

# Differential-Dynamic-Logic

Brandon Bohrer

February 23, 2021

## 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

<b>1</b>	<b>Identifier locale</b>	<b>3</b>
<b>2</b>	<b>Generic Mathematical Lemmas</b>	<b>4</b>
<b>3</b>	<b>Syntax</b>	<b>6</b>
3.1	Well-Formedness predicates . . . . .	10
3.2	Denotational Semantics . . . . .	14
3.3	States . . . . .	14
3.4	Interpretations . . . . .	16
3.5	Trivial Simplification Lemmas . . . . .	21

<b>4</b>	<b>Axioms</b>	<b>24</b>
4.1	Validity proofs for axioms . . . . .	25
4.2	Soundness proofs for rules . . . . .	27
<b>5</b>	<b>Characterization of Term Derivatives</b>	<b>28</b>
<b>6</b>	<b>Static Semantics</b>	<b>31</b>
6.1	Signature Definitions . . . . .	31
6.2	Variable Binding Definitions . . . . .	32
6.3	Lemmas for reasoning about static semantics . . . . .	34
<b>7</b>	<b>Coincidence Theorems and Corollaries</b>	<b>35</b>
7.1	Term Coincidence Theorems . . . . .	35
7.2	ODE Coincidence Theorems . . . . .	36
7.3	Coincidence Theorems for Programs and Formulas . . . . .	38
7.4	Corollaries: Alternate ODE semantics definition . . . . .	38
<b>8</b>	<b>Bound Effect Theorem</b>	<b>39</b>
<b>9</b>	<b>Differential Axioms</b>	<b>39</b>
9.1	Derivative Axioms . . . . .	40
9.2	ODE Axioms . . . . .	40
9.3	Proofs for Derivative Axioms . . . . .	42
9.4	Proofs for ODE Axioms . . . . .	42
<b>10</b>	<b>Uniform Substitution Definitions</b>	<b>45</b>
<b>11</b>	<b>Soundness proof for uniform substitution rule</b>	<b>56</b>
11.1	Lemmas about well-formedness of (adjoint) interpretations. . . . .	57
11.2	Lemmas about adjoint interpretations . . . . .	58
11.3	Substitution theorems for terms . . . . .	64
11.4	Substitution theorems for ODEs . . . . .	65
11.5	Substitution theorems for formulas and programs . . . . .	67
11.6	Soundness of substitution rule . . . . .	70
<b>12</b>	<b>Uniform and Bound Renaming</b>	<b>70</b>
12.1	Uniform Renaming Definitions . . . . .	70
12.2	Uniform Renaming Admissibility . . . . .	71
12.3	Uniform Renaming Soundness Proof and Lemmas . . . . .	72
12.4	Uniform Renaming Rule Soundness . . . . .	75
12.5	Bound Renaming Rule Soundness . . . . .	75
<b>13</b>	<b>Syntax Pretty-Printer</b>	<b>75</b>
<b>14</b>	<b>Proof Checker</b>	<b>77</b>

<b>15 Proof Checker Implementation</b>	<b>78</b>
15.1 Soundness . . . . .	83
<b>16 Example 1: Differential Invariants</b>	<b>85</b>
<b>17 Example 2: Concrete Hybrid System</b>	<b>89</b>
<b>18 dL Formalization</b>	<b>98</b>
theory <i>Ids</i>	
imports <i>Complex-Main</i>	
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
⟨proof⟩
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*:  $(\bigwedge i. v\$i = w\$i) \implies (v = w)$   
 ⟨proof⟩

**lemma** *norm-axis*:  $\text{norm } (\text{axis } i \ x) = \text{norm } x$   
 ⟨proof⟩

**lemma** *bounded-linear-axis*:  $\text{bounded-linear } (\text{axis } i)$   
 ⟨proof⟩

**lemma** *bounded-linear-vec*:  
**fixes**  $f::('a::\text{finite}) \Rightarrow 'b::\text{real-normed-vector} \Rightarrow 'c::\text{real-normed-vector}$   
**assumes**  $\text{bounds}:\bigwedge i. \text{bounded-linear } (f \ i)$   
**shows**  $\text{bounded-linear } (\lambda x. \chi \ i. f \ i \ x)$   
 ⟨proof⟩

**lift-definition** *blinfun-vec*:  $('a::\text{finite}) \Rightarrow 'b::\text{real-normed-vector} \Rightarrow_L \text{real}) \Rightarrow 'b \Rightarrow_L$   
 $(\text{real} \wedge 'a)$  **is**  $(\lambda(f::('a \Rightarrow 'b \Rightarrow \text{real})) (x::'b). \chi (i::'a). f \ i \ x)$   
 ⟨proof⟩

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

**lemma** *continuous-blinfun-vec*:  $(\bigwedge i. \text{continuous-on UNIV } (\text{blinfun-apply } (g \ i))) \implies$   
 $\text{continuous-on UNIV } (\text{blinfun-vec } g)$   
 ⟨proof⟩

**lemma** *blinfun-elim*:  $\bigwedge g. (\text{blinfun-apply } (\text{blinfun-vec } g)) = (\lambda x. \chi \ i. g \ i \ x)$   
 ⟨proof⟩

**lemma** *sup-plus*:  
**fixes**  $f \ g::('b::\text{metric-space}) \Rightarrow \text{real}$   
**assumes**  $\text{nonempty}: R \neq \{\}$   
**assumes**  $\text{bddf}: \text{bdd-above } (f \ 'R)$

**assumes** *bddg*:*bdd-above* ( $g \text{ ' } R$ )  
**shows**  $(\text{SUP } x \in R. f x + g x) \leq (\text{SUP } x \in R. f x) + (\text{SUP } x \in R. g x)$   
 $\langle \text{proof} \rangle$

**lemma** *continuous-blinfun-vec'*:  
**fixes**  $f::'a::\{\text{finite}, \text{linorder}\} \Rightarrow 'b::\{\text{metric-space}, \text{real-normed-vector}, \text{abs}\} \Rightarrow 'b$   
 $\Rightarrow_L \text{real}$   
**fixes**  $S::'b \text{ set}$   
**assumes**  $\text{conts}:\bigwedge i. \text{continuous-on UNIV } (f i)$   
**shows** *continuous-on UNIV*  $(\lambda x. \text{blinfun-vec } (\lambda i. f i x))$   
 $\langle \text{proof} \rangle$

**lemma** *has-derivative-vec*[*derivative-intros*]:  
**assumes**  $\bigwedge i. ((\lambda x. f i x) \text{ has-derivative } (\lambda h. f' i h)) F$   
**shows**  $((\lambda x. \chi i. f i x) \text{ has-derivative } (\lambda h. \chi i. f' i h)) F$   
 $\langle \text{proof} \rangle$

**lemma** *has-derivative-proj*:  
**fixes**  $j::('a::\text{finite})$   
**fixes**  $f::'a \Rightarrow \text{real} \Rightarrow \text{real}$   
**assumes**  $\text{asm}::((\lambda x. \chi i. f i x) \text{ has-derivative } (\lambda h. \chi i. f' i h)) F$   
**shows**  $((\lambda x. f j x) \text{ has-derivative } (\lambda h. f' j h)) F$   
 $\langle \text{proof} \rangle$

**lemma** *has-derivative-proj'*:  
**fixes**  $i::'a::\text{finite}$   
**shows**  $\forall x. ((\lambda x. x \$ i) \text{ has-derivative } (\lambda x::(\text{real}^{\sim} a). x \$ i)) (\text{at } x)$   
 $\langle \text{proof} \rangle$

**lemma** *constant-when-zero*:  
**fixes**  $v::\text{real} \Rightarrow (\text{real}, 'i::\text{finite}) \text{ vec}$   
**assumes**  $x0: (v t0) \$ i = x0$   
**assumes**  $\text{sol}: (v \text{ solves-ode } f) T S$   
**assumes**  $f0: \bigwedge s x. s \in T \Longrightarrow f s x \$ i = 0$   
**assumes**  $t0:t0 \in T$   
**assumes**  $t:t \in T$   
**assumes**  $\text{convex}: \text{convex } T$   
**shows**  $v t \$ i = x0$   
 $\langle \text{proof} \rangle$

**lemma**  
*solves-ode-subset*:  
**assumes**  $x: (x \text{ solves-ode } f) T X$   
**assumes**  $s: S \subseteq T$   
**shows**  $(x \text{ solves-ode } f) S X$   
 $\langle \text{proof} \rangle$

**lemma**  
*solves-ode-supset-range*:

```

assumes  $x$ : ( $x$  solves-ode  $f$ )  $T$   $X$ 
assumes  $y$ :  $X \subseteq Y$ 
shows ( $x$  solves-ode  $f$ )  $T$   $Y$ 
<proof>

lemma
  usolves-ode-subset:
assumes  $x$ : ( $x$  usolves-ode  $f$  from  $t0$ )  $T$   $X$ 
assumes  $s$ :  $S \subseteq T$ 
assumes  $t0$ :  $t0 \in S$ 
assumes  $S$ : is-interval  $S$ 
shows ( $x$  usolves-ode  $f$  from  $t0$ )  $S$   $X$ 
<proof>
lemma example:
  fixes  $x$   $t$ ::real and  $i$ ::('sz::finite)
  assumes  $t > 0$ 
  shows  $x = (ll\text{-on-open.flow UNIV } (\lambda t. \lambda x. \chi (i::('sz::finite)). 0) UNIV 0 (\chi i. x) t) \$ i$ 
<proof>

lemma MVT-ivl:
  fixes  $f$ ::'a::ordered-euclidean-space $\Rightarrow$ 'b::ordered-euclidean-space
  assumes  $fderiv$ :  $\bigwedge x. x \in D \implies (f \text{ has-derivative } J x) \text{ (at } x \text{ within } D)$ 
  assumes  $J\text{-ivl}$ :  $\bigwedge x. x \in D \implies J x u \geq J0$ 
  assumes  $line\text{-in}$ :  $\bigwedge x. x \in \{0..1\} \implies a + x *_R u \in D$ 
  shows  $f (a + u) - f a \geq J0$ 
<proof>

lemma MVT-ivl':
  fixes  $f$ ::'a::ordered-euclidean-space $\Rightarrow$ 'b::ordered-euclidean-space
  assumes  $fderiv$ :  $(\bigwedge x. x \in D \implies (f \text{ has-derivative } J x) \text{ (at } x \text{ within } D))$ 
  assumes  $J\text{-ivl}$ :  $\bigwedge x. x \in D \implies J x (a - b) \geq J0$ 
  assumes  $line\text{-in}$ :  $\bigwedge x. x \in \{0..1\} \implies b + x *_R (a - b) \in D$ 
  shows  $f a \geq f b + J0$ 
<proof>
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  $'c \Rightarrow ('a, 'c) \textit{trm}$  (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  $\vartheta$  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 ( $\$$  $\alpha$ )

— Assignment to a real-valued variable  $x := v$

| *Assign* 'c ('a, 'c) trm (**infixr** := 10)

— Assignment to a differential variable

| *DiffAssign* 'c ('a, 'c) trm

— Program  $?\varphi$  succeeds iff  $\varphi$  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**  $\cup\cup$  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  $\Rightarrow$  ('a, 'c) trm ( $\$$  $\varphi$ )

| *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  $\alpha$  where  $\varphi$  is true in end state

| *Diamond* ('a, 'b, 'c) hp ('a, 'b, 'c) formula ( $\langle\langle - \rangle - \rangle$  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  $\Rightarrow$  ('a, 'b, 'c) formula  $\Rightarrow$  ('a, 'b, 'c) formula  
(**infixl** || 7)

**where** *Or*  $P$   $Q$  = *Not* (*And* (*Not*  $P$ ) (*Not*  $Q$ ))

**definition** *Implies* :: ('a, 'b, 'c) formula  $\Rightarrow$  ('a, 'b, 'c) formula  $\Rightarrow$  ('a, 'b, 'c) formula  
(**infixr**  $\rightarrow$  10)

**where** *Implies*  $P$   $Q$  = *Or*  $Q$  (*Not*  $P$ )

**definition** *Equiv* :: ('a, 'b, 'c) formula  $\Rightarrow$  ('a, 'b, 'c) formula  $\Rightarrow$  ('a, 'b, 'c) formula  
(**infixl**  $\leftrightarrow$  10)

**where** *Equiv*  $P$   $Q$  = *Or* (*And*  $P$   $Q$ ) (*And* (*Not*  $P$ ) (*Not*  $Q$ ))

**definition** *Forall* :: 'c  $\Rightarrow$  ('a, 'b, 'c) formula  $\Rightarrow$  ('a, 'b, 'c) formula

**where** *Forall*  $x$   $P$  = *Not* (*Exists*  $x$  (*Not*  $P$ ))

**definition** *Equals* :: ('a, 'c) trm  $\Rightarrow$  ('a, 'c) trm  $\Rightarrow$  ('a, 'b, 'c) formula



**where**  $Equals \vartheta \vartheta' = ((Geq \vartheta \vartheta') \&\& (Geq \vartheta' \vartheta))$

**definition**  $Greater :: ('a, 'c) trm \Rightarrow ('a, 'c) trm \Rightarrow ('a, 'b, 'c) formula$   
**where**  $Greater \vartheta \vartheta' = ((Geq \vartheta \vartheta') \&\& (Not (Geq \vartheta' \vartheta)))$

**definition**  $Box :: ('a, 'b, 'c) hp \Rightarrow ('a, 'b, 'c) formula \Rightarrow ('a, 'b, 'c) formula$  ( $[[[-]]-$   
 $10)$

**where**  $Box \alpha P = Not (Diamond \alpha (Not P))$

**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**  $sizeF :: ('sf, 'sc, 'sz) formula \Rightarrow nat$

**and**  $sizeP :: ('sf, 'sc, 'sz) hp \Rightarrow nat$

**where**

$sizeP (Pvar a) = 1$   
 $| sizeP (Assign x \vartheta) = 1$   
 $| sizeP (DiffAssign x \vartheta) = 1$   
 $| sizeP (Test \varphi) = Suc (sizeF \varphi)$   
 $| sizeP (EvolveODE ODE \varphi) = Suc (sizeF \varphi)$   
 $| sizeP (Choice \alpha \beta) = Suc (sizeP \alpha + sizeP \beta)$   
 $| sizeP (Sequence \alpha \beta) = Suc (sizeP \alpha + sizeP \beta)$   
 $| sizeP (Loop \alpha) = Suc (sizeP \alpha)$   
 $| sizeF (Geq p q) = 1$   
 $| sizeF (Prop p args) = 1$   
 $| sizeF (Not p) = Suc (sizeF p)$   
 $| sizeF (And p q) = sizeF p + sizeF q$   
 $| sizeF (Exists x p) = Suc (sizeF p)$   
 $| sizeF (Diamond p q) = Suc (sizeP p + sizeF q)$   
 $| sizeF (InContext C \varphi) = Suc (sizeF \varphi)$

**lemma**  $sizeF\text{-diseq}: sizeF p \neq sizeF q \implies p \neq q$   $\langle proof \rangle$

**named-theorems**  $expr\text{-diseq}$  *Structural disequality rules for expressions*

**lemma**  $[expr\text{-diseq}]: p \neq And p q$   $\langle proof \rangle$

**lemma**  $[expr\text{-diseq}]: q \neq And p q$   $\langle proof \rangle$

**lemma**  $[expr\text{-diseq}]: p \neq Not p$   $\langle proof \rangle$

**lemma**  $[expr\text{-diseq}]: p \neq Or p q$   $\langle proof \rangle$

**lemma**  $[expr\text{-diseq}]: q \neq Or p q$   $\langle proof \rangle$

**lemma**  $[expr\text{-diseq}]: p \neq Implies p q$   $\langle proof \rangle$

