof −
   assume a1:q = (p ↔ q)
   then show False
    by (simp add: expr-diseq)
 qed
 show ?case
  apply(rule close-provable-sound)
   apply(rule sound-weaken-appR)
   apply(rule sound)
  apply(unfold res-eq)
  apply(unfold AIjeq)
  unfolding close-app-comm
  apply (rule sound-weaken-appL)
  using close-eq big-sound SG-dec
 by simp
qed

lemma rrule-sound: rrule-ok SG C i j L ⟹ i < length SG ⟹ j < length (snd
(SG ! i)) ⟹ sound (SG,C) ⟹ sound (close (append SG (Rrule-result L j (nth
SG i))) (nth SG i), C)
proof(induction rule: rrule-ok.induct)
 case (Rrule-And SG i j C p q)
 assume eq:snd (SG ! i) ! j = (p && q)
 assume i < length SG
 assume j < length (snd (SG ! i))
 assume sound:sound (SG, C)
 obtain Γ and Δ where SG-dec:(Γ,Δ) = (SG ! i)
```

by (*metis seq2fml.cases*)
have *andR-simp*:$\bigwedge\Gamma\ \Delta\ SS\ p\ q$.
($nth\ \Delta\ j$) = $And\ p\ q \Longrightarrow$
(Γ,Δ) = $SS \Longrightarrow$
Rrule-result AndR j SS = [(Γ, p # (*close* Δ (*nth* Δ j))), (Γ, q # (*close* Δ (*nth* Δ j)))]
subgoal for *AI SI SS p q* **apply**(*cases SS*) **by** *auto* **done**
have *res-eq*:*Rrule-result AndR j* (*SG* ! *i*) =
[(Γ, p # (*close* Δ (*nth* Δ j))), (Γ, q # (*close* Δ (*nth* Δ j)))]
using *SG-dec andR-simp* **apply** *auto*
using *SG-dec eq Implies-def Or-def*
using *fstI*
by (*metis andR-simp close.simps snd-conv*)
have *AIjeq*:Δ ! j = (p &&& q)
using *SG-dec eq snd-conv*
by *metis*
have *big-sound*:*sound* ([(Γ, p # (*close* Δ (*nth* Δ j))), (Γ, q # (*close* Δ (*nth* Δ j)))], (Γ,Δ))
apply(*rule soundI'*)
apply(*rule seq-semI'*)
proof −
fix I::('*sf*,'*sc*,'*sz*) *interp* **and** ν
assume *good*:*is-interp I*
assume *sgs*:($\bigwedge i.\ 0 \leq i \Longrightarrow$
i < *length* [(Γ, p # *close* Δ (*nth* Δ j)), (Γ, q # *close* Δ (*nth* Δ j))] $\Longrightarrow$
$\nu \in$ *seq-sem I* (*nth* [(Γ, p # *close* Δ (*nth* Δ j)), (Γ, q # *close* Δ (*nth* Δ j))] i))
assume Γ-*sem*:$\nu \in$ *fml-sem I* (*foldr* (&&) Γ *TT*)
have *sg1*:$\nu \in$ *seq-sem I* (Γ, p # *close* Δ (*nth* Δ j)) **using** *sgs*[*of 0*] **by** *auto*
have *sg2*:$\nu \in$ *seq-sem I* (Γ, q # *close* Δ (*nth* Δ j)) **using** *sgs*[*of 1*] **by** *auto*
have Δ*1*:$\nu \in$ *fml-sem I* (*foldr* (||) (p # *close* Δ (*nth* Δ j)) *FF*)
by(*rule seq-MP*[*OF sg1* Γ-*sem*])
have Δ*2*:$\nu \in$ *fml-sem I* (*foldr* (||) (q # *close* Δ (*nth* Δ j)) *FF*)
by(*rule seq-MP*[*OF sg2* Γ-*sem*])
have Δ':$\nu \in$ *fml-sem I* (*foldr* (||) ((p &&& q) # *close* Δ (*nth* Δ j)) *FF*)
using Δ*1* Δ*2* **by** *auto*
have *mem-eq*:$\bigwedge x$. *List.member* ((p &&& q) # *close* Δ (*nth* Δ j)) $x \Longrightarrow$ *List.member* Δ x
using *Rrule-And.prems SG-dec eq member-rec*(*1*) *nth-member*
by (*metis close-sub local.sublist-def snd-conv*)
have *myeq*:$\nu \in$ *fml-sem I* (*foldr* (||) ((p &&& q) # *close* Δ (*nth* Δ j)) *FF*) $\Longrightarrow$ $\nu \in$ *fml-sem I* (*foldr* (||) Δ *FF*)
using *seq-semI Rrule-And.prems SG-dec eq seq-MP seq-semI' mem-eq or-foldl-sem or-foldl-sem-conv*
by *metis*
then show $\nu \in$ *fml-sem I* (*foldr* (||) Δ *FF*)
using Δ' **by** *auto*
qed

have *list-neqI1*:$\bigwedge$*L1 L2 x. List.member L1 x* $\Longrightarrow$ $\neg$(*List.member L2 x*) $\Longrightarrow$ *L1* $\neq$ *L2*
by(*auto*)
have *list-neqI2*:$\bigwedge$*L1 L2 x.* $\neg$(*List.member L1 x*) $\Longrightarrow$ (*List.member L2 x*) $\Longrightarrow$ *L1* $\neq$ *L2*
by(*auto*)
have *notin-cons*:$\bigwedge$*x y ys. x* $\neq$ *y* $\Longrightarrow$ $\neg$(*List.member ys x*) $\Longrightarrow$ $\neg$(*List.member* (*y* # *ys*) *x*)
subgoal for *x y ys*
by(*induction ys*, *auto simp add*: *member-rec*)
done
have *notin-close*:$\bigwedge$*L x.* $\neg$(*List.member* (*close L x*) *x*)
subgoal for *L x*
by(*induction L*, *auto simp add*: *member-rec*)
done
have *neq-lemma*:$\bigwedge$*L x y. List.member L x* $\Longrightarrow$ *y* $\neq$ *x* $\Longrightarrow$ (*y* # (*close L x*)) $\neq$ *L*
subgoal for *L x y*
apply(*cases List.member L y*)
subgoal
apply(*rule list-neqI2*[*of y* # *close L x x*])
apply(*rule notin-cons*)
defer
apply(*rule notin-close*)
by(*auto*)
subgoal
apply(*rule list-neqI2*[*of y* # *close L x x*])
apply(*rule notin-cons*)
defer
apply(*rule notin-close*)
by(*auto*)
done
done
have *neq1*:*p* # *close* Δ (*p* && *q*) $\neq$ Δ
apply(*rule neq-lemma*)
apply (*metis Rrule-And.prems*(*2*) *SG-dec eq nth-member sndI*)
by(*auto simp add*: *expr-diseq*)
have *neq2*:*q* # *close* Δ (*p* && *q*) $\neq$ Δ
apply(*rule neq-lemma*)
apply (*metis Rrule-And.prems*(*2*) *SG-dec eq nth-member sndI*)
by(*auto simp add*: *expr-diseq*)
have *close-eq*:*close* [(Γ, *p* # *close* Δ (*p* && *q*)), (Γ, *q* # *close* Δ (*p* && *q*))] (Γ,Δ) = [(Γ, *p* # *close* Δ (*p* && *q*)), (Γ, *q* # *close* Δ (*p* && *q*))]
apply(*rule close-nonmember-eq*)
apply *auto*
using *neq1 neq2*
by (*simp add*: *member-rec*)
show *sound* (*close* (*SG* @ *Rrule-result AndR j* (*SG* ! *i*)) (*SG* ! *i*), *C*)
apply(*rule close-provable-sound*)
apply(*rule sound-weaken-appR*)

apply(*rule sound*)
apply(*unfold res-eq*)
apply(*unfold AIjeq*)
unfolding *close-app-comm*
apply (*rule sound-weaken-appL*)
using *close-eq big-sound SG-dec*
by (*simp add*: *AIjeq*)
next
case (*Rrule-Imply SG i j C p q*)
assume *eq*:*snd* (*SG* ! *i*) ! *j* = (*p* → *q*)
assume *i* < *length SG*
assume *j* < *length* (*snd* (*SG* ! *i*))
assume *sound*:*sound* (*SG*, *C*)
obtain Γ **and** Δ **where** *SG-dec*:(Γ,Δ) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have *impR-simp*:⋀Γ Δ *SS p q*.
(*nth* Δ *j*) = *Implies p q* ⟹
(Γ,Δ) = *SS* ⟹
Rrule-result ImplyR j SS = [(*p* # Γ, *q* # (*close* Δ (*nth* Δ *j*)))]
subgoal for *AI SI SS p q* **apply**(*cases SS*) **by** (*auto simp add*: *Implies-def Or-def*) **done**
have *res-eq*:*Rrule-result ImplyR j* (*SG* ! *i*) =
[(*p* # Γ, *q* # (*close* Δ (*nth* Δ *j*)))]
using *SG-dec impR-simp* **apply** *auto*
using *SG-dec eq Implies-def Or-def*
using *fstI*
by (*metis impR-simp close.simps snd-conv*)
have *AIjeq*:Δ ! *j* = (*p* → *q*)
using *SG-dec eq snd-conv*
by *metis*
have *close-eq*:*close* [(*p* # Γ, *q* # (*close* Δ (*nth* Δ *j*)))] (Γ,Δ) = [(*p* # Γ, *q* # (*close* Δ (*nth* Δ *j*)))]
apply(*rule close-nonmember-eq*)
by (*simp add*: *member-rec*)
have *big-sound*:*sound* ([(*p* # Γ, *q* # *close* Δ (Δ ! *j*))], (Γ,Δ))
apply(*rule soundI'*)
apply(*rule seq-semI'*)
proof −
fix *I* ::('*sf*,'*sc*,'*sz*) *interp* **and** ν::'*sz state*
assume *is-interp I*
assume *sgs*:(⋀*i*. *0* ≤ *i* ⟹ *i* < *length* [(*p* # Γ, *q* # *close* Δ (Δ ! *j*))] ⟹ ν ∈ *seq-sem I* ([(*p* # Γ, *q* # *close* Δ (Δ ! *j*))] ! *i*))