```

lemma [expr-diseq]:q ≠ Implies p q ⟨proof⟩
lemma [expr-diseq]:p ≠ Equiv p q ⟨proof⟩
lemma [expr-diseq]:q ≠ Equiv p q ⟨proof⟩
lemma [expr-diseq]:p ≠ Exists x p ⟨proof⟩
lemma [expr-diseq]:p ≠ Diamond a p ⟨proof⟩
lemma [expr-diseq]:p ≠ InContext C p ⟨proof⟩
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 ≡ λi.(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 = (λi. (if i = vid1 then t else (Const 0)))
  ⟨proof⟩

```

```

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*:  $dfree \vartheta_1 \implies dfree \vartheta_2 \implies dfree (Plus \vartheta_1 \vartheta_2)$   
 | *dfree-Times*:  $dfree \vartheta_1 \implies dfree \vartheta_2 \implies dfree (Times \vartheta_1 \vartheta_2)$

**inductive** *dsafe* :: ('a, 'c) trm  $\Rightarrow$  bool

**where**

*dsafe-Var*:  $dsafe (Var i)$   
 | *dsafe-Const*:  $dsafe (Const r)$   
 | *dsafe-Fun*:  $(\bigwedge i. dsafe (args i)) \implies dsafe (Function i args)$   
 | *dsafe-Plus*:  $dsafe \vartheta_1 \implies dsafe \vartheta_2 \implies dsafe (Plus \vartheta_1 \vartheta_2)$   
 | *dsafe-Times*:  $dsafe \vartheta_1 \implies dsafe \vartheta_2 \implies dsafe (Times \vartheta_1 \vartheta_2)$   
 | *dsafe-Diff*:  $dfree \vartheta \implies dsafe (Differential \vartheta)$   
 | *dsafe-DiffVar*:  $dsafe (\$' i)$

— Explicitly-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  $\Rightarrow$  'c set

**where**

*ODE-dom* (OVar c) = {}  
 | *ODE-dom* (OSing x  $\vartheta$ ) = {x}  
 | *ODE-dom* (OProd ODE1 ODE2) = *ODE-dom* ODE1  $\cup$  *ODE-dom* ODE2

**inductive** *osafe*:: ('a, 'c) ODE  $\Rightarrow$  bool

**where**

*osafe-Var*:  $osafe (OVar c)$   
 | *osafe-Sing*:  $dfree \vartheta \implies osafe (OSing x \vartheta)$   
 | *osafe-Prod*:  $osafe ODE1 \implies osafe ODE2 \implies ODE-dom ODE1 \cap ODE-dom ODE2 = \{\} \implies osafe (OProd ODE1 ODE2)$

— Programs/formulas without any differential terms. This definition not currently used but may

— be useful in the future.

**inductive** *hpfree*:: ('a, 'b, 'c) hp  $\Rightarrow$  bool

**and** *ffree*:: ('a, 'b, 'c) formula  $\Rightarrow$  bool

**where**

*hpfree* (Pvar x)  
 | *dfree* e  $\implies$  *hpfree* (Assign x e)  
 — Differential programs allowed but not differential terms  
 | *dfree* e  $\implies$  *hpfree* (DiffAssign x e)  
 | *ffree* P  $\implies$  *hpfree* (Test P)  
 — Differential programs allowed but not differential terms  
 | *osafe* ODE  $\implies$  *ffree* P  $\implies$  *hpfree* (EvolveODE ODE P)  
 | *hpfree* a  $\implies$  *hpfree* b  $\implies$  *hpfree* (Choice a b)  
 | *hpfree* a  $\implies$  *hpfree* b  $\implies$  *hpfree* (Sequence a b)  
 | *hpfree* a  $\implies$  *hpfree* (Loop a)  
 | *ffree* f  $\implies$  *ffree* (InContext C f)  
 |  $(\bigwedge arg. arg \in range\ args \implies dfree\ arg) \implies ffree (Prop\ p\ args)$   
 | *ffree* p  $\implies$  *ffree* (Not p)  
 | *ffree* p  $\implies$  *ffree* q  $\implies$  *ffree* (And p q)

|  $\text{ffree } p \implies \text{ffree } (\text{Exists } x \ p)$   
|  $\text{hpfree } a \implies \text{ffree } p \implies \text{ffree } (\text{Diamond } a \ p)$   
|  $\text{ffree } (\text{Predicational } P)$   
|  $\text{dfree } t1 \implies \text{dfree } t2 \implies \text{ffree } (\text{Geq } t1 \ t2)$

**inductive**  $\text{hpsafe}:: ('a, 'b, 'c) \text{ hp} \implies \text{bool}$   
**and**  $\text{fsafe}:: ('a, 'b, 'c) \text{ formula} \implies \text{bool}$   
**where**

$\text{hpsafe-Pvar}:\text{hpsafe } (\text{Pvar } x)$   
|  $\text{hpsafe-Assign}:\text{dsafe } e \implies \text{hpsafe } (\text{Assign } x \ e)$   
|  $\text{hpsafe-DiffAssign}:\text{dsafe } e \implies \text{hpsafe } (\text{DiffAssign } x \ e)$   
|  $\text{hpsafe-Test}:\text{fsafe } P \implies \text{hpsafe } (\text{Test } P)$   
|  $\text{hpsafe-Evolve}:\text{osafe } \text{ODE} \implies \text{fsafe } P \implies \text{hpsafe } (\text{EvolveODE } \text{ODE} \ P)$   
|  $\text{hpsafe-Choice}:\text{hpsafe } a \implies \text{hpsafe } b \implies \text{hpsafe } (\text{Choice } a \ b)$   
|  $\text{hpsafe-Sequence}:\text{hpsafe } a \implies \text{hpsafe } b \implies \text{hpsafe } (\text{Sequence } a \ b)$   
|  $\text{hpsafe-Loop}:\text{hpsafe } a \implies \text{hpsafe } (\text{Loop } a)$

|  $\text{fsafe-Geq}:\text{dsafe } t1 \implies \text{dsafe } t2 \implies \text{fsafe } (\text{Geq } t1 \ t2)$   
|  $\text{fsafe-Prop}:(\bigwedge i. \text{dsafe } (\text{args } i)) \implies \text{fsafe } (\text{Prop } p \ \text{args})$   
|  $\text{fsafe-Not}:\text{fsafe } p \implies \text{fsafe } (\text{Not } p)$   
|  $\text{fsafe-And}:\text{fsafe } p \implies \text{fsafe } q \implies \text{fsafe } (\text{And } p \ q)$   
|  $\text{fsafe-Exists}:\text{fsafe } p \implies \text{fsafe } (\text{Exists } x \ p)$   
|  $\text{fsafe-Diamond}:\text{hpsafe } a \implies \text{fsafe } p \implies \text{fsafe } (\text{Diamond } a \ p)$   
|  $\text{fsafe-InContext}:\text{fsafe } f \implies \text{fsafe } (\text{InContext } C \ f)$

— Auto-generated simplifier rules for safety predicates

**inductive-simps**

$\text{dfree-Plus-simps}[\text{simp}]: \text{dfree } (\text{Plus } a \ b)$   
**and**  $\text{dfree-Times-simps}[\text{simp}]: \text{dfree } (\text{Times } a \ b)$   
**and**  $\text{dfree-Var-simps}[\text{simp}]: \text{dfree } (\text{Var } x)$   
**and**  $\text{dfree-DiffVar-simps}[\text{simp}]: \text{dfree } (\text{DiffVar } x)$   
**and**  $\text{dfree-Differential-simps}[\text{simp}]: \text{dfree } (\text{Differential } x)$   
**and**  $\text{dfree-Fun-simps}[\text{simp}]: \text{dfree } (\text{Function } i \ \text{args})$   
**and**  $\text{dfree-Const-simps}[\text{simp}]: \text{dfree } (\text{Const } r)$

**inductive-simps**

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

**inductive-simps**

$\text{osafe-OVar-simps}[\text{simp}]: \text{osafe } (\text{OVar } c)$   
**and**  $\text{osafe-OSing-simps}[\text{simp}]: \text{osafe } (\text{OSing } x \ \vartheta)$   
**and**  $\text{osafe-OProd-simps}[\text{simp}]: \text{osafe } (\text{OProd } \text{ODE1} \ \text{ODE2})$

**inductive-simps**

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

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

**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**  $dfree\text{-}is\text{-}dsafe: dfree \vartheta \Longrightarrow dsafe \vartheta$   
 $\langle proof \rangle$

**lemma**  $hp\text{-}induct [case\text{-}names Var Assign DiffAssign Test Evolve Choice Compose Star]:$

$(\bigwedge x. P (\$ \alpha x)) \Longrightarrow$   
 $(\bigwedge x1 x2. P (x1 := x2)) \Longrightarrow$   
 $(\bigwedge x1 x2. P (DiffAssign x1 x2)) \Longrightarrow$   
 $(\bigwedge x. P (? x)) \Longrightarrow$   
 $(\bigwedge x1 x2. P (EvolveODE x1 x2)) \Longrightarrow$   
 $(\bigwedge x1 x2. P x1 \Longrightarrow P x2 \Longrightarrow P (x1 \cup\cup x2)) \Longrightarrow$   
 $(\bigwedge x1 x2. P x1 \Longrightarrow P x2 \Longrightarrow P (x1 ;; x2)) \Longrightarrow$   
 $(\bigwedge x. P x \Longrightarrow P x^{**}) \Longrightarrow$   
 $P hp$   
 $\langle proof \rangle$

**lemma**  $fml\text{-}induct:$

$(\bigwedge t1 t2. P (Geq t1 t2))$   
 $\Longrightarrow (\bigwedge p args. P (Prop p args))$   
 $\Longrightarrow (\bigwedge p. P p \Longrightarrow P (Not p))$   
 $\Longrightarrow (\bigwedge p q. P p \Longrightarrow P q \Longrightarrow P (And p q))$

```

 $\implies (\bigwedge x p. P p \implies P (\text{Exists } x p))$ 
 $\implies (\bigwedge a p. P p \implies P (\text{Diamond } a p))$ 
 $\implies (\bigwedge C p. P p \implies P (\text{InContext } C p))$ 
 $\implies P \varphi$ 
<proof>

```

**context** *ids* **begin**

```

lemma proj-sing1:(singleton  $\vartheta$  vid1) =  $\vartheta$ 
  <proof>

```

```

lemma proj-sing2:vid1  $\neq$  y  $\implies$  (singleton  $\vartheta$  y) = (Const 0)
  <proof>

```

**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  $\nu$ , 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  $J$ .

— Vector of reals of length `'a`

**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**  $'a \text{ state} = 'a \text{ Rvec} \times 'a \text{ Rvec}$

—  $'a \text{ simple-state}$  is half a state - either the  $x$ s or the  $x'$ s

**type-synonym**  $'a \text{ simple-state} = 'a \text{ Rvec}$

**definition**  $Vagree :: 'c::\text{finite state} \Rightarrow 'c \text{ state} \Rightarrow ('c + 'c) \text{ set} \Rightarrow \text{bool}$

**where**  $Vagree \nu \nu' V \equiv$

$(\forall i. \text{Inl } i \in V \longrightarrow \text{fst } \nu \$ i = \text{fst } \nu' \$ i)$

$\wedge (\forall i. \text{Inr } i \in V \longrightarrow \text{snd } \nu \$ i = \text{snd } \nu' \$ i)$

**definition**  $VSagree :: 'c::\text{finite simple-state} \Rightarrow 'c \text{ simple-state} \Rightarrow 'c \text{ set} \Rightarrow \text{bool}$

**where**  $VSagree \nu \nu' V \longleftrightarrow (\forall i \in V. (\nu \$ i) = (\nu' \$ i))$

— Agreement lemmas

**lemma**  $agree\text{-nil}: Vagree \nu \omega \{\}$

$\langle \text{proof} \rangle$

**lemma**  $agree\text{-supset}: A \supseteq B \Longrightarrow Vagree \nu \nu' A \Longrightarrow Vagree \nu \nu' B$

$\langle \text{proof} \rangle$

**lemma**  $VSagree\text{-nil}: VSagree \nu \omega \{\}$

$\langle \text{proof} \rangle$

**lemma**  $VSagree\text{-supset}: A \supseteq B \Longrightarrow VSagree \nu \nu' A \Longrightarrow VSagree \nu \nu' B$

$\langle \text{proof} \rangle$

**lemma**  $VSagree\text{-UNIV-eq}: VSagree A B \text{ UNIV} \Longrightarrow A = B$

$\langle \text{proof} \rangle$

**lemma**  $agree\text{-comm}: \bigwedge A B V. Vagree A B V \Longrightarrow Vagree B A V \langle \text{proof} \rangle$

**lemma**  $agree\text{-sub}: \bigwedge \nu \omega A B. A \subseteq B \Longrightarrow Vagree \nu \omega B \Longrightarrow Vagree \nu \omega A$

$\langle \text{proof} \rangle$

**lemma**  $agree\text{-UNIV-eq}: \bigwedge \nu \omega. Vagree \nu \omega \text{ UNIV} \Longrightarrow \nu = \omega$

$\langle \text{proof} \rangle$

**lemma**  $agree\text{-UNIV-fst}: \bigwedge \nu \omega. Vagree \nu \omega (\text{Inl } ' \text{ UNIV}) \Longrightarrow (\text{fst } \nu) = (\text{fst } \omega)$

$\langle \text{proof} \rangle$

**lemma**  $agree\text{-UNIV-snd}: \bigwedge \nu \omega. Vagree \nu \omega (\text{Inr } ' \text{ UNIV}) \Longrightarrow (\text{snd } \nu) = (\text{snd } \omega)$

$\langle \text{proof} \rangle$

**lemma**  $Vagree\text{-univ}: \bigwedge a b c d. Vagree (a,b) (c,d) \text{ UNIV} \Longrightarrow a = c \wedge b = d$

$\langle \text{proof} \rangle$

**lemma**  $agree\text{-union}: \bigwedge \nu \omega A B. Vagree \nu \omega A \Longrightarrow Vagree \nu \omega B \Longrightarrow Vagree \nu \omega (A \cup B)$

*<proof>*

**lemma** *agree-trans*:  $Vagree \nu \mu A \implies Vagree \mu \omega B \implies Vagree \nu \omega (A \cap B)$   
*<proof>*

**lemma** *agree-refl*:  $Vagree \nu \nu A$   
*<proof>*

**lemma** *VSagree-sub*:  $\bigwedge \nu \omega A B . A \subseteq B \implies VSagree \nu \omega B \implies VSagree \nu \omega A$   
*<proof>*

**lemma** *VSagree-refl*:  $VSagree \nu \nu A$   
*<proof>*

### 3.4 Interpretations

For convenience we pretend interpretations contain an extra field called `FunctionFrechet` specifying the Frechet derivative (`FunctionFrechet f \<nu>`) :  $R^m \rightarrow R$  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*  $f' . \forall x . (\text{Functions } I i \text{ has-derivative } f' x)$   
  (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*  $\equiv$   
   $\forall x . \forall i . ((\text{FDERIV } (\text{Functions } I i) x \text{ :> } (\text{FunctionFrechet } I i x)) \wedge \text{continuous-on UNIV } (\lambda x . \text{Blinfun } (\text{FunctionFrechet } I i x)))$

**lemma** *is\_interpD*:  $is\_interp\ I \implies \forall x . \forall i . (\text{FDERIV } (\text{Functions } I i) x \text{ :> } (\text{FunctionFrechet } I i x))$   
*<proof>*

**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**  $Iagree\text{-}Func:Iagree\ I\ J\ V \implies Inl\ f \in V \implies Functions\ I\ f = Functions\ J\ f$   
 $\langle proof \rangle$

**lemma**  $Iagree\text{-}Contexts:Iagree\ I\ J\ V \implies Inr\ (Inl\ C) \in V \implies Contexts\ I\ C =$   
 $Contexts\ J\ C$   
 $\langle proof \rangle$

**lemma**  $Iagree\text{-}Pred:Iagree\ I\ J\ V \implies Inr\ (Inr\ p) \in V \implies Predicates\ I\ p =$   
 $Predicates\ J\ p$   
 $\langle proof \rangle$

**lemma**  $Iagree\text{-}Prog:Iagree\ I\ J\ V \implies Inr\ (Inr\ a) \in V \implies Programs\ I\ a = Programs$   
 $J\ a$   
 $\langle proof \rangle$

**lemma**  $Iagree\text{-}ODE:Iagree\ I\ J\ V \implies Inr\ (Inr\ a) \in V \implies ODEs\ I\ a = ODEs\ J\ a$   
 $\langle proof \rangle$

**lemma**  $Iagree\text{-}comm:\bigwedge A\ B\ V. Iagree\ A\ B\ V \implies Iagree\ B\ A\ V$   
 $\langle proof \rangle$

**lemma**  $Iagree\text{-}sub:\bigwedge I\ J\ A\ B. A \subseteq B \implies Iagree\ I\ J\ B \implies Iagree\ I\ J\ A$   
 $\langle proof \rangle$

**lemma**  $Iagree\text{-}refl:Iagree\ I\ I\ A$   
 $\langle proof \rangle$

**primrec**  $stern\text{-}sem :: ('a::finite, 'b::finite, 'c::finite)\ interm \Rightarrow ('a, 'c)\ trm \Rightarrow 'c$   
 $simple\text{-}state \Rightarrow real$

**where**

$stern\text{-}sem\ I\ (Var\ x)\ v = v\ \$\ x$   
 $| stern\text{-}sem\ I\ (Function\ f\ args)\ v = Functions\ I\ f\ (\chi\ i. stern\text{-}sem\ I\ (args\ i)\ v)$   
 $| stern\text{-}sem\ I\ (Plus\ t1\ t2)\ v = stern\text{-}sem\ I\ t1\ v + stern\text{-}sem\ I\ t2\ v$   
 $| stern\text{-}sem\ I\ (Times\ t1\ t2)\ v = stern\text{-}sem\ I\ t1\ v * stern\text{-}sem\ I\ t2\ v$   
 $| stern\text{-}sem\ I\ (Const\ r)\ v = r$   
 $| stern\text{-}sem\ I\ (\$\ 'c)\ v = undefined$   
 $| stern\text{-}sem\ I\ (Differential\ d)\ v = undefined$

—  $frechet\ I\ \vartheta\ \nu$  syntactically computes the frechet derivative of the term  $\vartheta$  in the interpretation

—  $I$  at state  $\nu$  (containing only the unprimed variables). The frechet derivative is

a

— linear map from the differential state  $\nu$  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* = ( $\lambda v'$ . *v*'  $\cdot$  *axis* *x* 1)  
| *frechet* *I* (*Function* *f* *args*) *v* =  
  ( $\lambda v'$ . *FunctionFrechet* *I* *f* ( $\chi$  *i*. *stern-sem* *I* (*args* *i*) *v*) ( $\chi$  *i*. *frechet* *I* (*args* *i*) *v*'))  
| *frechet* *I* (*Plus* *t1* *t2*) *v* = ( $\lambda v'$ . *frechet* *I* *t1* *v* *v'* + *frechet* *I* *t2* *v* *v'*)  
| *frechet* *I* (*Times* *t1* *t2*) *v* =  
  ( $\lambda v'$ . *stern-sem* *I* *t1* *v* \* *frechet* *I* *t2* *v* *v'* + *frechet* *I* *t1* *v* *v'* \* *stern-sem* *I* *t2* *v*)  
| *frechet* *I* (*Const* *r*) *v* = ( $\lambda v'$ . 0)  
| *frechet* *I* ( $\$$ ' *c*) *v* = *undefined*  
| *frechet* *I* (*Differential* *d*) *v* = *undefined*

**definition** *directional-derivative* :: ('a::finite, 'b::finite, 'c::finite) *interp*  $\Rightarrow$  ('a, 'c) *trm*  $\Rightarrow$  'c *state*  $\Rightarrow$  *real*

**where** *directional-derivative* *I* *t* = ( $\lambda v$ . *frechet* *I* *t* (*fst* *v*) (*snd* *v*))

— Sem for terms that are allowed to contain differentials.

— Note there is some duplication with *stern-sem*.

**primrec** *dterm-sem* :: ('a::finite, 'b::finite, 'c::finite) *interp*  $\Rightarrow$  ('a, 'c) *trm*  $\Rightarrow$  'c *state*  $\Rightarrow$  *real*

**where**

*dterm-sem* *I* (*Var* *x*) = ( $\lambda v$ . *fst* *v*  $\$$  *x*)  
| *dterm-sem* *I* (*DiffVar* *x*) = ( $\lambda v$ . *snd* *v*  $\$$  *x*)  
| *dterm-sem* *I* (*Function* *f* *args*) = ( $\lambda v$ . *Functions* *I* *f* ( $\chi$  *i*. *dterm-sem* *I* (*args* *i*) *v*))  
| *dterm-sem* *I* (*Plus* *t1* *t2*) = ( $\lambda v$ . (*dterm-sem* *I* *t1* *v*) + (*dterm-sem* *I* *t2* *v*))  
| *dterm-sem* *I* (*Times* *t1* *t2*) = ( $\lambda v$ . (*dterm-sem* *I* *t1* *v*) \* (*dterm-sem* *I* *t2* *v*))  
| *dterm-sem* *I* (*Differential* *t*) = ( $\lambda v$ . *directional-derivative* *I* *t* *v*)  
| *dterm-sem* *I* (*Const* *c*) = ( $\lambda v$ . *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*  $\Rightarrow$  ('a, 'c) *ODE*  $\Rightarrow$  'c *Rvec*  $\Rightarrow$  'c *Rvec*

**where**

*ODE-sem-OVar*:*ODE-sem* *I* (*OVar* *x*) = *ODEs* *I* *x*  
| *ODE-sem-OSing*:*ODE-sem* *I* (*OSing* *x*  $\vartheta$ ) = ( $\lambda v$ . ( $\chi$  *i*. if *i* = *x* then *stern-sem* *I*  $\vartheta$  *v* 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*:  $ODE\text{-sem } I (OProd\ ODE1\ ODE2) = (\lambda\nu. ODE\text{-sem } I\ ODE1\ \nu + ODE\text{-sem } I\ ODE2\ \nu)$

— The bound variables of an ODE

**fun** *ODE-vars* :: ('a,'b,'c) *interp*  $\Rightarrow$  ('a, 'c) *ODE*  $\Rightarrow$  'c *set*

**where**

*ODE-vars* *I* (*OVar* *c*) = *ODEBV* *I* *c*

| *ODE-vars* *I* (*OSing* *x*  $\vartheta$ ) = {*x*}

| *ODE-vars* *I* (*OProd* *ODE1* *ODE2*) = *ODE-vars* *I* *ODE1*  $\cup$  *ODE-vars* *I* *ODE2*

**fun** *semBV* :: ('a, 'b,'c) *interp*  $\Rightarrow$  ('a, 'c) *ODE*  $\Rightarrow$  ('c + 'c) *set*

**where** *semBV* *I* *ODE* = *Inl* ' (*ODE-vars* *I* *ODE*)  $\cup$  *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*  $\in$  *semBV* *I* *ODE*  $\longleftrightarrow$  *Inr* *x*  $\in$  *semBV* *I* *ODE*

*<proof>*

**fun** *mk-xode*::('a::finite, 'b::finite, 'c::finite) *interp*  $\Rightarrow$  ('a::finite, 'c::finite) *ODE*  $\Rightarrow$  'c::finite *simple-state*  $\Rightarrow$  'c::finite *state*

**where** *mk-xode* *I* *ODE* *sol* = (*sol*, *ODE-sem* *I* *ODE* *sol*)

Given an initial state  $\nu$  and solution to an ODE at some point, construct the resulting state  $\omega$ . This is defined using the *SOME* operator because the concrete definition is unwieldy.

**definition** *mk-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** *mk-v* *I* *ODE*  $\nu$  *sol* = (*THE*  $\omega$ .