have *sg*:ν ∈ *seq-sem I* (*p* # Γ, *q* # *close* Δ (Δ ! *j*)) **using** *sgs*[*of 0*] **by** *auto*
assume Γ-*sem*:ν ∈ *fml-sem I* (*foldr* (&&) Γ *TT*)
show ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
using Γ-*sem sg*
AIjeq Rrule-Imply.prems(*2*) *SG-dec and-foldl-sem-conv close-sub impl-sem local.sublist-def member-rec*(*1*) *nth-member or-foldl-sem-conv seq-MP seq-semI snd-conv* Γ-*sub-sem and-foldl-sem or-foldl-sem seq-sem.simps sublistI*

proof −
have *f1*: ∀*fs p i*. ∃*f*. (*p* ∈ *fml-sem i* (*foldr* (&&) *fs* (*TT*::('*sf*, '*sc*, '*sz*) *formula*)) ∨ *List.member fs f*) ∧ (*p* ∉ *fml-sem i f* ∨ *p* ∈ *fml-sem i* (*foldr* (&&) *fs TT*))
using *and-foldl-sem-conv* **by** *blast*
have ∀*p i fs*. ∃*f*. ∀*pa ia fa fb pb ib fc fd*. *p* ∈ *fml-sem i* (*f*::('*sf*, '*sc*, '*sz*) *formula*) ∧ (*pa* ∈ *fml-sem ia* (*fa*::('*sf*, '*sc*, '*sz*) *formula*) ∨ *pa* ∈ *fml-sem ia* (*fa* → *fb*)) ∧ (*pb* ∉ *fml-sem ib* (*fc*::('*sf*, '*sc*, '*sz*) *formula*) ∨ *pb* ∈ *fml-sem ib* (*fd* → *fc*)) ∧ (*p* ∉ *fml-sem i* (*foldr* (||) *fs FF*) ∨ *List.member fs f*)
by (*metis impl-sem or-foldl-sem-conv*)
then obtain *ff* :: (*real*, '*sz*) *vec* × (*real*, '*sz*) *vec* ⇒ ('*sf*, '*sc*, '*sz*) *interp* ⇒ ('*sf*, '*sc*, '*sz*) *formula list* ⇒ ('*sf*, '*sc*, '*sz*) *formula* **where**
f2: ⋀*p i fs pa ia f fa pb ib fb fc*. *p* ∈ *fml-sem i* (*ff p i fs*) ∧ (*pa* ∈ *fml-sem ia* (*f*::('*sf*, '*sc*, '*sz*) *formula*) ∨ *pa* ∈ *fml-sem ia* (*f* → *fa*)) ∧ (*pb* ∉ *fml-sem ib* (*fb*::('*sf*, '*sc*, '*sz*) *formula*) ∨ *pb* ∈ *fml-sem ib* (*fc* → *fb*)) ∧ (*p* ∉ *fml-sem i* (*foldr* (||) *fs FF*) ∨ *List.member fs* (*ff p i fs*))
by *metis*
then have ⋀*fs*. *ν* ∉ *fml-sem I* (*foldr* (&&) (*p* # Γ) *TT*) ∨ ¬ *local.sublist* (*close* Δ (*p* → *q*)) *fs* ∨ *ff ν I* (*q* # *close* Δ (*p* → *q*)) = *q* ∨ *List.member fs* (*ff ν I* (*q* # *close* Δ (*p* → *q*)))
by (*metis* (*no-types*) *AIjeq local.sublist-def member-rec*(*1*) *seq-MP sg*)
then have ∃*f*. *List.member* Δ *f* ∧ *ν* ∈ *fml-sem I f*
using *f2 f1* **by** (*metis* (*no-types*) *AIjeq Rrule-Imply.prems*(*2*) *SG-dec* Γ*-sem and-foldl-sem close-sub member-rec*(*1*) *nth-member snd-conv*)
then show *?thesis*
using *or-foldl-sem* **by** *blast*
qed
qed
show *?case*
apply(*rule close-provable-sound*)
apply(*rule sound-weaken-appR*)
apply(*rule sound*)
using *res-eq*
apply(*unfold res-eq*)
apply(*unfold AIjeq*)
unfolding *close-app-comm*
apply (*rule sound-weaken-appL*)
using *close-eq big-sound SG-dec AIjeq*
by (*simp add*: *AIjeq*)
next
case (*Rrule-Cohide SG i j C*)
assume *i* < *length SG*
assume *j* < *length* (*snd* (*SG* ! *i*))
assume *chg*:(⋀Γ *q*. (*nth SG i*) ≠ (Γ, [*q*]))
assume *sound*:*sound* (*SG*, *C*)
obtain Γ **and** Δ **where** *SG-dec*:(Γ,Δ) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have *cohideR-simp*:
(Γ,Δ) = *SS* ⟹

Rrule-result CohideR j SS = [(Γ, [*nth* Δ *j*])] **for** Γ Δ *SS p q*
by (*cases SS*, *auto*)
have *res-eq*:*Rrule-result CohideR j* (*SG* ! *i*) = [(Γ, [*nth* Δ *j*])]
using *SG-dec* **by** (*rule cohideR-simp*)
have *close-eq*:*close* [(Γ, [*nth* Δ *j*])] (Γ,Δ) = [(Γ, [*nth* Δ *j*])]
using *chg*
by (*metis SG-dec close-nonmember-eq member-rec*(*1*) *member-rec*(*2*))
have *big-sound*:*sound* ([(Γ, [*nth* Δ *j*])], (Γ,Δ))
apply(*rule soundI'*)
apply(*rule seq-semI'*)
by (*metis* (*no-types*, *lifting*) *Rrule-Cohide.prems*(*2*) *SG-dec length-greater-0-conv less-or-eq-imp-le list.distinct*(*1*) *member-singD nth-Cons-0 nth-member or-foldl-sem or-foldl-sem-conv seq-MP snd-conv*)
show *?case*
apply(*rule close-provable-sound*)
apply(*rule sound-weaken-appR*)
apply(*rule sound*)
using *res-eq*
apply(*unfold res-eq*)
unfolding *close-app-comm*
apply (*rule sound-weaken-appL*)
using *big-sound SG-dec*
apply(*cases* [*nth* Δ *j*] = Δ)
apply(*auto*)
using *chg* **by** (*metis*)+
next
case (*Rrule-CohideRR SG i j C*)
assume *i* < *length SG*
assume *j* < *length* (*snd* (*SG* ! *i*))
assume *chg*:(⋀*q*. (*nth SG i*) ≠ ([], [*q*]))
assume *sound*:*sound* (*SG*, *C*)
obtain Γ **and** Δ **where** *SG-dec*:(Γ,Δ) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have *cohideRR-simp*:
(Γ,Δ) = *SS* ⟹
Rrule-result CohideRR j SS = [([], [*nth* Δ *j*])] **for** Γ Δ *SS p q*
by (*cases SS*, *auto*)
have *res-eq*:*Rrule-result CohideRR j* (*SG* ! *i*) = [([], [*nth* Δ *j*])]
using *SG-dec* **by** (*rule cohideRR-simp*)
have *close-eq*:*close* [([], [*nth* Δ *j*])] (Γ,Δ) = [([], [*nth* Δ *j*])]
using *chg*
by (*metis SG-dec close-nonmember-eq member-rec*(*1*) *member-rec*(*2*))
have *big-sound*:*sound* ([([], [*nth* Δ *j*])], (Γ,Δ))
apply(*rule soundI'*)
apply(*rule seq-semI'*)
by (*metis* (*no-types*, *lifting*) *Rrule-CohideRR.prems*(*2*) *SG-dec and-foldl-sem-conv length-greater-0-conv less-or-eq-imp-le list.distinct*(*1*) *member-rec*(*2*) *member-singD nth-Cons-0 nth-member or-foldl-sem or-foldl-sem-conv seq-MP snd-conv*)
show *?case*

apply(*rule close-provable-sound*)
apply(*rule sound-weaken-appR*)
apply(*rule sound*)
using *res-eq*
apply(*unfold res-eq*)
unfolding *close-app-comm*
apply (*rule sound-weaken-appL*)
using *big-sound SG-dec*
apply(*cases* [*nth* Δ *j*] = Δ)
apply(*auto*)
using *chg* **by** (*metis*)+
next
case (*Rrule-True SG i j C*)
assume *tt*:*snd* (*SG* ! *i*) ! *j* = *TT*
assume *iL*:*i* < *length SG*
assume *iJ*:*j* < *length* (*snd* (*SG* ! *i*))
assume *sound*:*sound* (*SG*, *C*)
obtain Γ **and** Δ **where** *SG-dec*:(Γ,Δ) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have ⋀*I* ν. *is-interp I* ⟹ ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
proof −
fix *I*::('*sf*,'*sc*,'*sz*)*interp* **and** ν::'*sz state*
assume *good*:*is-interp I*
have *mem2*:*List.member* Δ (Δ ! *j*)
using *iJ nth-member*
by (*metis SG-dec snd-conv*)
then show ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
using *mem2*
using *or-foldl-sem*
by (*metis SG-dec UNIV-I snd-conv tt tt-sem*)
qed
then have *seq-valid*:*seq-valid* (*SG* ! *i*)
unfolding *seq-valid-def* **using** *SG-dec*
by (*metis UNIV-eq-I seq-semI'*)
show *?case*
using *closeI-valid-sound*[*OF sound seq-valid*]
by (*simp add*: *sound-weaken-appR*)
next
case (*Rrule-Equiv SG i j C p q*)
assume *eq*:*snd* (*SG* ! *i*) ! *j* = (*p* ↔ *q*)
assume *iL*:*i* < *length SG*
assume *jL*:*j* < *length* (*snd* (*SG* ! *i*))
assume *sound*:*sound* (*SG*, *C*)
obtain Γ **and** Δ **where** *SG-dec*:(Γ,Δ) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have *equivR-simp*:⋀Γ Δ *SS p q*.
(*nth* Δ *j*) = *Equiv p q* ⟹
(Γ,Δ) = *SS* ⟹
Rrule-result EquivR j SS = [(*p* # Γ, *q* # (*closeI* Δ *j*)), (*q* # Γ, *p* # (*closeI* Δ