*Vagree*  $\omega$   $\nu$  ( $-$  *semBV* *I* *ODE*)

$\wedge$  *Vagree*  $\omega$  (*mk-xode* *I* *ODE* *sol*) (*semBV* *I* *ODE*))

— *repv*  $\nu$  *x* *r* replaces the value of (unprimed) variable *x* in the state  $\nu$  with *r*

**fun** *repv* :: 'c::finite *state*  $\Rightarrow$  'c  $\Rightarrow$  *real*  $\Rightarrow$  'c *state*

**where** *repv* *v* *x* *r* = (( $\chi$  *y*. *if* *x* = *y* *then* *r* *else* *vec-nth* (*fst* *v*) *y*), *snd* *v*)

— *repd*  $\nu$  *x'* *r* replaces the value of (primed) variable *x'* in the state  $\nu$  with *r*

**fun** *repd* :: 'c::finite *state*  $\Rightarrow$  'c  $\Rightarrow$  *real*  $\Rightarrow$  'c *state*

**where** *repd* *v* *x* *r* = (*fst* *v*, ( $\chi$  *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*  $\Rightarrow$  ('a::finite, 'b::finite, 'c::finite) *formula*  $\Rightarrow$  'c::finite *state set* **and**

*prog-sem* :: ('a::finite, 'b::finite, 'c::finite) *interp*  $\Rightarrow$  ('a::finite, 'b::finite, 'c::finite) *hp*  $\Rightarrow$  ('c::finite *state* \* 'c::finite *state*) *set*

**where**

*fml-sem* *I* (*Geq* *t1* *t2*) = {*v*. *dterm-sem* *I* *t1* *v*  $\geq$  *dterm-sem* *I* *t2* *v*}

| *fml-sem* *I* (*Prop* *P* *terms*) = { $\nu$ . *Predicates* *I* *P* ( $\chi$  *i*. *dterm-sem* *I* (*terms* *i*)  $\nu$ )}

```

| fml-sem I (Not  $\varphi$ ) = {v. v  $\notin$  fml-sem I  $\varphi$ }
| fml-sem I (And  $\varphi$   $\psi$ ) = fml-sem I  $\varphi$   $\cap$  fml-sem I  $\psi$ 
| fml-sem I (Exists x  $\varphi$ ) = {v | v r. (repv v x r)  $\in$  fml-sem I  $\varphi$ }
| fml-sem I (Diamond  $\alpha$   $\varphi$ ) = { $\nu$  |  $\nu$   $\omega$ . ( $\nu$ ,  $\omega$ )  $\in$  prog-sem I  $\alpha$   $\wedge$   $\omega$   $\in$  fml-sem I  $\varphi$ }
| fml-sem I (InContext c  $\varphi$ ) = Contexts I c (fml-sem I  $\varphi$ )

| prog-sem I (Pvar p) = Programs I p
| prog-sem I (Assign x t) = {( $\nu$ ,  $\omega$ ).  $\omega$  = repv  $\nu$  x (dterm-sem I t  $\nu$ )}
| prog-sem I (DiffAssign x t) = {( $\nu$ ,  $\omega$ ).  $\omega$  = repd  $\nu$  x (dterm-sem I t  $\nu$ )}
| prog-sem I (Test  $\varphi$ ) = {( $\nu$ ,  $\nu$ ) |  $\nu$ .  $\nu$   $\in$  fml-sem I  $\varphi$ }
| prog-sem I (Choice  $\alpha$   $\beta$ ) = prog-sem I  $\alpha$   $\cup$  prog-sem I  $\beta$ 
| prog-sem I (Sequence  $\alpha$   $\beta$ ) = prog-sem I  $\alpha$  O prog-sem I  $\beta$ 
| prog-sem I (Loop  $\alpha$ ) = (prog-sem I  $\alpha$ )*
| prog-sem I (EvolveODE ODE  $\varphi$ ) =
  ({( $\nu$ , mk-v I ODE  $\nu$  (sol t)) |  $\nu$  sol t.
    t  $\geq$  0  $\wedge$ 
    (sol solves-ode ( $\lambda$ -. ODE-sem I ODE)) {0..t} {x. mk-v I ODE  $\nu$  x  $\in$  fml-sem
I  $\varphi$ }  $\wedge$ 
    sol 0 = fst  $\nu$ })

```

**context** *ids begin*

**definition** *valid* :: ('sf, 'sc, 'sz) formula  $\Rightarrow$  bool

**where** *valid*  $\varphi \equiv (\forall I. \forall \nu. \text{is-interp } I \longrightarrow \nu \in \text{fml-sem } I \varphi)$

**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 $\omega$ (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  $\nu$  sol =

(( $\chi$  i. (if Inl i  $\in$  semBV I ODE then sol else (fst  $\nu$ )) \$ i),  
( $\chi$  i. (if Inr i  $\in$  semBV I ODE then ODE-sem I ODE sol else (snd  $\nu$ )) \$ i))

**lemma** *mk-v-exists*: $\exists \omega. \text{Vagree } \omega \nu (- \text{semBV } I \text{ ODE})$

$\wedge \text{Vagree } \omega (\text{mk-xode } I \text{ ODE } \text{sol}) (\text{semBV } I \text{ ODE})$

*<proof>*

**lemma** *mk-v-agree*: $\text{Vagree } (\text{mk-v } I \text{ ODE } \nu \text{ sol}) \nu (- \text{semBV } I \text{ ODE})$

$\wedge \text{Vagree } (\text{mk-v } I \text{ ODE } \nu \text{ sol}) (\text{mk-xode } I \text{ ODE } \text{sol}) (\text{semBV } I \text{ ODE})$

*<proof>*

**lemma** *mk-v-concrete*: $\text{mk-v } I \text{ ODE } \nu \text{ sol} = ((\chi i. (\text{if Inl } i \in \text{semBV } I \text{ ODE then sol else (fst } \nu)) \$ i),$

$(\chi i. (\text{if Inr } i \in \text{semBV } I \text{ ODE then ODE-sem } I \text{ ODE sol else (snd } \nu)) \$ i))$

*<proof>*

### 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*:

$$\text{sterm-sem } I \text{ (Var } x) = (\lambda v. v \ \$ \ x)$$

*<proof>*

**lemma** *sconst-case*:

$$\text{sterm-sem } I \text{ (Const } r) = (\lambda v. r)$$

*<proof>*

**lemma** *sfunction-case*:

$$\text{sterm-sem } I \text{ (Function } f \ \text{args)} = (\lambda v. \text{Functions } I \ f \ (\chi \ i. \text{sterm-sem } I \ (\text{args } i) \ v))$$

*<proof>*

**lemma** *splus-case*:

$$\text{sterm-sem } I \text{ (Plus } t1 \ t2) = (\lambda v. (\text{sterm-sem } I \ t1 \ v) + (\text{sterm-sem } I \ t2 \ v))$$

*<proof>*

**lemma** *stimes-case*:

$$\text{sterm-sem } I \text{ (Times } t1 \ t2) = (\lambda v. (\text{sterm-sem } I \ t1 \ v) * (\text{sterm-sem } I \ t2 \ v))$$

*<proof>*

**lemma** *or-sem [simp]*:

$$\text{fml-sem } I \text{ (Or } \varphi \ \psi) = \text{fml-sem } I \ \varphi \cup \text{fml-sem } I \ \psi$$

*<proof>*

**lemma** *iff-sem [simp]*:  $(\nu \in \text{fml-sem } I \ (A \leftrightarrow B))$

$$\longleftrightarrow ((\nu \in \text{fml-sem } I \ A) \longleftrightarrow (\nu \in \text{fml-sem } I \ B))$$

*<proof>*

**lemma** *box-sem [simp]*:  $\text{fml-sem } I \ (\text{Box } \alpha \ \varphi) = \{\nu. \forall \omega. (\nu, \omega) \in \text{prog-sem } I \ \alpha \longrightarrow \omega \in \text{fml-sem } I \ \varphi\}$

*<proof>*

**lemma** *forall-sem [simp]*:  $\text{fml-sem } I \ (\text{Forall } x \ \varphi) = \{v. \forall r. (\text{repv } v \ x \ r) \in \text{fml-sem } I \ \varphi\}$

*<proof>*

**lemma** *greater-sem [simp]*:  $\text{fml-sem } I \ (\text{Greater } \vartheta \ \vartheta') = \{v. \text{dterm-sem } I \ \vartheta \ v > \text{dterm-sem } I \ \vartheta' \ v\}$

*<proof>*

**lemma** *loop-sem*:  $\text{prog-sem } I \ (\text{Loop } \alpha) = (\text{prog-sem } I \ \alpha)^*$

*<proof>*

**lemma** *impl-sem* [*simp*]:  $(\nu \in \text{fml-sem } I (A \rightarrow B))$   
 $= ((\nu \in \text{fml-sem } I A) \longrightarrow (\nu \in \text{fml-sem } I B))$   
 $\langle \text{proof} \rangle$

**lemma** *equals-sem* [*simp*]:  $(\nu \in \text{fml-sem } I (\text{Equals } \vartheta \vartheta'))$   
 $= (\text{dterm-sem } I \vartheta \nu = \text{dterm-sem } I \vartheta' \nu)$   
 $\langle \text{proof} \rangle$

**lemma** *diamond-sem* [*simp*]:  $\text{fml-sem } I (\text{Diamond } \alpha \varphi)$   
 $= \{\nu. \exists \omega. (\nu, \omega) \in \text{prog-sem } I \alpha \wedge \omega \in \text{fml-sem } I \varphi\}$   
 $\langle \text{proof} \rangle$

**lemma** *tt-sem* [*simp*]:  $\text{fml-sem } I TT = UNIV$   $\langle \text{proof} \rangle$

**lemma** *ff-sem* [*simp*]:  $\text{fml-sem } I FF = \{\}$   $\langle \text{proof} \rangle$

**lemma** *iff-to-impl*:  $((\nu \in \text{fml-sem } I A) \longleftrightarrow (\nu \in \text{fml-sem } I B))$   
 $\longleftrightarrow (((\nu \in \text{fml-sem } I A) \longrightarrow (\nu \in \text{fml-sem } I B))$   
 $\wedge ((\nu \in \text{fml-sem } I B) \longrightarrow (\nu \in \text{fml-sem } I A)))$   
 $\langle \text{proof} \rangle$

**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*  $I S = \text{fml-sem } I (\text{seq2fml } S)$

**lemma** *and-foldl-sem*:  $\nu \in \text{fml-sem } I (\text{foldr And } \Gamma TT) \Longrightarrow (\bigwedge \varphi. \text{List.member } \Gamma \varphi \Longrightarrow \nu \in \text{fml-sem } I \varphi)$   
 $\langle \text{proof} \rangle$

**lemma** *and-foldl-sem-conv*:  $(\bigwedge \varphi. \text{List.member } \Gamma \varphi \Longrightarrow \nu \in \text{fml-sem } I \varphi) \Longrightarrow \nu \in \text{fml-sem } I (\text{foldr And } \Gamma TT)$   
 $\langle \text{proof} \rangle$

**lemma** *or-foldl-sem*:  $\text{List.member } \Gamma \varphi \Longrightarrow \nu \in \text{fml-sem } I \varphi \Longrightarrow \nu \in \text{fml-sem } I (\text{foldr Or } \Gamma FF)$   
 $\langle \text{proof} \rangle$

**lemma** *or-foldl-sem-conv*:  $\nu \in \text{fml-sem } I (\text{foldr Or } \Gamma FF) \Longrightarrow \exists \varphi. \nu \in \text{fml-sem } I \varphi \wedge \text{List.member } \Gamma \varphi$   
 $\langle \text{proof} \rangle$

**lemma** *seq-semI'*:  $(\nu \in \text{fml-sem } I (\text{foldr And } \Gamma TT) \Longrightarrow \nu \in \text{fml-sem } I (\text{foldr Or } \Delta FF)) \Longrightarrow \nu \in \text{seq-sem } I (\Gamma, \Delta)$   
 $\langle \text{proof} \rangle$

**lemma** *seq-semD'*:  $\bigwedge P. \nu \in \text{seq-sem } I (\Gamma, \Delta) \Longrightarrow ((\nu \in \text{fml-sem } I (\text{foldr And } \Gamma$

$TT) \implies \nu \in \text{fml-sem } I (\text{foldr Or } \Delta \text{ FF}) \implies P \implies P$   
 ⟨proof⟩

**definition**  $\text{sublist}::'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$   
**where**  $\text{sublist } A \ B \equiv (\forall x. \text{List.member } A \ x \longrightarrow \text{List.member } B \ x)$

**lemma**  $\text{sublistI}:(\bigwedge x. \text{List.member } A \ x \implies \text{List.member } B \ x) \implies \text{sublist } A \ B$   
 ⟨proof⟩

**lemma**  $\Gamma\text{-sub-sem}:\text{sublist } \Gamma 1 \ \Gamma 2 \implies \nu \in \text{fml-sem } I (\text{foldr And } \Gamma 2 \ \text{TT}) \implies \nu \in \text{fml-sem } I (\text{foldr And } \Gamma 1 \ \text{TT})$   
 ⟨proof⟩

**lemma**  $\text{seq-semI}:\text{List.member } \Delta \ \psi \implies ((\bigwedge \varphi. \text{List.member } \Gamma \ \varphi \implies \nu \in \text{fml-sem } I \ \varphi) \implies \nu \in \text{fml-sem } I \ \psi) \implies \nu \in \text{seq-sem } I (\Gamma, \Delta)$   
 ⟨proof⟩

**lemma**  $\text{seq-semD}:\nu \in \text{seq-sem } I (\Gamma, \Delta) \implies (\bigwedge \varphi. \text{List.member } \Gamma \ \varphi \implies \nu \in \text{fml-sem } I \ \varphi) \implies \exists \varphi. (\text{List.member } \Delta \ \varphi) \wedge \nu \in \text{fml-sem } I \ \varphi$   
 ⟨proof⟩

**lemma**  $\text{seq-MP}:\nu \in \text{seq-sem } I (\Gamma, \Delta) \implies \nu \in \text{fml-sem } I (\text{foldr And } \Gamma \ \text{TT}) \implies \nu \in \text{fml-sem } I (\text{foldr Or } \Delta \ \text{FF})$   
 ⟨proof⟩

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

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

**lemma**  $\text{soundI}:(\bigwedge I. \text{is-interp } I \implies (\bigwedge i. i \geq 0 \implies i < \text{length } SG \implies \text{seq-sem } I (\text{nth } SG \ i) = \text{UNIV}) \implies \text{seq-sem } I \ G = \text{UNIV}) \implies \text{sound } (SG, G)$   
 ⟨proof⟩

**lemma**  $\text{soundI}':(\bigwedge I \ \nu. \text{is-interp } I \implies (\bigwedge i. i \geq 0 \implies i < \text{length } SG \implies \nu \in \text{seq-sem } I (\text{nth } SG \ i)) \implies \nu \in \text{seq-sem } I \ G) \implies \text{sound } (SG, G)$   
 ⟨proof⟩