```
j))]
    subgoal for AI SI SS p q apply(cases SS) by (auto simp add: Equiv-def
Implies-def Or-def) done
  have res-eq:Rrule-result EquivR j (SG ! i) =
    [(p # Γ, q # (closeI Δ j)), (q # Γ, p # (closeI Δ j))]
    apply(rule equivR-simp)
    subgoal using eq SG-dec by (metis snd-conv)
    by (rule SG-dec)
  have AIjeq:Δ ! j = (p ↔ q)
    using SG-dec eq snd-conv
    by metis
  have close-eq:close [(p # Γ, q # (closeI Δ j)), (q # Γ, p # (closeI Δ j))] (Γ,Δ)
= [(p # Γ, q # (closeI Δ j)), (q # Γ, p # (closeI Δ j))]
    apply(rule close-nonmember-eq)
    by (simp add: member-rec)
  have big-sound:sound ([(p # Γ, q # (closeI Δ j)), (q # Γ, p # (closeI Δ j))],
(Γ,Δ))
    apply(rule soundI')
    apply(rule seq-semI')
  proof −
    fix I ::('sf,'sc,'sz) interp and ν::'sz state
    assume good:is-interp I
   assume sgs:(⋀i. 0 ≤ i ⟹ i < length [(p # Γ, q # (closeI Δ j)), (q # Γ, p #
(closeI Δ j))] ⟹ ν ∈ seq-sem I ([(p # Γ, q # (closeI Δ j)), (q # Γ, p # (closeI
Δ j))] ! i))
    have sg1:ν ∈ seq-sem I (p # Γ, q # close Δ (Δ ! j)) using sgs[of 0] by auto
    have sg2:ν ∈ seq-sem I (q # Γ, p # (closeI Δ j)) using sgs[of 1] by auto
    assume Γ-sem:ν ∈ fml-sem I (foldr (&&) Γ TT)
    have case1:ν ∈ fml-sem I p ⟹ ν ∈ fml-sem I (foldr (||) Δ FF)
    proof −
      assume sem:ν ∈ fml-sem I p
      have ν ∈ fml-sem I (foldr (||) (q # (close Δ (nth Δ j))) FF)
        using sem Γ-sem sg1 by auto
      then show ν ∈ fml-sem I (foldr (||) Δ FF)
          using AIjeq SG-dec close-sub[of Δ nth Δ j] iff-sem[of ν I p q] jL lo-
cal.sublist-def
        member-rec(1)[of q close Δ (nth Δ j)] sem snd-conv
        or-foldl-sem-conv[of ν I q # close Δ (nth Δ j)]
        or-foldl-sem[of Δ, where I=I and ν=ν]
        nth-member[of j snd (SG ! i)]
        by metis
    qed
    have case2:ν ∉ fml-sem I p ⟹ ν ∈ fml-sem I (foldr (||) Δ FF)
    proof −
      assume sem:ν ∉ fml-sem I p
      have ν ∈ fml-sem I q ⟹  ν ∉ fml-sem I (foldr (||) Δ FF) ⟹ False
        using
          and-foldl-sem[OF Γ-sem]
          and-foldl-sem-conv
```

closeI.simps
close-sub
local.sublist-def
member-rec(1)[of p closeI Δ j]
member-rec(1)[of q Γ]
or-foldl-sem[of Δ]
or-foldl-sem-conv[of ν I p # closeI Δ j]
sem
sg2
seq-MP[of ν I q # Γ p # closeI Δ j, OF sg2]
proof −
assume *a1*: $\nu \in$ *fml-sem I q*
assume *a2*: $\nu \notin$ *fml-sem I* (*foldr* (||) Δ *FF*)
obtain *fff* :: (*'sf*, *'sc*, *'sz*) *formula* **where**
$\nu \in$ *fml-sem I fff* ∧ *List.member* (*p # close* Δ (Δ ! *j*)) *fff*
using *a1* **by** (*metis* (*no-types*) ‹⋀φ. *List.member* Γ φ ⟹ ν ∈ *fml-sem I* φ› ‹⋀*y*. *List.member* (*q # Γ*) *y* = (*q* = *y* ∨ *List.member* Γ *y*)› ‹ν ∈ *fml-sem I* (*foldr* (&&) (*q # Γ*) *TT*) ⟹ ν ∈ *fml-sem I* (*foldr* (||) (*p # closeI* Δ *j*) *FF*)› ‹ν ∈ *fml-sem I* (*foldr* (||) (*p # closeI* Δ *j*) *FF*) ⟹ ∃φ. ν ∈ *fml-sem I* φ ∧ *List.member* (*p # closeI* Δ *j*) φ› *and-foldl-sem-conv closeI.simps*)
then show *?thesis*
using *a2* **by** (*metis* (*no-types*) ‹⋀φ ν *I*. ⟦*List.member* Δ φ; ν ∈ *fml-sem I* φ⟧ ⟹ ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)› ‹⋀*y*. *List.member* (*p # closeI* Δ *j*) *y* = (*p* = *y* ∨ *List.member* (*closeI* Δ *j*) *y*)› *closeI.simps close-sub local.sublist-def sem*)
qed
show ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
by (*metis AIjeq SG-dec* ‹⟦ν ∈ *fml-sem I q*; ν ∉ *fml-sem I* (*foldr* (||) Δ *FF*)⟧ ⟹ *False*› *iff-sem jL nth-member or-foldl-sem sem snd-eqD*)
qed
show ν ∈ *fml-sem I* (*foldr* (||) Δ *FF*)
by(*cases* ν ∈ *fml-sem I p*, (*simp add*: *case1 case2*)+)
qed
show *?case*
apply(*rule close-provable-sound*)
apply(*rule sound-weaken-appR*)
apply(*rule sound*)
using *res-eq*
apply(*unfold res-eq*)
unfolding *close-app-comm*
apply (*rule sound-weaken-appL*)
using *close-eq big-sound SG-dec AIjeq*
by (*simp add*: *AIjeq*)
qed

lemma *step-sound*:*step-ok R i S* ⟹ *i* ≥ *0* ⟹ *i* < *length* (*fst R*) ⟹ *sound R* ⟹ *sound* (*step-result R* (*i*,*S*))
proof(*induction rule*: *step-ok.induct*)
case (*Step-Axiom SG i a C*)

assume *is-axiom*:*SG* ! *i* = ([], [*get-axiom a*])
assume *sound*:*sound* (*SG*, *C*)
assume *i0*:*0* ≤ *i*
assume *i* < *length* (*fst* (*SG*, *C*))
then have *iL*:*i* < *length* (*SG*)
by *auto*
have *seq-valid* ([], [*get-axiom a*])
apply(*rule fml-seq-valid*)
by(*rule axiom-valid*)
then have *seq-valid*:*seq-valid* (*SG* ! *i*)
using *is-axiom* **by** *auto*
— *i0 iL*
then show *?case*
using *closeI-valid-sound*[*OF sound seq-valid*] **by** *simp*
next
case (*Step-AxSubst SG i a σ C*)
assume *is-axiom*:*SG* ! *i* = ([], [*Fsubst* (*get-axiom a*) *σ*])
assume *sound*:*sound* (*SG*, *C*)
assume *ssafe*:*ssafe σ*
assume *i0*:*0* ≤ *i*
assume *Fadmit*:*Fadmit σ* (*get-axiom a*)
assume *i* < *length* (*fst* (*SG*, *C*))
then have *iL*:*i* < *length* (*SG*)
by *auto*
have *valid-axiom*:*valid* (*get-axiom a*)
by(*rule axiom-valid*)
have *subst-valid*:*valid* (*Fsubst* (*get-axiom a*) *σ*)
apply(*rule subst-fml-valid*)
apply(*rule Fadmit*)
apply(*rule axiom-safe*)
apply(*rule ssafe*)
by(*rule valid-axiom*)
have *seq-valid* ([], [(*Fsubst* (*get-axiom a*) *σ*)])
apply(*rule fml-seq-valid*)
by(*rule subst-valid*)
then have *seq-valid*:*seq-valid* (*SG* ! *i*)
using *is-axiom* **by** *auto*
— *i0 iL*
then show *?case*
using *closeI-valid-sound*[*OF sound seq-valid*] **by** *simp*
next
case (*Step-Lrule R i j L*)
then show *?case*
using *lrule-sound*
using *step-result.simps*(*2*) *surj-pair*
by *simp*
next
case (*Step-Rrule R i SG j L*)
then show *?case*

using *rrule-sound*
using *step-result.simps*(*2*) *surj-pair*
by *simp*
next
case (*Step-Cut* φ *i SG C*)
assume *safe*:*fsafe* φ
assume $i <$ *length* (*fst* (*SG*, *C*))
then have *iL*:$i <$ *length SG* **by** *auto*
assume *sound*:*sound* (*SG*, *C*)
obtain Γ **and** Δ **where** *SG-dec*:(Γ,Δ) = (*SG* ! *i*)
by (*metis seq2fml.cases*)
have *sound* ((φ # Γ, Δ) # (Γ, φ # Δ) # [*y*←*SG* . *y* ≠ *SG* ! *i*], *C*)
apply(*rule soundI-memv*)
proof −
fix *I*::('*sf*,'*sc*,'*sz*) *interp* **and** ν::'*sz state*
assume *good*:*is-interp I*
assume *sgs*:($\bigwedge\varphi'$ ν. *List.member* ((φ # Γ, Δ) # (Γ, φ # Δ) # [*y*←*SG* . *y* ≠ *SG* ! *i*]) φ' ⟹ ν ∈ *seq-sem I* φ')
have *sg1*:$\bigwedge\nu$. ν ∈ *seq-sem I* (φ # Γ, Δ) **using** *sgs* **by** (*meson member-rec*(*1*))
have *sg2*:$\bigwedge\nu$. ν ∈ *seq-sem I* (Γ, φ # Δ) **using** *sgs* **by** (*meson member-rec*(*1*))
have *sgs*:$\bigwedge\varphi$ ν. (*List.member* (*close SG* (*nth SG i*)) φ) ⟹ ν ∈ *seq-sem I* φ
using *sgs* **by** (*simp add*: *member-rec*(*1*))
then have *sgs*:$\bigwedge\varphi$ ν. (*List.member* (*close SG* (Γ,Δ)) φ) ⟹ ν ∈ *seq-sem I* φ
using *SG-dec* **by** *auto*
have *sgNew*:$\bigwedge\nu$. ν ∈ *seq-sem I* (Γ, Δ)
using *sg1 sg2* **by** *auto*
have *same-mem*:$\bigwedge x$. *List.member SG x* ⟹ *List.member* ((Γ,Δ) # *close SG* (Γ,Δ)) *x*
subgoal for *s*
by(*induction SG*, *auto simp add*: *member-rec*)
done
have *SGS*:($\bigwedge\varphi'$ ν. *List.member SG* φ' ⟹ ν ∈ *seq-sem I* φ')
using *sgNew sgs same-mem member-rec*(*1*) *seq-MP*
by *metis*
show ν ∈ *seq-sem I C*
using *sound* **apply** *simp*
apply(*drule soundD-memv*)
apply(*rule good*)
using *SGS*
apply *blast*
by *auto*
qed
then show *?case*
using *SG-dec case-prod-conv*
proof −
have ($\bigwedge f$. ((*case nth SG i of* (*x*, *xa*) ⇒ ((*f x xa*)::('*sf*, '*sc*, '*sz*) *rule*)) = (*f* Γ Δ)))
by (*metis* (*no-types*) *SG-dec case-prod-conv*)
then show *?thesis*

by (*simp add*: ‹*sound* ((φ # Γ, Δ) # (Γ, φ # Δ) # [y←*SG* . y ≠ *SG* ! *i*], *C*)›)
qed
next
case (*Step-G SG i C a p*)
assume *eq*:*SG* ! *i* = ([], [([[*a*]]*p*)])
assume *iL*:*i* < *length* (*fst* (*SG*, *C*))
assume *sound*:*sound* (*SG*, *C*)
have *sound* (([], [*p*]) # (*close SG* ([], [([[*a*]] *p*)])), *C*)
apply(*rule soundI-memv*)
proof −
fix *I*::('*sf*,'*sc*,'*sz*) *interp* **and** ν::'*sz state*
assume *is-interp I*
assume *sgs*:(⋀φ ν. *List.member* (([], [*p*]) # *close SG* ([], [([[*a*]]*p*)])) φ ⟹ ν ∈ *seq-sem I* φ)
have *sg0*:(⋀ν. ν ∈ *seq-sem I* ([], [*p*]))
using *sgs* **by** (*meson member-rec*(*1*))
then have *sg0'*:(⋀ν. ν ∈ *seq-sem I* ([], [([[*a*]]*p*)]))
by *auto*
have *sgTail*:(⋀φ ν. *List.member* (*close SG* ([], [([[*a*]]*p*)])) φ ⟹ ν ∈ *seq-sem I* φ)
using *sgs* **by** (*simp add*: *member-rec*(*1*))
have *same-mem*:⋀*x*. *List.member SG x* ⟹ *List.member* (([], [([[*a*]]*p*)]) # *close SG* ([], [([[*a*]]*p*)])) *x*
subgoal for *s*
by(*induction SG*, *auto simp add*: *member-rec*)
done
have *sgsC*:(⋀φ ν. *List.member SG* φ ⟹ ν ∈ *seq-sem I* φ)
apply *auto*
using *sgTail sg0'* *same-mem member-rec*
by (*metis seq-MP*)
then show ν ∈ *seq-sem I C*
using *sound*
by (*metis UNIV-eq-I* ‹*is-interp I*› *iso-tuple-UNIV-I soundD-mem*)
qed
then show *?case*
by(*auto simp add*: *eq Box-def*)
next
case (*Step-CloseId SG i j k C*)
assume *match*:*fst* (*SG* ! *i*) ! *j* = *snd* (*SG* ! *i*) ! *k*
assume *jL*:*j* < *length* (*fst* (*SG* ! *i*))
assume *kL*:*k* < *length* (*snd* (*SG* ! *i*))
assume *iL*:*i* < *length* (*fst* (*SG*, *C*))
then have *iL*:*i* < *length* (*SG*)
by *auto*
assume *sound*:*sound* (*SG*, *C*)
obtain Γ Δ **where** *SG-dec*:(Γ, Δ) = *SG* ! *i*
using *prod.collapse* **by** *blast*
have *j*Γ:*j* < *length* Γ

using *SG-dec jL*
by (*metis fst-conv*)
have $k\Delta$:$k < length\ \Delta$
using *SG-dec kL*
by (*metis snd-conv*)
have $\bigwedge I\ \nu.\ is\text{-}interp\ I \Longrightarrow \nu \in fml\text{-}sem\ I\ (foldr\ (\&\&)\ \Gamma\ TT) \Longrightarrow \nu \in fml\text{-}sem\ I\ (foldr\ (||)\ \Delta\ FF)$
proof −
fix I::$('sf,'sc,'sz)interp$ **and** ν::$'sz\ state$
assume *good*:*is-interp I*
assume Γ-*sem*:$\nu \in fml\text{-}sem\ I\ (foldr\ (\&\&)\ \Gamma\ TT)$
have *mem*:*List.member* $\Gamma\ (\Gamma\ !\ j)$
using $j\Gamma$ *nth-member* **by** *blast*
have *mem2*:*List.member* $\Delta\ (\Delta\ !\ k)$
using $k\Delta$ *nth-member* **by** *blast*
have $\nu \in fml\text{-}sem\ I\ (\Gamma\ !\ j)$
using Γ-*sem mem*
using *and-foldl-sem* **by** *blast*
then have $\nu \in fml\text{-}sem\ I\ (\Delta\ !\ k)$
using *match SG-dec*
by (*metis fst-conv snd-conv*)
then show $\nu \in fml\text{-}sem\ I\ (foldr\ (||)\ \Delta\ FF)$
using *mem2*
using *or-foldl-sem* **by** *blast*
qed
then have *seq-valid*:*seq-valid* $(SG\ !\ i)$
unfolding *seq-valid-def* **using** *SG-dec*
by (*metis UNIV-eq-I seq-semI'*)
then show *sound* $(step\text{-}result\ (SG,\ C)\ (i,\ CloseId\ j\ k))$
using *closeI-valid-sound*[*OF sound seq-valid*] **by** *simp*
next
case (*Step-DEAxiom-schema SG i ODE* σ *C*)
assume *isNth*:*nth SG i* =
([], [*Fsubst* (([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]]*P pid1*) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]][[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*)) σ])
assume *FA*:*Fadmit* σ
(([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]]*P pid1*) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]][[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*))
assume *disj*:$\{Inl\ vid1,\ Inr\ vid1\} \cap BVO\ ODE = \{\}$
have *schem-valid*:*valid* (([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]] (*P pid1*)) ↔
([[*EvolveODE* ((*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*)) (*p1 vid2 vid1*)]]
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*))
using *DE-sys-valid*[*OF disj*] **by** *auto*
assume *ssafe*:*ssafe* σ

assume *osafe*:*osafe ODE*
have *subst-valid*:*valid* (*Fsubst* ((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]]*P pid1*) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]][[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*)) *σ*)
apply(*rule subst-fml-valid*)
apply(*rule FA*)
subgoal using *disj* **by**(*auto simp add*: *f1-def Box-def p1-def P-def Equiv-def Or-def expand-singleton osafe*, *induction ODE*, *auto*)
subgoal by (*rule ssafe*)
by (*rule schem-valid*)
assume *0* ≤ *i*
assume *i* < *length* (*fst* (*SG*, *C*))
assume *sound*:*sound* (*SG*, *C*)
have *seq-valid* ([], [(*Fsubst* ((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]]*P pid1*) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]][[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*)) *σ*)])
apply(*rule fml-seq-valid*)
by(*rule subst-valid*)
then have *seq-valid*:*seq-valid* (*SG* ! *i*)
using *isNth* **by** *auto*
— *i0 iL*
then show *?case*
using *closeI-valid-sound*[*OF sound seq-valid*] **by** *simp*
next
case (*Step-CE SG i φ ψ σ C*)
assume *isNth*:*SG* ! *i* = ([], [*Fsubst* (*InContext pid1 φ* ↔ *InContext pid1 ψ*) *σ*])
assume *valid*:*valid* (*φ* ↔ *ψ*)
assume *FA*:*Fadmit σ* (*InContext pid1 φ* ↔ *InContext pid1 ψ*)
assume *0* ≤ *i*
assume *i* < *length* (*fst* (*SG*, *C*))
assume *sound*:*sound* (*SG*, *C*)
assume *fsafe1*:*fsafe φ*
assume *fsafe2*:*fsafe ψ*
assume *ssafe*:*ssafe σ*
have *schem-valid*:*valid* (*InContext pid1 φ* ↔ *InContext pid1 ψ*)
using *valid* **unfolding** *valid-def*
by (*metis CE-holds-def CE-sound fml-sem.simps*(*7*) *iff-sem surj-pair valid-def*)+
have *subst-valid*:*valid* (*Fsubst* (*InContext pid1 φ* ↔ *InContext pid1 ψ*) *σ*)
apply(*rule subst-fml-valid*)
apply(*rule FA*)
subgoal by(*auto simp add*: *f1-def Box-def p1-def P-def Equiv-def Or-def expand-singleton fsafe1 fsafe2*)
subgoal by (*rule ssafe*)
by (*rule schem-valid*)
have *seq-valid* ([], [*Fsubst* (*InContext pid1 φ* ↔ *InContext pid1 ψ*) *σ*])
apply(*rule fml-seq-valid*)
by(*rule subst-valid*)

then have *seq-valid*:*seq-valid* (*SG* ! *i*)
using *isNth* **by** *auto*
show *sound* (*step-result* (*SG*, *C*) (*i*, *CE* φ ψ σ))
using *closeI-valid-sound*[*OF sound seq-valid*] **by** *simp*
next
case (*Step-CQ SG i p* ϑ ϑ' σ *C*)
assume *isNth*:*nth SG i* = ([], [*Fsubst* (*Equiv* (*Prop p* (*singleton* ϑ)) (*Prop p* (*singleton* ϑ'))) σ])
assume *valid*:*valid* (*Equals* ϑ ϑ')
assume *FA*:*Fadmit* σ (\$$\varphi$ *p* (*singleton* ϑ) $\leftrightarrow$ \$$\varphi$ *p* (*singleton* ϑ'))
assume *0* $\leq$ *i*
assume *i* < *length* (*fst* (*SG*, *C*))
assume *sound*:*sound* (*SG*, *C*)
assume *dsafe1*:*dsafe* ϑ
assume *dsafe2*:*dsafe* ϑ'
assume *ssafe*:*ssafe* σ
have *schem-valid*:*valid* (\$$\varphi$ *p* (*singleton* ϑ) $\leftrightarrow$ \$$\varphi$ *p* (*singleton* ϑ'))
using *valid* **unfolding** *valid-def*