**lemma**  $\text{soundI-mem}:(\bigwedge I. \text{is-interp } I \implies (\bigwedge \varphi. \text{List.member } SG \ \varphi \implies \text{seq-sem } I \ \varphi = \text{UNIV}) \implies \text{seq-sem } I \ C = \text{UNIV}) \implies \text{sound } (SG, C)$   
 ⟨proof⟩

**lemma**  $\text{soundI-memv}:(\bigwedge I. \text{is-interp } I \implies (\bigwedge \varphi \ \nu. \text{List.member } SG \ \varphi \implies \nu \in \text{seq-sem } I \ \varphi) \implies (\bigwedge \nu. \nu \in \text{seq-sem } I \ C) \implies \text{sound } (SG, C)$

*<proof>*

**lemma** *soundI-memv'*: $(\bigwedge I. \text{is-interp } I \implies (\bigwedge \varphi \nu. \text{List.member } SG \ \varphi \implies \nu \in \text{seq-sem } I \ \varphi) \implies (\bigwedge \nu. \nu \in \text{seq-sem } I \ C)) \implies R = (SG, C) \implies \text{sound } R$   
*<proof>*

**lemma** *soundD-mem:sound*  $(SG, C) \implies (\bigwedge I. \text{is-interp } I \implies (\bigwedge \varphi. \text{List.member } SG \ \varphi \implies \text{seq-sem } I \ \varphi = UNIV) \implies \text{seq-sem } I \ C = UNIV)$   
*<proof>*

**lemma** *soundD-memv:sound*  $(SG, C) \implies (\bigwedge I. \text{is-interp } I \implies (\bigwedge \varphi \nu. \text{List.member } SG \ \varphi \implies \nu \in \text{seq-sem } I \ \varphi) \implies (\bigwedge \nu. \nu \in \text{seq-sem } I \ C))$   
*<proof>*

**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 to 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*  $\equiv$

$([[\text{vid1} := (\$f \text{fid1 empty})]] (\text{Prop vid1} (\text{singleton} (\text{Var vid1}))))$

$\leftrightarrow \text{Prop vid1} (\text{singleton} (\$f \text{fid1 empty}))$

**definition** *diff-assign-axiom* :: ('sf, 'sc, 'sz) formula

**where** [*axiom-defs*]:*diff-assign-axiom*  $\equiv$

$([[\text{DiffAssign vid1} (\$f \text{fid1 empty})]] (\text{Prop vid1} (\text{singleton} (\text{DiffVar vid1}))))$

$\leftrightarrow \text{Prop vid1} (\text{singleton} (\$f \text{fid1 empty}))$



**definition** *loop-iterate-axiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*loop-iterate-axiom*  $\equiv$  ([[ $\alpha$  vid1\*\*]]Predicational pid1)  
 $\leftrightarrow$  ((Predicational pid1) && ([[ $\alpha$  vid1]] [[ $\alpha$  vid1\*\*]]Predicational pid1))

**definition** *test-axiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*test-axiom*  $\equiv$   
 ([[ $\varphi$  vid2 empty]]) $\varphi$  vid1 empty  $\leftrightarrow$  (( $\varphi$  vid2 empty)  $\rightarrow$  ( $\varphi$  vid1 empty))

**definition** *box-axiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*box-axiom*  $\equiv$  (( $\alpha$  vid1)Predicational pid1)  $\leftrightarrow$  !([[ $\alpha$  vid1]]!(Predicational pid1))

**definition** *choice-axiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*choice-axiom*  $\equiv$  ([[ $\alpha$  vid1  $\cup\cup$   $\alpha$  vid2]]Predicational pid1)  
 $\leftrightarrow$  ((([[ $\alpha$  vid1]]Predicational pid1) && ([[ $\alpha$  vid2]]Predicational pid1))

**definition** *compose-axiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*compose-axiom*  $\equiv$  ([[ $\alpha$  vid1 ;;  $\alpha$  vid2]]Predicational pid1)  $\leftrightarrow$   
 ([[ $\alpha$  vid1]] [[  $\alpha$  vid2]]Predicational pid1)

**definition** *Kaxiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*Kaxiom*  $\equiv$  ([[ $\alpha$  vid1]]((Predicational pid1)  $\rightarrow$  (Predicational pid2)))  
 $\rightarrow$  ([[ $\alpha$  vid1]]Predicational pid1)  $\rightarrow$  ([[ $\alpha$  vid1]]Predicational pid2)

**definition** *Iaxiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*Iaxiom*  $\equiv$   
 ([[ $\alpha$  vid1\*\*]](Predicational pid1  $\rightarrow$  ([[ $\alpha$  vid1]]Predicational pid1)))  
 $\rightarrow$ ((Predicational pid1  $\rightarrow$  ([[ $\alpha$  vid1\*\*]]Predicational pid1)))

**definition** *Vaxiom* :: ('sf, 'sc, 'sz) formula  
**where** [axiom-defs]:*Vaxiom*  $\equiv$  ( $\varphi$  vid1 empty)  $\rightarrow$  ([[ $\alpha$  vid1]]( $\varphi$  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  
 <proof>

**lemma** *assign-lem1*:  
 dterm-sem I (if i = vid1 then Var vid1 else (Const 0))  
 (vec-lambda ( $\lambda y$ . if vid1 = y then Functions I fid1  
 (vec-lambda ( $\lambda i$ . dterm-sem I (empty i)  $\nu$ )) else vec-nth (fst  $\nu$ ) y), snd  $\nu$ )  
 =  
 dterm-sem I (if i = vid1 then  $\$$  fid1 empty else (Const 0))  $\nu$

*<proof>*

**lemma** *diff-assign-lem1:*

*dterm-sem I (if i = vid1 then DiffVar vid1 else (Const 0))*  
*(fst  $\nu$ , vec-lambda ( $\lambda y$ . if vid1 = y then Functions I fid1 (vec-lambda*  
*( $\lambda i$ . dterm-sem I (empty i)  $\nu$ )) else vec-nth (snd  $\nu$ ) y))*  
*=*  
*dterm-sem I (if i = vid1 then  $\$f$  fid1 empty else (Const 0))  $\nu$*

*<proof>*

**theorem** *assign-valid: valid assign-axiom*

*<proof>*

**theorem** *diff-assign-valid: valid diff-assign-axiom*

*<proof>*

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

*<proof>*

**lemma** *loop-forward:  $\nu \in \text{fml-sem } I \text{ } ([[ \$\alpha \text{ id1**} ]]\text{Predicational pid1})$*

*$\longrightarrow \nu \in \text{fml-sem } I \text{ } (\text{Predicational pid1} \&\& [[ \$\alpha \text{ id1} ]][ [ \$\alpha \text{ id1**} ]]\text{Predicational pid1})$*

*<proof>*

**lemma** *loop-backward:*

*$\nu \in \text{fml-sem } I \text{ } (\text{Predicational pid1} \&\& [[ \$\alpha \text{ id1} ]][ [ \$\alpha \text{ id1**} ]]\text{Predicational pid1})$*

*$\longrightarrow \nu \in \text{fml-sem } I \text{ } ([[ \$\alpha \text{ id1**} ]]\text{Predicational pid1})$*

*<proof>*

**theorem** *loop-valid: valid loop-iterate-axiom*

*<proof>*

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

*<proof>*

**theorem** *choice-valid: valid choice-axiom*

*<proof>*

**theorem** *compose-valid: valid compose-axiom*

*<proof>*

**theorem** *K-valid: valid Kaxiom*

*<proof>*

**lemma** *I-axiom-lemma:*

**fixes** *I::('sf,'sc,'sz) interp and  $\nu$*

**assumes** *is-interp I*

**assumes** *IS: $\nu \in \text{fml-sem } I \text{ } ([[ \$\alpha \text{ vid1**} ]]\text{Predicational pid1} \rightarrow$*   
*[[ \$\alpha \text{ vid1} ]]\text{Predicational pid1})*

**assumes**  $BC:\nu \in \text{fml-sem } I$  (*Predicational pid1*)  
**shows**  $\nu \in \text{fml-sem } I$  ( $[[\$ \alpha \text{ vid1} **]]$ (*Predicational pid1*)  
 $\langle \text{proof} \rangle$

**theorem** *I-valid: valid Iaxiom*  
 $\langle \text{proof} \rangle$

**theorem** *V-valid: valid Vaxiom*  
 $\langle \text{proof} \rangle$

**definition** *G-holds* ::  $(\text{'sf}, \text{'sc}, \text{'sz}) \text{ formula} \Rightarrow (\text{'sf}, \text{'sc}, \text{'sz}) \text{ hp} \Rightarrow \text{bool}$   
**where** *G-holds*  $\varphi \alpha \equiv \text{valid } \varphi \longrightarrow \text{valid } ([[ \alpha ]]$

**definition** *Skolem-holds* ::  $(\text{'sf}, \text{'sc}, \text{'sz}) \text{ formula} \Rightarrow \text{'sz} \Rightarrow \text{bool}$   
**where** *Skolem-holds*  $\varphi \text{ var} \equiv \text{valid } \varphi \longrightarrow \text{valid } (\text{Forall } \text{var } \varphi)$

**definition** *MP-holds* ::  $(\text{'sf}, \text{'sc}, \text{'sz}) \text{ formula} \Rightarrow (\text{'sf}, \text{'sc}, \text{'sz}) \text{ formula} \Rightarrow \text{bool}$   
**where** *MP-holds*  $\varphi \psi \equiv \text{valid } (\varphi \rightarrow \psi) \longrightarrow \text{valid } \varphi \longrightarrow \text{valid } \psi$

**definition** *CT-holds* ::  $\text{'sf} \Rightarrow (\text{'sf}, \text{'sz}) \text{ trm} \Rightarrow (\text{'sf}, \text{'sz}) \text{ trm} \Rightarrow \text{bool}$   
**where** *CT-holds*  $g \vartheta \vartheta' \equiv \text{valid } (\text{Equals } \vartheta \vartheta')$   
 $\longrightarrow \text{valid } (\text{Equals } (\text{Function } g (\text{singleton } \vartheta)) (\text{Function } g (\text{singleton } \vartheta')))$

**definition** *CQ-holds* ::  $\text{'sz} \Rightarrow (\text{'sf}, \text{'sz}) \text{ trm} \Rightarrow (\text{'sf}, \text{'sz}) \text{ trm} \Rightarrow \text{bool}$   
**where** *CQ-holds*  $p \vartheta \vartheta' \equiv \text{valid } (\text{Equals } \vartheta \vartheta')$   
 $\longrightarrow \text{valid } ((\text{Prop } p (\text{singleton } \vartheta)) \leftrightarrow (\text{Prop } p (\text{singleton } \vartheta')))$

**definition** *CE-holds* ::  $\text{'sc} \Rightarrow (\text{'sf}, \text{'sc}, \text{'sz}) \text{ formula} \Rightarrow (\text{'sf}, \text{'sc}, \text{'sz}) \text{ formula} \Rightarrow \text{bool}$   
**where** *CE-holds*  $\text{var } \varphi \psi \equiv \text{valid } (\varphi \leftrightarrow \psi)$   
 $\longrightarrow \text{valid } (\text{InContext } \text{var } \varphi \leftrightarrow \text{InContext } \text{var } \psi)$

## 4.2 Soundness proofs for rules

**theorem** *G-sound: G-holds*  $\varphi \alpha$   
 $\langle \text{proof} \rangle$

**theorem** *Skolem-sound: Skolem-holds*  $\varphi \text{ var}$   
 $\langle \text{proof} \rangle$

**theorem** *MP-sound: MP-holds*  $\varphi \psi$   
 $\langle \text{proof} \rangle$

**lemma** *CT-lemma*:  $\bigwedge I :: (\text{'sf} :: \text{finite}, \text{'sc} :: \text{finite}, \text{'sz} :: \{\text{finite}, \text{linorder}\}) \text{ interp. } \bigwedge a :: (\text{real}, \text{'sz}) \text{ vec. } \bigwedge b :: (\text{real}, \text{'sz}) \text{ vec. } \forall I :: (\text{'sf}, \text{'sc}, \text{'sz}) \text{ interp. is-interp } I \longrightarrow (\forall a \text{ b. } \text{dterm-sem } I \vartheta (a, b) = \text{dterm-sem } I \vartheta' (a, b)) \implies$   
 $\text{is-interp } I \implies$   
 $\text{Functions } I \text{ var } (\text{vec-lambda } (\lambda i. \text{dterm-sem } I (\text{if } i = \text{vid1} \text{ then } \vartheta \text{ else } (\text{Const } 0)) (a, b))) =$

$$\text{Functions } I \text{ var } (\text{vec-lambda } (\lambda i. \text{dterm-sem } I \text{ (if } i = \text{vid1 then } \vartheta' \text{ else } (\text{Const } 0)) (a, b)))$$
 <proof>

**theorem** *CT-sound*: *CT-holds var  $\vartheta \vartheta'$*   
 <proof>

**theorem** *CQ-sound*: *CQ-holds var  $\vartheta \vartheta'$*   
 <proof>

**theorem** *CE-sound*: *CE-holds var  $\varphi \psi$*   
 <proof>

**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**  $(\lambda(v::'a \text{ Rvec}). v \cdot \text{axis } i \ 1) = (\lambda(v::'a \text{ Rvec}). v \$ i)$

<proof>

**theorem** *svar-deriv*:

**fixes** *x::'sv::finite and  $\nu::'sv \text{ Rvec}$  and  $F::\text{real filter}$*

**shows**  $((\lambda v. v \$ x) \text{ has-derivative } (\lambda v'. v' \cdot (\chi \ i. \text{if } i = x \text{ then } 1 \text{ else } 0))) \text{ (at } \nu)$

<proof>

**lemma** *function-case-inner*:

**assumes** *good-interp*:

$(\forall x \ i. (\text{Functions } I \ i \text{ has-derivative } \text{FunctionFrechet } I \ i \ x) \text{ (at } x))$