by (*metis CQ-holds-def CQ-sound fml-sem.simps*(7) *iff-sem surj-pair valid-def*)+
have *subst-valid*:*valid* (*Fsubst* (\$$\varphi$ *p* (*singleton* ϑ) $\leftrightarrow$ \$$\varphi$ *p* (*singleton* ϑ')) σ)
apply(*rule subst-fml-valid*)
apply(*rule FA*)
using *schem-valid ssafe* **by** (*auto simp add*: *f1-def Box-def p1-def P-def Equiv-def Or-def expand-singleton dsafe1 dsafe2 expand-singleton*)
have *seq-valid* ([], [*Fsubst* (\$$\varphi$ *p* (*singleton* ϑ) $\leftrightarrow$ \$$\varphi$ *p* (*singleton* ϑ')) σ])
apply(*rule fml-seq-valid*)
by(*rule subst-valid*)
then have *seq-valid*:*seq-valid* (*SG* ! *i*)
using *isNth* **by** *auto*
show *sound* (*step-result* (*SG*, *C*) (*i*, *CQ* ϑ ϑ' σ))
using *closeI-valid-sound*[*OF sound seq-valid*] **by** *simp*
qed

lemma *deriv-sound*:*deriv-ok R D* $\Longrightarrow$ *sound R* $\Longrightarrow$ *sound* (*deriv-result R D*)
apply(*induction rule*: *deriv-ok.induct*)
using *step-sound* **by** *auto*

lemma *proof-sound*:*proof-ok Pf* $\Longrightarrow$ *sound* (*proof-result Pf*)
apply(*induct rule*: *proof-ok.induct*)
unfolding *proof-result.simps* **apply**(*rule deriv-sound*)
apply *assumption*
by(*rule start-proof-sound*)

16 Example 1: Differential Invariants

definition *DIAndConcl*::('*sf*,'*sc*,'*sz*) *sequent*
where *DIAndConcl* = ([], [*Implies* (*And* (*Predicational pid1*) (*Predicational pid2*))

(*Implies* ([[*Pvar vid1*]](*And* (*Predicational pid3*) (*Predicational pid4*)))

([[*Pvar vid1*]](*And* (*Predicational pid1*) (*Predicational pid2*))))])

definition *DIAndSG1*::('*sf*,'*sc*,'*sz*) *formula*
where *DIAndSG1* = (*Implies* (*Predicational pid1*) (*Implies* ([[*Pvar vid1*]](*Predicational pid3*)) ([[*Pvar vid1*]](*Predicational pid1*))))

definition *DIAndSG2*::('*sf*,'*sc*,'*sz*) *formula*
where *DIAndSG2* = (*Implies* (*Predicational pid2*) (*Implies* ([[*Pvar vid1*]](*Predicational pid4*)) ([[*Pvar vid1*]](*Predicational pid2*))))

definition *DIAndCut*::('*sf*,'*sc*,'*sz*) *formula*
where *DIAndCut* =
(([[α *vid1*]]((*And* (*Predicational* (*pid3*)) (*Predicational* (*pid4*)))) → (*And* (*Predicational* (*pid1*)) (*Predicational* (*pid2*))))
→ ([[α *vid1*]](*And* (*Predicational* (*pid3*)) (*Predicational* (*pid4*)))) → ([[α *vid1*]](*And* (*Predicational* (*pid1*)) (*Predicational* (*pid2*)))))

definition *DIAndSubst*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndSubst* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*And* (*Predicational* (*Inl pid3*)) (*Predicational* (*Inl pid4*)))
else (*if C* = *pid2 then Some*(*And* (*Predicational* (*Inl pid1*)) (*Predicational* (*Inl pid2*))) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)

— [*a*]*R*&*H*−>*R*−>[*a*]*R*&*H*−>[*a*]*R DIAndSubst34*
definition *DIAndSubst341*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndSubst341* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*And* (*Predicational* (*Inl pid3*)) (*Predicational* (*Inl pid4*)))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid3*)) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)
definition *DIAndSubst342*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndSubst342* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*And* (*Predicational* (*Inl pid3*)) (*Predicational* (*Inl pid4*)))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid4*)) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)

⦈

— $[a]P$, $[a]R\&H$, P, Q |- $[a]Q{->}P\&Q{->}[a]Q{->}[a]P\&Q$, $[a]P\&Q$;;
definition *DIAndSubst12*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndSubst12* =
⦇ *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*Predicational* (*Inl pid2*))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid1*) && *Predicational* (*Inl pid2*)) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⦈

— *P* -> *Q*->*P*&*Q*
definition *DIAndCurry12*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndCurry12* =
⦇ *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*Predicational* (*Inl pid1*))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid2*) → (*Predicational* (*Inl pid1*) && *Predicational* (*Inl pid2*))) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⦈

definition *DIAnd* :: ('*sf*,'*sc*,'*sz*) *rule*
where *DIAnd* =
([([],[*DIAndSG1*]),([],[*DIAndSG2*])],
DIAndConcl)

definition *DIAndCutP1* :: ('*sf*,'*sc*,'*sz*) *formula*
where *DIAndCutP1* = ([[*Pvar vid1*]](*Predicational pid1*))

definition *DIAndCutP2* :: ('*sf*,'*sc*,'*sz*) *formula*
where *DIAndCutP2* = ([[*Pvar vid1*]](*Predicational pid2*))

definition *DIAndCutP12* :: ('*sf*,'*sc*,'*sz*) *formula*
where *DIAndCutP12* = (([[*Pvar vid1*]](*Pc pid1*) → (*Pc pid2* → (*And* (*Pc pid1*) (*Pc pid2*))))
→ (([[*Pvar vid1*]]*Pc pid1*) → ([[*Pvar vid1*]](*Pc pid2* → (*And* (*Pc pid1*) (*Pc pid2*))))))

definition *DIAndCut34Elim1* :: ('*sf*,'*sc*,'*sz*) *formula*
where *DIAndCut34Elim1* = (([[*Pvar vid1*]](*Pc pid3* && *Pc pid4*) → (*Pc pid3*))
→ (([[*Pvar vid1*]](*Pc pid3* && *Pc pid4*)) → ([[*Pvar vid1*]](*Pc pid3*))))

definition *DIAndCut34Elim2* :: ('*sf*,'*sc*,'*sz*) *formula*
where *DIAndCut34Elim2* = (([[*Pvar vid1*]](*Pc pid3* && *Pc pid4*) → (*Pc pid4*))

$\rightarrow$ ((([[*Pvar vid1*]](*Pc pid3* && *Pc pid4*)) $\rightarrow$ ([[*Pvar vid1*]](*Pc pid4*))))

definition *DIAndCut12Intro* :: (*'sf*,*'sc*,*'sz*) *formula*
where *DIAndCut12Intro* = ((([[*Pvar vid1*]](*Pc pid2* $\rightarrow$ (*Pc pid1* && *Pc pid2*)))
$\rightarrow$ ((([[*Pvar vid1*]](*Pc pid2*)) $\rightarrow$ ([[*Pvar vid1*]](*Pc pid1* && *Pc pid2*))))

definition *DIAndProof* :: (*'sf*, *'sc*, *'sz*) *pf*
where *DIAndProof* =
(*DIAndConcl*, [
(*0*, *Rrule ImplyR 0*) — 1
,(*0*, *Lrule AndL 0*)
,(*0*, *Rrule ImplyR 0*)
,(*0*, *Cut DIAndCutP1*)
,(*1*, *Cut DIAndSG1*)
,(*0*, *Rrule CohideR 0*)
,(*Suc* (*Suc 0*), *Lrule ImplyL 0*)
,(*Suc* (*Suc* (*Suc 0*)), *CloseId 1 0*)
,(*Suc* (*Suc 0*), *Lrule ImplyL 0*)
,(*Suc* (*Suc 0*), *CloseId 0 0*)
,(*Suc* (*Suc 0*), *Cut DIAndCut34Elim1*) — 11
,(*0*, *Lrule ImplyL 0*)
,(*Suc* (*Suc* (*Suc 0*)), *Lrule ImplyL 0*)
,(*0*, *Rrule CohideRR 0*)
,(*0*, *Rrule CohideRR 0*)
,(*Suc 0*, *Rrule CohideRR 0*)
,(*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))), *G*)
,(*0*, *Rrule ImplyR 0*)
,(*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))), *Lrule AndL 0*)
,(*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))), *CloseId 0 0*)
,(*Suc* (*Suc* (*Suc 0*)), *AxSubst AK DIAndSubst341*) — 21
,(*Suc* (*Suc 0*), *CloseId 0 0*)
,(*Suc 0*, *CloseId 0 0*)
,(*0*, *Cut DIAndCut12Intro*)
,(*Suc 0*, *Rrule CohideRR 0*)
,(*Suc* (*Suc 0*), *AxSubst AK DIAndSubst12*)
,(*0*, *Lrule ImplyL 0*)
,(*1*, *Lrule ImplyL 0*)
,(*Suc* (*Suc 0*), *CloseId 0 0*)
,(*Suc 0*, *Cut DIAndCutP12*)
,(*0*, *Lrule ImplyL 0*) — 31
,(*0*, *Rrule CohideRR 0*)
,(*Suc* (*Suc* (*Suc* (*Suc 0*))), *AxSubst AK DIAndCurry12*)
,(*Suc* (*Suc* (*Suc 0*)), *Rrule CohideRR 0*)
,(*Suc* (*Suc 0*), *Lrule ImplyL 0*)
,(*Suc* (*Suc 0*), *G*)
,(*0*, *Rrule ImplyR 0*)
,(*Suc* (*Suc* (*Suc* (*Suc 0*))), *Rrule ImplyR 0*)
,(*Suc* (*Suc* (*Suc* (*Suc 0*))), *Rrule AndR 0*)
,(*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))), *CloseId 0 0*)