**assumes** *IH*: $((\lambda v. \chi \ i. \text{sterm-sem } I \ (args \ i) \ v)$

$\text{has-derivative } (\lambda v. (\chi \ i. \text{frechet } I \ (args \ i) \ \nu \ v))) \text{ (at } \nu)$

**shows**  $((\lambda v. \text{Functions } I \ f \ (\chi \ i. \text{sterm-sem } I \ (args \ i) \ v))$

$\text{has-derivative } (\lambda v. \text{frechet } I \ (\$f \ f \ args) \ \nu \ v) \text{ (at } \nu)$

<proof>

**lemma** *func-lemma2*: $(\forall x i. (\text{Functions } I i \text{ has-derivative } (\text{THE } f'. \forall x. (\text{Functions } I i \text{ has-derivative } f' x) (\text{at } x)) x) (\text{at } x) \wedge$   
*continuous-on UNIV*  $(\lambda x. \text{Blinfun } ((\text{THE } f'. \forall x. (\text{Functions } I i \text{ has-derivative } f' x) (\text{at } x)) x))) \implies$   
 $(\bigwedge \vartheta. \vartheta \in \text{range args} \implies (\text{stern-sem } I \vartheta \text{ has-derivative frechet } I \vartheta \nu) (\text{at } \nu))$   
 $\implies$   
 $((\lambda v. \text{Functions } I f (\text{vec-lambda}(\lambda i. \text{stern-sem } I (\text{args } i) v))) \text{ has-derivative } (\lambda v'.$   
 $(\text{THE } f'. \forall x. (\text{Functions } I f \text{ has-derivative } f' x) (\text{at } x)) (\chi i. \text{stern-sem } I (\text{args } i)$   
 $\nu) (\chi i. \text{frechet } I (\text{args } i) \nu v')) (\text{at } \nu)$   
 $\langle \text{proof} \rangle$

**lemma** *func-lemma*:

*is-interp*  $I \implies$   
 $(\bigwedge \vartheta :: ('a::\text{finite}, 'c::\text{finite}) \text{ trm}. \vartheta \in \text{range args} \implies (\text{stern-sem } I \vartheta \text{ has-derivative}$   
 $\text{frechet } I \vartheta \nu) (\text{at } \nu)) \implies$   
 $(\text{stern-sem } I (\$f f \text{ args}) \text{ has-derivative frechet } I (\$f f \text{ args}) \nu) (\text{at } \nu)$   
 $\langle \text{proof} \rangle$

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**  $I :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  **and**  $\nu$   
**assumes** *good-interp*: *is-interp*  $I$   
**shows**  $\text{dfree } \vartheta \implies \text{FDERIV } (\text{stern-sem } I \vartheta) \nu :> (\text{frechet } I \vartheta \nu)$   
 $\langle \text{proof} \rangle$

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

**lemma** *stern-determines-frechet*:

**fixes**  $I :: ('a1::\text{finite}, 'b1::\text{finite}, 'c::\text{finite}) \text{ interp}$   
**and**  $J :: ('a2::\text{finite}, 'b2::\text{finite}, 'c::\text{finite}) \text{ interp}$   
**and**  $\vartheta1 :: ('a1::\text{finite}, 'c::\text{finite}) \text{ trm}$   
**and**  $\vartheta2 :: ('a2::\text{finite}, 'c::\text{finite}) \text{ trm}$   
**and**  $\nu$   
**assumes** *good-interp1*:*is-interp*  $I$   
**assumes** *good-interp2*:*is-interp*  $J$   
**assumes** *free1*:*dfree*  $\vartheta1$   
**assumes** *free2*:*dfree*  $\vartheta2$   
**assumes** *sem*:*stern-sem*  $I \vartheta1 = \text{stern-sem } J \vartheta2$   
**shows**  $\text{frechet } I \vartheta1 (\text{fst } \nu) (\text{snd } \nu) = \text{frechet } J \vartheta2 (\text{fst } \nu) (\text{snd } \nu)$   
 $\langle \text{proof} \rangle$

**lemma** *the-deriv*:

**assumes** *deriv*: $(f \text{ has-derivative } F) (\text{at } x)$   
**shows**  $(\text{THE } G. (f \text{ has-derivative } G) (\text{at } x)) = F$   
 $\langle \text{proof} \rangle$

**lemma** *the-all-deriv*:

**assumes** *deriv*: $\forall x. (f \text{ has-derivative } F x) (at x)$

**shows** (*THE*  $G. \forall x. (f \text{ has-derivative } G x) (at x) = F$ )  
*<proof>*

**typedef** (*'a, 'c*) *strm* =  $\{\vartheta :: ('a, 'c) \text{ trm. } d\text{free } \vartheta\}$

**morphisms** *raw-term simple-term*

*<proof>*

**typedef** (*'a, 'b, 'c*) *good-interp* =  $\{I :: ('a :: \text{finite}, 'b :: \text{finite}, 'c :: \text{finite}) \text{ interp. } is\text{-interp } I\}$

**morphisms** *raw-interp good-interp*

*<proof>*

**lemma** *frechet-linear*:

**assumes** *good-interp:is-interp I*

**fixes** *v*  $\vartheta$

**shows**  $d\text{free } \vartheta \implies \text{bounded-linear } (\text{frechet } I \vartheta v)$

*<proof>*

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

**setup-lifting** *type-definition-strm*

**lift-definition** *blin-frechet*::(*'sf, 'sc, 'sz*) *good-interp*  $\Rightarrow$  (*'sf, 'sz*) *strm*  $\Rightarrow$  (*real, 'sz*)  
*vec*  $\Rightarrow$  (*real, 'sz*) *vec*  $\Rightarrow_L$  *real is frechet*

*<proof>*

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

**lemma** *frechet-blin:is-interp I*  $\implies$   $d\text{free } \vartheta \implies (\lambda v. \text{Blinfun } (\lambda v'. \text{frechet } I \vartheta v v'))$   
= *blin-frechet (good-interp I) (simple-term*  $\vartheta)$

*<proof>*

**lemma** *stern-continuous*:

**assumes** *good-interp:is-interp I*

**shows**  $d\text{free } \vartheta \implies \text{continuous-on } UNIV (\text{stern-sem } I \vartheta)$

*<proof>*

**lemma** *stern-continuous'*:

**assumes** *good-interp:is-interp I*

**shows**  $d\text{free } \vartheta \implies \text{continuous-on } S (\text{stern-sem } I \vartheta)$

*<proof>*

**lemma** *frechet-continuous*:

**fixes** *I* :: (*'sf, 'sc, 'sz*) *interp*

**assumes** *good-interp:is-interp I*

**shows**  $d\text{free } \vartheta \implies \text{continuous-on } UNIV (\text{blin-frechet } (\text{good-interp } I) (\text{simple-term}$

```

 $\vartheta$ )
<proof>
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

```

$SIGO (OVar\ c) = \{Inr\ c\}$   
 $| SIGO (OSing\ x\ \vartheta) = \{Inl\ x \mid x. x \in SIGT\ \vartheta\}$   
 $| SIGO (OProd\ ODE1\ ODE2) = SIGO\ ODE1 \cup SIGO\ ODE2$

**primrec**  $SIGP :: ('a, 'b, 'c)\ hp \Rightarrow ('a + 'b + 'c)\ set$   
**and**  $SIGF :: ('a, 'b, 'c)\ formula \Rightarrow ('a + 'b + 'c)\ set$   
**where**

$SIGP (Pvar\ var) = \{Inr\ (Inr\ var)\}$   
 $| SIGP (Assign\ var\ t) = \{Inl\ x \mid x. x \in SIGT\ t\}$   
 $| SIGP (DiffAssign\ var\ t) = \{Inl\ x \mid x. x \in SIGT\ t\}$   
 $| SIGP (Test\ p) = SIGF\ p$   
 $| SIGP (EvolveODE\ ODE\ p) = SIGF\ p \cup \{Inl\ x \mid x. Inl\ x \in SIGO\ ODE\} \cup \{Inr\ (Inr\ x) \mid x. Inr\ x \in SIGO\ ODE\}$   
 $| SIGP (Choice\ a\ b) = SIGP\ a \cup SIGP\ b$   
 $| SIGP (Sequence\ a\ b) = SIGP\ a \cup SIGP\ b$   
 $| SIGP (Loop\ a) = SIGP\ a$   
 $| SIGF (Geq\ t1\ t2) = \{Inl\ x \mid x. x \in SIGT\ t1 \cup SIGT\ t2\}$   
 $| SIGF (Prop\ var\ args) = \{Inr\ (Inr\ var)\} \cup \{Inl\ x \mid x. x \in (\bigcup i. SIGT\ (args\ i))\}$   
 $| SIGF (Not\ p) = SIGF\ p$   
 $| SIGF (And\ p1\ p2) = SIGF\ p1 \cup SIGF\ p2$   
 $| SIGF (Exists\ var\ p) = SIGF\ p$   
 $| SIGF (Diamond\ a\ p) = SIGP\ a \cup SIGF\ p$   
 $| SIGF (InContext\ var\ p) = \{Inr\ (Inl\ var)\} \cup SIGF\ p$

**fun**  $primify :: ('a + 'a) \Rightarrow ('a + 'a)\ set$   
**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**  $FVT :: ('a, 'c)\ trm \Rightarrow ('c + 'c)\ set$   
**where**

$FVT (Var\ x) = \{Inl\ x\}$   
 $| FVT (Const\ x) = \{\}$   
 $| FVT (Function\ f\ args) = (\bigcup i. FVT\ (args\ i))$   
 $| FVT (Plus\ f\ g) = FVT\ f \cup FVT\ g$   
 $| FVT (Times\ f\ g) = FVT\ f \cup FVT\ g$   
 $| FVT (Differential\ f) = (\bigcup x \in (FVT\ f). primify\ x)$   
 $| FVT (DiffVar\ x) = \{Inr\ x\}$

**fun**  $FVDiff :: ('a, 'c)\ trm \Rightarrow ('c + 'c)\ set$   
**where**  $FVDiff\ f = (\bigcup x \in (FVT\ f). primify\ x)$



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

```

fun FVO :: ('a, 'c) ODE  $\Rightarrow$  'c set
where
  FVO (OVar c) = UNIV
| FVO (OSing x  $\vartheta$ ) = {x}  $\cup$  {x . Inl x  $\in$  FVT  $\vartheta$ }
| FVO (OProd ODE1 ODE2) = FVO ODE1  $\cup$  FVO ODE2

```

Bound variables of ODEs, formulas, programs

```

fun BVO :: ('a, 'c) ODE  $\Rightarrow$  ('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-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 -) = (Inl ' (ODE-dom ODE))  $\cup$  (Inr ' (ODE-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

```

$FVF (Geq f g) = FVT f \cup FVT g$   
 $FVF (Prop p args) = (\bigcup i. FVT (args i))$   
 $FVF (Not p) = FVF p$   
 $FVF (And p q) = FVF p \cup FVF q$   
 $FVF (Exists x p) = FVF p - \{Inl x\}$   
 $FVF (Diamond \alpha p) = FVP \alpha \cup (FVF p - MBV \alpha)$   
 $FVF (InContext C p) = UNIV$   
 $FVP (Pvar a) = UNIV$   
 $FVP (Assign x v) = FVT v$   
 $FVP (DiffAssign x v) = FVT v$   
 $FVP (Test \varphi) = FVF \varphi$   
 $FVP (EvolveODE ODE \varphi) = BVO ODE \cup (Inl \text{ ' } FVO ODE) \cup FVF \varphi$   
 $FVP (Choice \alpha \beta) = FVP \alpha \cup FVP \beta$   
 $FVP (Sequence \alpha \beta) = FVP \alpha \cup (FVP \beta - MBV \alpha)$   
 $FVP (Loop \alpha) = FVP \alpha$

### 6.3 Lemmas for reasoning about static semantics

**lemma** *primify-contains*:  $x \in \text{primify } x$   
*<proof>*

**lemma** *FVDiff-sub*:  $FVT f \subseteq FVDiff f$   
*<proof>*

**lemma** *fvdiff-plus1*:  $FVDiff (Plus t1 t2) = FVDiff t1 \cup FVDiff t2$   
*<proof>*

**lemma** *agree-func-fvt*:  $Vagree \nu \nu' (FVT (Function f args)) \implies Vagree \nu \nu' (FVT (args i))$   
*<proof>*

**lemma** *agree-plus1*:  $Vagree \nu \nu' (FVDiff (Plus t1 t2)) \implies Vagree \nu \nu' (FVDiff t1)$   
*<proof>*

**lemma** *agree-plus2*:  $Vagree \nu \nu' (FVDiff (Plus t1 t2)) \implies Vagree \nu \nu' (FVDiff t2)$   
*<proof>*

**lemma** *agree-times1*:  $Vagree \nu \nu' (FVDiff (Times t1 t2)) \implies Vagree \nu \nu' (FVDiff t1)$   
*<proof>*

**lemma** *agree-times2*:  $Vagree \nu \nu' (FVDiff (Times t1 t2)) \implies Vagree \nu \nu' (FVDiff t2)$   
*<proof>*

**lemma** *agree-func*:  $Vagree \nu \nu' (FVDiff (\$f var args)) \implies (\bigwedge i. Vagree \nu \nu' (FVDiff (args i)))$   
*<proof>*

```

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-term*:  $Vagree\ \nu\ \nu'\ (FVT\ \vartheta) \implies sterm\text{-}sem\ I\ \vartheta\ (fst\ \nu) = sterm\text{-}sem\ I\ \vartheta\ (fst\ \nu')$   
 ⟨proof⟩

**lemma** *coincidence-term'*:  $dfree\ \vartheta \implies Vagree\ \nu\ \nu'\ (FVT\ \vartheta) \implies Iagree\ I\ J\ \{Inl\ x\ |x.\ x \in SIGT\ \vartheta\} \implies sterm\text{-}sem\ I\ \vartheta\ (fst\ \nu) = sterm\text{-}sem\ J\ \vartheta\ (fst\ \nu')$   
 ⟨proof⟩

**lemma** *sum-unique-nonzero*:  
**fixes**  $i::'sv::finite$  **and**  $f::'sv \Rightarrow real$

**assumes**  $\text{restZero} \wedge j. j \in (\text{UNIV} :: 'sv \text{ set}) \implies j \neq i \implies f j = 0$   
**shows**  $(\sum j \in (\text{UNIV} :: 'sv \text{ set}). f j) = f i$   
 $\langle \text{proof} \rangle$

**lemma** *coincidence-frechet* :

**fixes**  $I :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  **and**  $\nu :: 'c \text{ state}$  **and**  $\nu' :: 'c \text{ state}$   
**shows**  $\text{dfree } \vartheta \implies \text{Vagree } \nu \nu' (\text{FVDiff } \vartheta) \implies \text{frechet } I \ \vartheta \ (\text{fst } \nu) \ (\text{snd } \nu) =$   
 $\text{frechet } I \ \vartheta \ (\text{fst } \nu') \ (\text{snd } \nu')$   
 $\langle \text{proof} \rangle$

**lemma** *coincidence-frechet'* :

**fixes**  $I J :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  **and**  $\nu :: 'c \text{ state}$  **and**  $\nu' :: 'c \text{ state}$   
**shows**  $\text{dfree } \vartheta \implies \text{Vagree } \nu \nu' (\text{FVDiff } \vartheta) \implies \text{Iagree } I J \ \{\text{Inl } x \mid x. x \in (\text{SIGT}$   
 $\vartheta)\} \implies \text{frechet } I \ \vartheta \ (\text{fst } \nu) \ (\text{snd } \nu) = \text{frechet } J \ \vartheta \ (\text{fst } \nu') \ (\text{snd } \nu')$   
 $\langle \text{proof} \rangle$

**lemma** *coincidence-dterm*:

**fixes**  $I :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  **and**  $\nu :: 'c \text{ state}$  **and**  $\nu' :: 'c \text{ state}$   
**shows**  $\text{dsafe } \vartheta \implies \text{Vagree } \nu \nu' (\text{FVT } \vartheta) \implies \text{dterm-sem } I \ \vartheta \ \nu = \text{dterm-sem } I \ \vartheta \ \nu'$   
 $\langle \text{proof} \rangle$

**lemma** *coincidence-dterm'*:

**fixes**  $I J :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  **and**  $\nu :: 'c::\text{finite state}$  **and**  
 $\nu' :: 'c::\text{finite state}$   
**shows**  $\text{dsafe } \vartheta \implies \text{Vagree } \nu \nu' (\text{FVT } \vartheta) \implies \text{Iagree } I J \ \{\text{Inl } x \mid x. x \in (\text{SIGT}$   
 $\vartheta)\} \implies \text{dterm-sem } I \ \vartheta \ \nu = \text{dterm-sem } J \ \vartheta \ \nu'$   
 $\langle \text{proof} \rangle$

## 7.2 ODE Coincidence Theorems

**lemma** *coincidence-ode*:

**fixes**  $I J :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  **and**  $\nu :: 'c::\text{finite state}$  **and**  
 $\nu' :: 'c::\text{finite state}$   
**shows**  $\text{osafe ODE} \implies$   
 $\text{Vagree } \nu \nu' (\text{Inl } \text{ FVO ODE}) \implies$   
 $\text{Iagree } I J \ (\{\text{Inl } x \mid x. \text{Inl } x \in \text{SIGO ODE}\} \cup \{\text{Inr } (\text{Inr } x) \mid x. \text{Inr } x \in$   
 $\text{SIGO ODE}\}) \implies$   
 $\text{ODE-sem } I \ \text{ODE} \ (\text{fst } \nu) = \text{ODE-sem } J \ \text{ODE} \ (\text{fst } \nu')$   
 $\langle \text{proof} \rangle$

**lemma** *coincidence-ode'*:

**fixes**  $I J :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  **and**  $\nu :: 'c \text{ simple-state}$  **and**  
 $\nu' :: 'c \text{ simple-state}$   
**shows**  $\text{osafe ODE} \implies$   
 $\text{VSagree } \nu \nu' (\text{FVO ODE}) \implies$   
 $\text{Iagree } I J \ (\{\text{Inl } x \mid x. \text{Inl } x \in \text{SIGO ODE}\} \cup \{\text{Inr } (\text{Inr } x) \mid x. \text{Inr } x \in$   
 $\text{SIGO ODE}\}) \implies$   
 $\text{ODE-sem } I \ \text{ODE} \ \nu = \text{ODE-sem } J \ \text{ODE} \ \nu'$

$\langle \text{proof} \rangle$

**lemma** *alt-sem-lemma*:  $\bigwedge I :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{interp. } \bigwedge \text{ODE} :: ('a::\text{finite}, 'c::\text{finite}) \text{ODE. } \bigwedge \text{sol. } \bigwedge t :: \text{real. } \bigwedge \text{ab. } \text{osafe ODE} \implies$   
 $\text{ODE-sem } I \text{ ODE } (\text{sol } t) = \text{ODE-sem } I \text{ ODE } (\chi \ i. \ \text{if } i \in \text{FVO ODE then sol } t \ \$$   
 $i \ \text{else ab } \$ \ i)$   
 $\langle \text{proof} \rangle$

**lemma** *bvo-to-fvo*:  $\text{Inl } x \in \text{BVO ODE} \implies x \in \text{FVO ODE}$   
 $\langle \text{proof} \rangle$

**lemma** *ode-to-fvo*:  $x \in \text{ODE-vars } I \text{ ODE} \implies x \in \text{FVO ODE}$   
 $\langle \text{proof} \rangle$

**definition** *coincide-hp* ::  $('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{hp} \implies ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{interp} \implies \text{bool}$   
**where** *coincide-hp*  $\alpha \ I \ J \longleftrightarrow (\forall \nu \nu' \mu \ V. \ \text{Iagree } I \ J \ (\text{SIGP } \alpha) \longrightarrow \text{Vagree } \nu \nu' \ V \longrightarrow V \supseteq (\text{FVP } \alpha) \longrightarrow (\nu, \mu) \in \text{prog-sem } I \ \alpha \longrightarrow (\exists \mu'. (\nu', \mu') \in \text{prog-sem } J \ \alpha \wedge \text{Vagree } \mu \ \mu' \ (\text{MBV } \alpha \cup V)))$

**definition** *ode-sem-equiv* ::  $('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{hp} \implies ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{interp} \implies \text{bool}$   
**where** *ode-sem-equiv*  $\alpha \ I \longleftrightarrow$   
 $(\forall \text{ODE} :: ('a::\text{finite}, 'c::\text{finite}) \text{ODE. } \forall \varphi :: ('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{formula. } \text{osafe ODE} \longrightarrow \text{fsafe } \varphi \longrightarrow$   
 $(\alpha = \text{EvolveODE ODE } \varphi) \longrightarrow$   
 $\{(\nu, \text{mk-v } I \text{ ODE } \nu \ (\text{sol } t)) \mid \nu \ \text{sol } t. \ t \geq 0 \wedge$   
 $(\text{sol solves-ode } (\lambda-. \text{ODE-sem } I \text{ ODE})) \{0..t\} \{x. \text{mk-v } I \text{ ODE } \nu \ x \in \text{fml-sem } I \ \varphi\} \wedge$   
 $\text{VSagree } (\text{sol } 0) \ (\text{fst } \nu) \{x \mid x. \text{Inl } x \in \text{FVP } (\text{EvolveODE ODE } \varphi)\}\} =$   
 $\{(\nu, \text{mk-v } I \text{ ODE } \nu \ (\text{sol } t)) \mid \nu \ \text{sol } t. \ t \geq 0 \wedge$   
 $(\text{sol solves-ode } (\lambda-. \text{ODE-sem } I \text{ ODE})) \{0..t\} \{x. \text{mk-v } I \text{ ODE } \nu \ x \in \text{fml-sem } I \ \varphi\} \wedge$   
 $\text{sol } 0 = \text{fst } \nu\})$

**definition** *coincide-hp'* ::  $('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{hp} \implies \text{bool}$   
**where** *coincide-hp'*  $\alpha \longleftrightarrow (\forall \ I \ J. \ \text{coincide-hp } \alpha \ I \ J \wedge \text{ode-sem-equiv } \alpha \ I)$

**definition** *coincide-fml* ::  $('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{formula} \implies \text{bool}$   
**where** *coincide-fml*  $\varphi \longleftrightarrow (\forall \nu \nu' \ I \ J. \ \text{Iagree } I \ J \ (\text{SIGF } \varphi) \longrightarrow \text{Vagree } \nu \nu' \ (\text{FVF } \varphi) \longrightarrow \nu \in \text{fml-sem } I \ \varphi \longleftrightarrow \nu' \in \text{fml-sem } J \ \varphi)$

**lemma** *coinc-fml [simp]*:  $\text{coincide-fml } \varphi = (\forall \nu \nu' \ I \ J. \ \text{Iagree } I \ J \ (\text{SIGF } \varphi) \longrightarrow \text{Vagree } \nu \nu' \ (\text{FVF } \varphi) \longrightarrow \nu \in \text{fml-sem } I \ \varphi \longleftrightarrow \nu' \in \text{fml-sem } J \ \varphi)$   
 $\langle \text{proof} \rangle$

### 7.3 Coincidence Theorems for Programs and Formulas

**lemma** *coincidence-hp-fml*:

**fixes**  $\alpha::('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ hp}$

**fixes**  $\varphi::('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ formula}$

**shows**  $(\text{hpsafe } \alpha \longrightarrow \text{coincide-hp'} \alpha) \wedge (\text{fsafe } \varphi \longrightarrow \text{coincide-fml } \varphi)$   
 $\langle \text{proof} \rangle$

**lemma** *coincidence-formula*:  $\bigwedge \nu \nu' I J. \text{fsafe } (\varphi::('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ formula}) \implies \text{Iagree } I J (\text{SIGF } \varphi) \implies \text{Vagree } \nu \nu' (\text{FVF } \varphi) \implies (\nu \in \text{fml-sem } I \varphi \longleftrightarrow \nu' \in \text{fml-sem } J \varphi)$

$\langle \text{proof} \rangle$

**lemma** *coincidence-hp*:

**fixes**  $\nu \nu' \mu V I J$

**assumes**  $\text{safe:hpsafe } (\alpha::('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ hp})$

**assumes**  $\text{IA:Iagree } I J (\text{SIGP } \alpha)$

**assumes**  $\text{VA:Vagree } \nu \nu' V$

**assumes**  $\text{sub:} V \supseteq (\text{FVP } \alpha)$

**assumes**  $\text{sem:}(\nu, \mu) \in \text{prog-sem } I \alpha$

**shows**  $(\exists \mu'. (\nu', \mu') \in \text{prog-sem } J \alpha \wedge \text{Vagree } \mu \mu' (\text{MBV } \alpha \cup V))$   
 $\langle \text{proof} \rangle$

### 7.4 Corollaries: Alternate ODE semantics definition

**lemma** *ode-sem-eq*:

**fixes**  $I::('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp}$  and  $\text{ODE}::('a, 'c) \text{ ODE}$  and  $\varphi::('a, 'b, 'c) \text{ formula}$

**assumes**  $\text{osafe:osafe } \text{ODE}$

**assumes**  $\text{fsafe:fsafe } \varphi$

**shows**

$(\{(\nu, \text{mk-v } I \text{ ODE } \nu (\text{sol } t)) \mid \nu \text{ sol } t. t \geq 0 \wedge (\text{sol solves-ode } (\lambda-. \text{ODE-sem } I \text{ ODE})) \{0..t\} \{x. \text{mk-v } I \text{ ODE } \nu x \in \text{fml-sem } I \varphi\} \wedge \text{VSagree } (\text{sol } 0) (\text{fst } \nu) \{x \mid x. \text{Inl } x \in \text{FVP } (\text{EvolveODE } \text{ODE } \varphi)\}\}) =$   
 $(\{(\nu, \text{mk-v } I \text{ ODE } \nu (\text{sol } t)) \mid \nu \text{ sol } t. t \geq 0 \wedge (\text{sol solves-ode } (\lambda-. \text{ODE-sem } I \text{ ODE})) \{0..t\} \{x. \text{mk-v } I \text{ ODE } \nu x \in \text{fml-sem } I \varphi\} \wedge (\text{sol } 0) = (\text{fst } \nu)\})$

$\langle \text{proof} \rangle$

**lemma** *ode-alt-sem*:  $\bigwedge I::('a::\text{finite}, 'b::\text{finite}, 'c::\text{finite}) \text{ interp. } \bigwedge \text{ODE}::('a, 'c) \text{ ODE.}$

$\bigwedge \varphi::('a, 'b, 'c) \text{ formula. } \text{osafe } \text{ODE} \implies \text{fsafe } \varphi \implies$

$\text{prog-sem } I (\text{EvolveODE } \text{ODE } \varphi)$

$=$

$\{(\nu, \text{mk-v } I \text{ ODE } \nu (\text{sol } t)) \mid \nu \text{ sol } t. t \geq 0 \wedge (\text{sol solves-ode } (\lambda-. \text{ODE-sem } I \text{ ODE})) \{0..t\} \{x. \text{mk-v } I \text{ ODE } \nu x \in \text{fml-sem}$