```
  ,(Suc (Suc (Suc (Suc 0))), CloseId 1 0) — 41
  ,(Suc (Suc  0), CloseId 0 0)
  ,(Suc 0, Cut DIAndCut34Elim2)
  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(Suc (Suc (Suc (Suc 0))), AxSubst AK DIAndSubst342) — 46
  ,(Suc (Suc (Suc 0)), Rrule CohideRR 0)
  ,(Suc (Suc (Suc 0)), G) — 48
  ,(0, Rrule ImplyR 0)
  ,(Suc (Suc (Suc 0)), Lrule AndL 0) — 50
  ,(Suc (Suc (Suc 0)), CloseId 1 0)
  ,(Suc (Suc 0), Lrule ImplyL 0)
  ,(Suc 0, CloseId 0 0)
  ,(1, Cut DIAndSG2)
  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(Suc (Suc (Suc 0)), CloseId 4 0)
  ,(Suc (Suc 0), Lrule ImplyL 0)
  ,(Suc (Suc (Suc 0)), CloseId 0 0)
  ,(Suc (Suc (Suc 0)), CloseId 0 0)
  ,(1, CloseId 1 0)
  ])


fun proof-take :: nat ⇒ ('sf,'sc,'sz) pf ⇒ ('sf,'sc,'sz) pf
where proof-take n (C,D) = (C,List.take n D)

fun last-step::('sf,'sc,'sz) pf ⇒ nat ⇒ nat * ('sf,'sc,'sz ) step
where last-step (C,D) n = List.last (take n D)

lemma DIAndSound-lemma:sound (proof-result (proof-take 61 DIAndProof))
  apply(rule proof-sound)
   unfolding DIAndProof-def DIAndConcl-def   DIAndCutP1-def DIAndSG1-def
DIAndCut34Elim1-def DIAndSubst341-def DIAndCut12Intro-def DIAndSubst12-def
    DIAndCutP12-def DIAndCurry12-def DIAndSubst342-def
    DIAndCut34Elim2-def — 43
    DIAndSG2-def — 54
  apply (auto simp add: prover)
  done
```

17 Example 2: Concrete Hybrid System

```
definition SystemConcl::('sf,'sc,'sz) sequent
where SystemConcl =
  ([],[
  Implies (And (Geq (Var vid1) (Const 0)) (Geq (f0 fid1) (Const 0)))
  ([[EvolveODE (OProd (OSing vid1 (f0 fid1)) (OSing vid2 (Var vid1))) (TT)]]Geq
(Var vid1) (Const 0))
  ])
```

definition *SystemDICut* :: (*'sf*,*'sc*,*'sz*) *formula*
where *SystemDICut* =
Implies
(*Implies TT* ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]]
(*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))
(*Implies*
(*Implies TT* (*Geq* (*Var vid1*) (*Const 0*)))
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]](*Geq* (*Var vid1*) (*Const 0*))))

definition *SystemDCCut*::(*'sf*,*'sc*,*'sz*) *formula*
where *SystemDCCut* =
(([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]](*Geq* (*f0 fid1*) (*Const 0*))) →
(([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]]((*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))
↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))))

definition *SystemVCut*::(*'sf*,*'sc*,*'sz*) *formula*
where *SystemVCut* =
Implies (*Geq* (*f0 fid1*) (*Const 0*)) ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]](*Geq* (*f0 fid1*) (*Const 0*)))

definition *SystemVCut2*::(*'sf*,*'sc*,*'sz*) *formula*
where *SystemVCut2* =
Implies (*Geq* (*f0 fid1*) (*Const 0*)) ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]](*Geq* (*f0 fid1*) (*Const 0*)))

definition *SystemDECut*::(*'sf*,*'sc*,*'sz*) *formula*
where *SystemDECut* = (([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ((*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]]
[[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))

definition *SystemKCut*::(*'sf*,*'sc*,*'sz*) *formula*
where *SystemKCut* =
(*Implies* ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]]
(*Implies* ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))) ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))))

(*Implies* ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*))))) ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))))

definition *SystemEquivCut*::('*sf*,'*sc*,'*sz*) *formula*
where *SystemEquivCut* =
(*Equiv* (*Implies* ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))) ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))) (*Implies* ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))) ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*DiffVar vid1*) (*Const 0*)))))

definition *SystemDiffAssignCut*::('*sf*,'*sc*,'*sz*) *formula*
where *SystemDiffAssignCut* =
(([[*DiffAssign vid1* ($*f fid1 empty*)]] (*Geq* (*DiffVar vid1*) (*Const 0*))) ↔ (*Geq* ($*f fid1 empty*) (*Const 0*)))

definition *SystemCEFml1*::('*sf*,'*sc*,'*sz*) *formula*
where *SystemCEFml1* = *Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))

definition *SystemCEFml2*::('*sf*,'*sc*,'*sz*) *formula*
where *SystemCEFml2* = *Geq* (*DiffVar vid1*) (*Const 0*)

definition *CQ1Concl*::('*sf*,'*sc*,'*sz*) *formula*
where *CQ1Concl* = (*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)) ↔ *Geq* (*DiffVar vid1*) (*Differential* (*Const 0*)))

definition *CQ2Concl*::('*sf*,'*sc*,'*sz*) *formula*
where *CQ2Concl* = (*Geq* (*DiffVar vid1*) (*Differential* (*Const 0*)) ↔ *Geq* ($' *vid1*) (*Const 0*))

definition *CEReq*::('*sf*,'*sc*,'*sz*) *formula*
where *CEReq* = (*Geq* (*Differential* (*trm.Var vid1*)) (*Differential* (*Const 0*)) ↔ *Geq* ($' *vid1*) (*Const 0*))

definition *CQRightSubst*::('*sf*,'*sc*,'*sz*) *subst*
where *CQRightSubst* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ*p*. (*if p* = *vid1 then* (*Some* (*Geq* (*DiffVar vid1*) (*Function* (*Inr vid1*) *empty*))) *else None*)),
SContexts = (λ-. *None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)

definition *CQLeftSubst*::('*sf*,'*sc*,'*sz*) *subst*
where *CQLeftSubst* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ*p*. (*if p* = *vid1 then* (*Some* (*Geq* (*Function* (*Inr vid1*) *empty*)
(*Differential* (*Const 0*)))) *else None*)),
SContexts = (λ-. *None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)

definition *CEProof*::('*sf*,'*sc*,'*sz*) *pf*
where *CEProof* = (([],[*CEReq*]), [
(*0*, *Cut CQ1Concl*)
,(*0*, *Cut CQ2Concl*)
,(*1*, *Rrule CohideRR 0*)
,(*Suc* (*Suc 0*), *CQ* (*Differential* (*Const 0*)) (*Const 0*) *CQRightSubst*)
,(*1*, *Rrule CohideRR 0*)
,(*1*, *CQ* (*Differential* (*Var vid1*)) (*DiffVar vid1*) *CQLeftSubst*)
,(*0*, *Rrule EquivR 0*)
,(*0*, *Lrule EquivForwardL 1*)
,(*Suc* (*Suc 0*), *Lrule EquivForwardL 1*)
,(*Suc* (*Suc* (*Suc 0*)), *CloseId 0 0*)
,(*Suc* (*Suc 0*), *CloseId 0 0*)
,(*Suc 0*, *CloseId 0 0*)
,(*0*, *Lrule EquivBackwardL* (*Suc* (*Suc 0*)))
,(*0*, *CloseId 0 0*)
,(*0*, *Lrule EquivBackwardL* (*Suc 0*))
,(*0*, *CloseId 0 0*)
,(*0*, *CloseId 0 0*)
])

lemma *CE-result-correct*:*proof-result CEProof* = ([],([],[*CEReq*]))
unfolding *CEProof-def CEReq-def CQ1Concl-def CQ2Concl-def Implies-def Or-def f0-def TT-def Equiv-def Box-def CQRightSubst-def*
by (*auto simp add*: *id-simps*)

definition *DiffConstSubst*::('*sf*,'*sc*,'*sz*) *subst*
where *DiffConstSubst* = (|
SFunctions = (λ*f*. (*if f* = *fid1 then* (*Some* (*Const 0*)) *else None*)),
SPredicates = (λ-. *None*),
SContexts = (λ-. *None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)

definition *DiffConstProof*::('*sf*,'*sc*,'*sz*) *pf*
where *DiffConstProof* = (([],[(*Equals* (*Differential* (*Const 0*)) (*Const 0*))]), [

(*0*, *AxSubst AdConst DiffConstSubst*)])

lemma *diffconst-result-correct*:*proof-result DiffConstProof* = ([], ([],[*Equals* (*Differential* (*Const 0*)) (*Const 0*)]))
by(*auto simp add*: *prover DiffConstProof-def*)

lemma *diffconst-sound-lemma*:*sound* (*proof-result DiffConstProof*)
apply(*rule proof-sound*)
unfolding *DiffConstProof-def*
by (*auto simp add*: *prover DiffConstProof-def DiffConstSubst-def Equals-def empty-def TUadmit-def*)

lemma *valid-of-sound*:*sound* ([], ([],[φ])) $\Longrightarrow$ *valid* φ
unfolding *valid-def sound-def TT-def FF-def*
apply (*auto simp add*: *TT-def FF-def Or-def*)
subgoal for *I a b*
apply(*erule allE*[**where** *x*=*I*])
by(*auto*)
done

lemma *almost-diff-const-sound*:*sound* ([], ([], [*Equals* (*Differential* (*Const 0*)) (*Const 0*)]))
using *diffconst-result-correct diffconst-sound-lemma* **by** *simp*

lemma *almost-diff-const*:*valid* (*Equals* (*Differential* (*Const 0*)) (*Const 0*))
using *almost-diff-const-sound valid-of-sound* **by** *auto*

— Note: this is just unpacking the definition: the axiom is defined as literally this formula
lemma *almost-diff-var*:*valid* (*Equals* (*Differential* (*trm.Var vid1*)) ($' *vid1*))
using *diff-var-axiom-valid* **unfolding** *diff-var-axiom-def* **by** *auto*

lemma *CESound-lemma*:*sound* (*proof-result CEProof*)
apply(*rule proof-sound*)
unfolding *CEProof-def CEReq-def CQ1Concl-def CQ2Concl-def Equiv-def CQRight-Subst-def diff-const-axiom-valid diff-var-axiom-valid empty-def Or-def expand-singleton*

diff-var-axiom-def
by (*auto simp add*: *prover CEProof-def CEReq-def CQ1Concl-def CQ2Concl-def Equiv-def*
CQRightSubst-def diff-const-axiom-valid diff-var-axiom-valid empty-def Or-def expand-singleton
TUadmit-def NTUadmit-def almost-diff-const CQLeftSubst-def almost-diff-var)

lemma *sound-to-valid*:*sound* ([], ([], [φ])) $\Longrightarrow$ *valid* φ
unfolding *valid-def* **apply** *auto*
apply(*drule soundD-mem*)
by (*auto simp add*: *member-rec*(*2*))

lemma *CE1pre:sound* ([], ([], [*CEReq*]))
using *CE-result-correct CESound-lemma*
by *simp*

lemma *CE1pre-valid:valid CEReq*
by (*rule sound-to-valid*[*OF CE1pre*])

lemma *CE1pre-valid2:valid* (! (! (*Geq* (*Differential* (*trm.Var vid1*)) (*Differential* (*Const 0*)) && *Geq* ($' *vid1*) (*Const 0*)) &&
! (! (*Geq* (*Differential* (*trm.Var vid1*)) (*Differential* (*Const 0*))) && ! (*Geq* ($' *vid1*) (*Const 0*)))))
using *CE1pre-valid* **unfolding** *CEReq-def Equiv-def Or-def* **by** *auto*

definition *SystemDISubst*::('*sf*,'*sc*,'*sz*) *subst*
where *SystemDISubst* =
(| *SFunctions* = (λ*f*.
(if *f* = *fid1* then *Some*(*Function* (*Inr vid1*) *empty*)
else if *f* = *fid2* then *Some*(*Const 0*)
else *None*)),
SPredicates = (λ*p*. if *p* = *vid1* then *Some TT* else *None*),
SContexts = (λ-. *None*),
SPrograms = (λ-. *None*),
SODEs = (λ*c*. if *c* = *vid1* then *Some* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*trm.Var vid1*))) else *None*)
|)

definition *SystemDCSubst*::('*sf*,'*sc*,'*sz*) *subst*
where *SystemDCSubst* =
(| *SFunctions* = (λ
f. *None*),
SPredicates = (λ*p*. *None*),
SContexts = (λ*C*.
if *C* = *pid1* then
Some TT
else if *C* = *pid2* then
Some (*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))
else if *C* = *pid3* then
Some (*Geq* (*Function fid1 empty*) (*Const 0*))
else
None),
SPrograms = (λ-. *None*),
SODEs = (λ*c*. if *c* = *vid1* then *Some* (*OProd* (*OSing vid1* (*Function fid1 empty*)) (*OSing vid2* (*trm.Var vid1*))) else *None*)
|)

definition *SystemVSubst*::('*sf*,'*sc*,'*sz*) *subst*

where *SystemVSubst* =
⦇ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ*p*. *if p* = *vid1 then Some* (*Geq* (*Function* (*Inl fid1*) *empty*) (*Const 0*)) *else None*),
SContexts = (λ-. *None*),
SPrograms = (λ*a*. *if a* = *vid1 then*
Some (*EvolveODE* (*OProd*
(*OSing vid1* (*Function fid1 empty*))
(*OSing vid2* (*Var vid1*)))
(*And TT* (*Geq* (*Function fid1 empty*) (*Const 0*))))
else None),
SODEs = (λ-. *None*)
⦈

definition *SystemVSubst2*::(*'sf*,*'sc*,*'sz*) *subst*
where *SystemVSubst2* =
⦇ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ*p*. *if p* = *vid1 then Some* (*Geq* (*Function* (*Inl fid1*) *empty*) (*Const 0*)) *else None*),
SContexts = (λ-. *None*),
SPrograms = (λ*a*. *if a* = *vid1 then*
Some (*EvolveODE* (*OProd*
(*OSing vid1* (*Function fid1 empty*))
(*OSing vid2* (*Var vid1*)))
TT)
else None),
SODEs = (λ-. *None*)
⦈

definition *SystemDESubst*::(*'sf*,*'sc*,*'sz*) *subst*
where *SystemDESubst* =
⦇ *SFunctions* = (λ*f*. *if f* = *fid1 then Some*(*Function* (*Inl fid1*) *empty*) *else None*),
SPredicates = (λ*p*. *if p* = *vid2 then Some*(*And TT* (*Geq* (*Function* (*Inl fid1*) *empty*) (*Const 0*))) *else None*),
SContexts = (λ*C*. *if C* = *pid1 then Some*(*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))) *else None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⦈

lemma *systemdesubst-correct*:∃ *ODE*.(([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ((*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]]
[[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))
= *Fsubst* ((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*)) *ODE*) (*p1 vid2 vid1*)]] (*P pid1*)) ↔

```
      ([[EvolveODE ((OProd (OSing vid1 (f1 fid1 vid1))) ODE) (p1 vid2 vid1)]]
          [[DiffAssign vid1 (f1 fid1 vid1)]]P pid1))) SystemDESubst
  apply(rule exI[where x=OSing vid2 (trm.Var vid1)])
  by(auto simp add: f0-def f1-def Box-def Or-def Equiv-def empty-def TT-def P-def
p1-def SystemDESubst-def empty-def)

— [{dx=, dy=x&r>=r&>=r}]r>=r&>=r−>[D{x}:=]D{x}>=D{r}−>
— [{dx=, dy=x&r>=r&>=r}]r>=r&>=r−>
— [{dx=, dy=x&r>=r&>=r}][D{x}:=]D{x}>=D{r}
— ([[$α vid1]]((Predicational pid1) → (Predicational pid2)))
— → ([[$α vid1]]Predicational pid1) → ([[$α vid1]]Predicational pid2)
definition SystemKSubst::('sf,'sc,'sz) subst
where SystemKSubst = (| SFunctions = (λf. None),
    SPredicates = (λ-. None),
    SContexts = (λC. if C = pid1 then
       (Some (And (Geq (Const 0) (Const 0)) (Geq (Function fid1 empty) (Const
0))))
      else if C = pid2 then
         (Some ([[DiffAssign vid1 (Function fid1 empty)]](Geq (Differential (Var
vid1)) (Differential (Const 0))))) else None),
     SPrograms = (λc. if c = vid1 then Some (EvolveODE (OProd (OSing vid1
(Function fid1 empty)) (OSing vid2 (Var vid1))) (And (Geq (Const 0) (Const 0))
(Geq (Function fid1 empty) (Const 0)))) else None),
    SODEs = (λ-. None)
  |)

lemma subst-imp-simp:Fsubst (Implies p q) σ = (Implies (Fsubst p σ) (Fsubst q
σ))
  unfolding Implies-def Or-def by auto

lemma subst-equiv-simp:Fsubst (Equiv p q) σ = (Equiv (Fsubst p σ) (Fsubst q σ))
  unfolding Implies-def Or-def Equiv-def by auto

lemma subst-box-simp:Fsubst (Box p q) σ = (Box (Psubst p σ) (Fsubst q σ))
  unfolding Box-def Or-def by auto

lemma pfsubst-box-simp:PFsubst (Box p q) σ = (Box (PPsubst p σ) (PFsubst q
σ))
  unfolding Box-def Or-def by auto

lemma pfsubst-imp-simp:PFsubst (Implies p q) σ = (Implies (PFsubst p σ) (PFsubst
q σ))
  unfolding Box-def Implies-def Or-def by auto

definition SystemDWSubst::('sf,'sc,'sz) subst
where SystemDWSubst = (| SFunctions = (λf. None),
    SPredicates = (λ-. None),
     SContexts = (λC. if C = pid1 then Some (And (Geq (Const 0) (Const 0))
(Geq (Function fid1 empty) (Const 0))) else None),
```

SPrograms = (λ-. *None*),
SODEs = (λ*c*. *if c* = *vid1 then Some* (*OProd* (*OSing vid1* (*Function fid1 empty*)) (*OSing vid2* (*Var vid1*))) *else None*)
⟫

definition *SystemCESubst*::('*sf*,'*sc*,'*sz*) *subst*
where *SystemCESubst* = ⟪ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. *if C* = *pid1 then Some*(*Implies*(*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*))) ([[*DiffAssign vid1* (*Function fid1 empty*)]](*Predicational* (*Inr* ()))))) *else None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⟫

lemma *SystemCESubstOK*:
step-ok
([(],[*Equiv* (*Implies*(*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*))) ([[*DiffAssign vid1* (*Function fid1 empty*)]](*SystemCEFml1*)))
(*Implies*(*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*))) ([[*DiffAssign vid1* (*Function fid1 empty*)]]((*SystemCEFml2*))))
])],
([],[]))

0
(*CE SystemCEFml1 SystemCEFml2 SystemCESubst*)
apply(*rule Step-CE*)
subgoal by(*auto simp add*: *subst-equiv-simp subst-imp-simp subst-box-simp SystemCESubst-def SystemCEFml1-def SystemCEFml2-def pfsubst-imp-simp pfsubst-box-simp*)
subgoal using *CE1pre-valid*
by (*auto simp add*: *CEReq-def SystemCEFml1-def SystemCEFml2-def CE1pre-valid*)
subgoal unfolding *SystemCEFml1-def* **by** *auto*
subgoal unfolding *SystemCEFml2-def* **by** *auto*
subgoal unfolding *SystemCESubst-def ssafe-def Implies-def Box-def Or-def empty-def* **by** *auto*
unfolding *SystemCESubst-def Equiv-def Or-def SystemCEFml1-def SystemCEFml2-def TUadmit-def* **apply** (*auto simp add*: *TUadmit-def FUadmit-def Box-def Implies-def Or-def*)
unfolding *PFUadmit-def* **by** *auto*

— [*D*{*x*}:=*f*]*Dv*{*x*}>=*r*<->*f*>=*r*
— [[*DiffAssign vid1* (*$f fid1 empty*)]] (*Prop vid1* (*singleton* (*DiffVar vid1*))))
— ↔ *Prop vid1* (*singleton* (*$f fid1 empty*))
definition *SystemDiffAssignSubst*::('*sf*,'*sc*,'*sz*) *subst*
where *SystemDiffAssignSubst* = ⟪ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ*p*. *if p* = *vid1 then Some* (*Geq* (*Function* (*Inr vid1*) *empty*) (*Const 0*)) *else None*),
SContexts = (λ-. *None*),

```
  SPrograms = (λ-. None),
  SODEs = (λ-. None)
|)

lemma SystemDICutCorrect:SystemDICut = Fsubst DIGeqaxiom SystemDISubst
  unfolding SystemDICut-def DIGeqaxiom-def SystemDISubst-def
  by (auto simp add: f1-def p1-def f0-def Implies-def Or-def id-simps TT-def Box-def
empty-def)

— v≥0 ∧ A()≥0 → [{x'=v, v'=A()}]v≥0
definition SystemProof :: ('sf, 'sc, 'sz) pf
where SystemProof =
  (SystemConcl, [
  (0, Rrule ImplyR 0)
  ,(0, Lrule AndL 0)
  ,(0, Cut SystemDICut)
  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(0, Lrule ImplyL 0)
  ,(Suc (Suc 0), CloseId 0 0)
  ,(Suc 0, AxSubst ADIGeq SystemDISubst) — 8
  ,(Suc 0, Rrule ImplyR 0)
  ,(0, CloseId 1 0)
  ,(Suc 0, CloseId 1 0)
  ,(0, Rrule AndR 0)
  ,(0, Rrule ImplyR 0)
  ,(0, Cut SystemDCCut)
  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(0, Lrule EquivBackwardL 0)
  ,(0, Rrule CohideR 0)
  ,(0, AxSubst ADC SystemDCSubst) — 17
  ,(0, CloseId 0 0)
  ,(0, Rrule CohideRR 0)
  ,(0, Cut SystemVCut)
  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(0, Cut SystemDECut)
  ,(0, Lrule EquivBackwardL 0)
  ,(0, Rrule CohideRR 0)
  ,(1, CloseId (Suc 1) 0) — Last step
  ,(Suc 1, CloseId 0 0)
  ,(1, AxSubst AV SystemVSubst) — 28
  ,(0, Cut SystemVCut2)

  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(Suc 1, CloseId 0 0)
  ,(Suc 1, CloseId (Suc 2) 0)
```

```
  ,(Suc 1, AxSubst AV SystemVSubst2) — 34
  ,(0, Rrule CohideRR 0)
  ,(0, DEAxiomSchema (OSing vid2 (trm.Var vid1)) SystemDESubst) — 36
  ,(0, Cut SystemKCut)
  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(0, Lrule ImplyL 0)
  ,(0, Rrule CohideRR 0)
  ,(0, AxSubst AK SystemKSubst) — 42
  ,(0, CloseId 0 0)
  ,(0, Rrule CohideR 0)
  ,(1, AxSubst ADW SystemDWSubst) — 45
  ,(0, G)
  ,(0, Cut SystemEquivCut)
  ,(0, Lrule EquivBackwardL 0)
  ,(0, Rrule CohideR 0)
  ,(0, CloseId 0 0)
  ,(0, Rrule CohideR 0)
  ,(0, CE SystemCEFml1 SystemCEFml2 SystemCESubst) — 52
  ,(0, Rrule ImplyR 0)
  ,(0, Lrule AndL 0)
  ,(0, Cut SystemDiffAssignCut)
  ,(0, Lrule EquivBackwardL 0)
  ,(0, Rrule CohideRR 0)
  ,(0, CloseId 0 0)
  ,(0, CloseId 1 0)
  ,(0, AxSubst Adassign SystemDiffAssignSubst) — 60
  ])

lemma system-result-correct:proof-result SystemProof =
  ([],
  ([],[Implies (And (Geq (Var vid1) (Const 0)) (Geq (f0 fid1) (Const 0)))
          ([[EvolveODE (OProd (OSing vid1 (f0 fid1)) (OSing vid2 (Var vid1)))
(TT)]]Geq (Var vid1) (Const 0))]))
   unfolding SystemProof-def SystemConcl-def Implies-def Or-def f0-def TT-def
Equiv-def SystemDICut-def SystemDCCut-def
  proof-result.simps deriv-result.simps start-proof.simps Box-def SystemDCSubst-def
SystemVCut-def SystemDECut-def SystemKCut-def SystemEquivCut-def
  SystemDiffAssignCut-def SystemVCut2-def

  apply( simp add: prover)
  done

lemma SystemSound-lemma:sound (proof-result SystemProof)
  apply(rule proof-sound)
 unfolding SystemProof-def SystemConcl-def CQ1Concl-def CQ2Concl-def Equiv-def
CQRightSubst-def diff-const-axiom-valid diff-var-axiom-valid empty-def Or-def ex-
pand-singleton
```

diff-var-axiom-def SystemDICut-def

apply (*auto simp add*: *prover CEProof-def CEReq-def CQ1Concl-def CQ2Concl-def Equiv-def*
CQRightSubst-def diff-const-axiom-valid diff-var-axiom-valid empty-def Or-def expand-singleton
TUadmit-def NTUadmit-def almost-diff-const CQLeftSubst-def almost-diff-var f0-def TT-def SystemDISubst-def f1-def p1-def SystemDCCut-def SystemDCSubst-def
SystemVCut-def SystemDECut-def SystemVSubst-def
SystemVCut2-def SystemVSubst2-def SystemDESubst-def P-def SystemKCut-def SystemKSubst-def SystemDWSubst-def SystemEquivCut-def
SystemCESubst-def SystemCEFml1-def SystemCEFml2-def CE1pre-valid2 SystemDiffAssignCut-def SystemDiffAssignSubst-def)
done

lemma *system-sound*:*sound* ([], *SystemConcl*)
using *SystemSound-lemma system-result-correct* **unfolding** *SystemConcl-def* **by** *auto*

lemma *DIAnd-result-correct*:*proof-result* (*proof-take 61 DIAndProof*) = *DIAnd*
unfolding *DIAndProof-def DIAndConcl-def Implies-def Or-def*
proof-result.simps deriv-result.simps start-proof.simps DIAndCutP12-def DIAndSG1-def DIAndSG2-def DIAndCutP1-def Box-def DIAndCut34Elim1-def DIAndCut12Intro-def DIAndCut34Elim2-def DIAnd-def
using *pne12 pne13 pne14 pne23 pne24 pne34* **by** (*auto*)

theorem *DIAnd-sound*: *sound DIAnd*
using *DIAndSound-lemma DIAnd-result-correct* **by** *auto*

end end

18 dL Formalization

theory *Differential-Dynamic-Logic*
imports
Complex-Main
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
Frechet-Correctness
Static-Semantics
Coincidence
Bound-Effect
Axioms
Differential-Axioms
USubst
USubst-Lemma

Uniform-Renaming
Proof-Checker
begin
end

References

[1] R. Bohrer, V. Rahli, I. Vukotic, M. Völp, and A. Platzer. Formally verified differential dynamic logic. In Y. Bertot and V. Vafeiadis, editors, *Certified Programs and Proofs - 6th ACM SIGPLAN Conference, CPP 2017, Paris, France, January 16-17, 2017*, pages 208–221. ACM, 2017.

[2] A. Platzer. A uniform substitution calculus for differential dynamic logic. In A. P. Felty and A. Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 467–481. Springer, 2015.

[3] A. Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, 2016.