```

I  $\varphi$ }  $\wedge$ 
  VSagree (sol 0) (fst  $\nu$ ) {x | x. Inl x  $\in$  FVP (EvolveODE ODE  $\varphi$ )}}

  <proof>
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  $\bigwedge \nu :: 'sz \text{ state. } \bigwedge \omega :: 'sz \text{ state. } \text{hpsafe } \alpha \implies (\nu, \omega) \in \text{prog-sem } I \alpha \implies$ 
  Vagree  $\nu \omega$  ( $-$  (BVP  $\alpha$ ))
  <proof>
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*  $\equiv$  Equals (Differential (\$f fid1 empty)) (Const 0)

**definition** *diff-var-axiom* :: ('sf, 'sc, 'sz) formula

**where** [axiom-defs]:*diff-var-axiom*  $\equiv$  Equals (Differential (Var vid1)) (DiffVar vid1)

**definition** *state-fun* :: 'sf  $\Rightarrow$  ('sf, 'sz) trm

**where** [axiom-defs]:*state-fun*  $f = (\$f f (\lambda i. \text{Var } i))$

**definition** *diff-plus-axiom* :: ('sf, 'sc, 'sz) formula

**where** [axiom-defs]:*diff-plus-axiom*  $\equiv$  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*  $\equiv$  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*  $\equiv$  [[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)  $\rightarrow$  ([[EvolveODE (OVar vid1) (Predicational pid1)]](Predicational pid2))  $\leftrightarrow$



$([[[EvolveODE (OVar vid1) (And (Predicational pid1) (Predicational pid3))]]Predicational pid2))$

**definition**  $DEaxiom :: ('sf, 'sc, 'sz) formula$

**where**  $[axiom-defs]:DEaxiom =$

$(([[[EvolveODE (OSing vid1 (f1 fid1 vid1)) (p1 vid2 vid1)]] (P pid1))$

$\leftrightarrow$

$([[[EvolveODE (OSing vid1 (f1 fid1 vid1)) (p1 vid2 vid1)]]$   
 $[[DiffAssign vid1 (f1 fid1 vid1)]]P pid1))$

**definition**  $DSaxiom :: ('sf, 'sc, 'sz) formula$

**where**  $[axiom-defs]:DSaxiom =$

$(([[[EvolveODE (OSing vid1 (f0 fid1)) (p1 vid2 vid1)]]p1 vid3 vid1)$

$\leftrightarrow$

$(Forall vid2$

$(Implies (Geq (Var vid2) (Const 0))$

$(Implies$

$(Forall vid3$

$(Implies (And (Geq (Var vid3) (Const 0)) (Geq (Var vid2) (Var vid3)))$

$(Prop vid2 (singleton (Plus (Var vid1) (Times (f0 fid1) (Var vid3))))))$

$([[Assign vid1 (Plus (Var vid1) (Times (f0 fid1) (Var vid2)))]p1 vid3 vid1))))$

$— (Q \rightarrow [c \& Q](f(x)' \geq g(x)'))$

$— \rightarrow$

$— ([c \& Q](f(x) \geq g(x))) \dashrightarrow (Q \rightarrow (f(x) \geq g(x)))$

**definition**  $DIGeqaxiom :: ('sf, 'sc, 'sz) formula$

**where**  $[axiom-defs]:DIGeqaxiom =$

$Implies$

$(Implies (Prop vid1 empty) ([[EvolveODE (OVar vid1) (Prop vid1 empty)]](Geq$   
 $(Differential (f1 fid1 vid1)) (Differential (f1 fid2 vid1)))))$

$(Implies$

$(Implies(Prop vid1 empty) (Geq (f1 fid1 vid1) (f1 fid2 vid1)))$

$([[EvolveODE (OVar vid1) (Prop vid1 empty)]](Geq (f1 fid1 vid1) (f1 fid2$   
 $vid1))))$

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

$— (Q \rightarrow [c \& Q](f(x)' \geq g(x)'))$

$— \rightarrow$

$— ([c \& Q](f(x) > g(x))) \leftrightarrow (Q \rightarrow (f(x) > g(x)))$

**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**  $FunctionFrechet I id1 (vec-lambda (\lambda i. sterm-sem I (empty i) (fst \nu)))$

$(vec-lambda(\lambda i. frechet I (empty i) (fst \nu) (snd \nu))) = 0$

$\langle proof \rangle$

**lemma** *constant-deriv-zero:is-interp I  $\implies$  directional-derivative I (\$f id1 empty)*

$\nu = 0$

$\langle proof \rangle$

**theorem** *diff-const-axiom-valid: valid diff-const-axiom*

$\langle proof \rangle$

**theorem** *diff-var-axiom-valid: valid diff-var-axiom*

$\langle proof \rangle$

**theorem** *diff-plus-axiom-valid: valid diff-plus-axiom*

$\langle proof \rangle$

**theorem** *diff-times-axiom-valid: valid diff-times-axiom*

$\langle proof \rangle$

### 9.4 Proofs for ODE Axioms

**lemma** *DW-valid:valid DWaxiom*

$\langle proof \rangle$

**lemma** *DE-lemma*:

**fixes** *ab bb*::'sz simple-state

**and** *sol*::real  $\implies$  'sz simple-state

**and** *I*::('sf, 'sc, 'sz) interp

**shows**

$repr (mk-v I (OSing vid1 (f1 fid1 vid1)) (ab, bb) (sol t)) vid1 (dterm-sem I (f1$   
 $fid1 vid1) (mk-v I (OSing vid1 (f1 fid1 vid1)) (ab, bb) (sol t)))$

$= mk-v I (OSing vid1 (f1 fid1 vid1)) (ab, bb) (sol t)$

*<proof>*

**lemma** *DE-valid:valid DEaxiom*

*<proof>*

**lemma** *ODE-zero: $\bigwedge i. Inl\ i \notin BVO\ ODE \implies Inr\ i \notin BVO\ ODE \implies ODE\text{-sem}\ I$*   
*ODE  $\nu$  \$ i = 0*

*<proof>*

**lemma** *DE-sys-valid:*

**assumes** *disj:{Inl vid1, Inr vid1}  $\cap$  BVO ODE = {}*

**shows** *valid ([[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))*

*<proof>*

**lemma** *DC-valid:valid DCaxiom*

*<proof>*

**lemma** *DS-valid:valid DSaxiom*

*<proof>*

**lemma** *MVT0-within:*

**fixes** *f :: real  $\Rightarrow$  real*

**and** *f' :: real  $\Rightarrow$  real  $\Rightarrow$  real*

**and** *s t :: real*

**assumes** *f': $\bigwedge x. x \in \{0..t\} \implies (f\ \text{has-derivative}\ (f'\ x))\ (at\ x\ \text{within}\ \{0..t\})$*

**assumes** *geq': $\bigwedge x. x \in \{0..t\} \implies f'\ x\ s \geq 0$*

**assumes** *int-s:s > 0  $\wedge$  s  $\leq$  t*

**assumes** *t: 0 < t*

**shows** *f s  $\geq$  f 0*

*<proof>*

**lemma** *MVT':*

**fixes** *f g :: real  $\Rightarrow$  real*

**fixes** *f' g' :: real  $\Rightarrow$  real  $\Rightarrow$  real*

**fixes** *s t :: real*

**assumes** *f': $\bigwedge s. s \in \{0..t\} \implies (f\ \text{has-derivative}\ (f'\ s))\ (at\ s\ \text{within}\ \{0..t\})$*

**assumes** *g': $\bigwedge s. s \in \{0..t\} \implies (g\ \text{has-derivative}\ (g'\ s))\ (at\ s\ \text{within}\ \{0..t\})$*

**assumes** *geq': $\bigwedge x. x \in \{0..t\} \implies f'\ x\ s \geq g'\ x\ s$*

**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>*

**lemma** *MVT'-gr:*

**fixes** *f g :: real  $\Rightarrow$  real*

**fixes**  $f' g' :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}$   
**fixes**  $s t :: \text{real}$   
**assumes**  $f': \bigwedge s. s \in \{0..t\} \Longrightarrow (f \text{ has-derivative } (f' s)) \text{ (at } s \text{ within } \{0..t\})$   
**assumes**  $g': \bigwedge s. s \in \{0..t\} \Longrightarrow (g \text{ has-derivative } (g' s)) \text{ (at } s \text{ within } \{0..t\})$   
**assumes**  $geq': \bigwedge x. x \in \{0..t\} \Longrightarrow f' x s \geq g' x s$   
**assumes**  $geq0: f 0 > g 0$   
**assumes**  $int\text{-}s: s > 0 \wedge s \leq t$   
**assumes**  $t: t > 0$   
**shows**  $f s > g s$   
 $\langle \text{proof} \rangle$

**lemma** *frech-linear*:

**fixes**  $x \vartheta \nu \nu' I$   
**assumes**  $\text{good-interp}: \text{is-interp } I$   
**assumes**  $\text{free}: \text{dfree } \vartheta$   
**shows**  $x * \text{frechet } I \vartheta \nu \nu' = \text{frechet } I \vartheta \nu (x *_R \nu')$   
 $\langle \text{proof} \rangle$

**lemma** *rift-in-space-time*:

**fixes**  $\text{sol } I \text{ ODE } \psi \vartheta t s b$   
**assumes**  $\text{good-interp}: \text{is-interp } I$   
**assumes**  $\text{free}: \text{dfree } \vartheta$   
**assumes**  $\text{osafe}: \text{osafe } \text{ODE}$   
**assumes**  $\text{sol}: (\text{sol solves-ode } (\lambda \nu'. \text{ODE-sem } I \text{ ODE } \nu')) \{0..t\}$   
 $\{x. \text{mk-v } I \text{ ODE } (\text{sol } 0, b) x \in \text{fml-sem } I \psi\}$   
**assumes**  $\text{FVT}: \text{FVT } \vartheta \subseteq \text{semBV } I \text{ ODE}$   
**assumes**  $\text{ivl}: s \in \{0..t\}$   
**shows**  $((\lambda t. \text{stern-sem } I \vartheta (\text{fst } (\text{mk-v } I \text{ ODE } (\text{sol } 0, b) (\text{sol } t))))$   
— This is Frechet derivative, so equivalent to:  
—  $\text{has-real-derivative } \text{frechet } I \vartheta (\text{fst}((\text{mk-v } I \text{ ODE } (\text{sol } 0, b) (\text{sol } s)))) (\text{snd}$   
 $(\text{mk-v } I \text{ ODE } (\text{sol } 0, b) (\text{sol } s)))) (\text{at } s \text{ within } \{0..t\})$   
 $\text{has-derivative } (\lambda t'. t' * \text{frechet } I \vartheta (\text{fst}((\text{mk-v } I \text{ ODE } (\text{sol } 0, b) (\text{sol } s)))) (\text{snd}$   
 $(\text{mk-v } I \text{ ODE } (\text{sol } 0, b) (\text{sol } s)))) (\text{at } s \text{ within } \{0..t\})$   
 $\langle \text{proof} \rangle$

**lemma** *dterm-stern-dfree*:

$\text{dfree } \vartheta \Longrightarrow (\bigwedge \nu \nu'. \text{stern-sem } I \vartheta \nu = \text{dterm-sem } I \vartheta (\nu, \nu'))$   
 $\langle \text{proof} \rangle$

**lemma** *DIGeq-valid:valid DIGeqaxiom*

$\langle \text{proof} \rangle$

**lemma** *DIGr-valid:valid DIGraxiom*

$\langle \text{proof} \rangle$

**lemma** *DG-valid:valid DGaxiom*

$\langle \text{proof} \rangle$

**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 than the input. We use this operation when substituting 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  $\rightarrow$  ('a + 'c, 'c) trm
  SPredicates     :: 'c  $\rightarrow$  ('a + 'c, 'b, 'c) formula
  SContexts      :: 'b  $\rightarrow$  ('a, 'b + unit, 'c) formula
  SPrograms      :: 'c  $\rightarrow$  ('a, 'b, 'c) hp
  SODEs          :: 'c  $\rightarrow$  ('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`  $\sigma$   $\vartheta$   $U \iff ((\bigcup i \in \{i. \text{Inr } i \in \text{SIGT } \vartheta\}. \text{FVT } (\sigma \ i)) \cap U) = \{\}$

**inductive** `TadmitFFO` :: ('d  $\Rightarrow$  ('a, 'c) trm)  $\Rightarrow$  ('a + 'd, 'c) trm  $\Rightarrow$  bool

**where**

`TadmitFFO-Diff`: `TadmitFFO`  $\sigma$   $\vartheta \implies \text{NTUadmit } \sigma \ \vartheta \ \text{UNIV} \implies \text{TadmitFFO } \sigma$  (Differential  $\vartheta$ )

| `TadmitFFO-Fun1`:  $(\bigwedge i. \text{TadmitFFO } \sigma \ (\text{args } i)) \implies \text{TadmitFFO } \sigma \ (\text{Function } (\text{Inl } f) \ \text{args})$

| `TadmitFFO-Fun2`:  $(\bigwedge i. \text{TadmitFFO } \sigma \ (\text{args } i)) \implies \text{dfree } (\sigma \ f) \implies \text{TadmitFFO}$

$\sigma$  (*Function* (*Inr*  $f$ ) *args*)  
| *TadmitFFO-Plus*:  $TadmitFFO \sigma \vartheta 1 \implies TadmitFFO \sigma \vartheta 2 \implies TadmitFFO \sigma$   
(*Plus*  $\vartheta 1 \vartheta 2$ )  
| *TadmitFFO-Times*:  $TadmitFFO \sigma \vartheta 1 \implies TadmitFFO \sigma \vartheta 2 \implies TadmitFFO \sigma$   
(*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*:  $TadmitFFO \sigma \vartheta \implies NTUadmit \sigma \vartheta UNIV \implies dfree$  ( $TsubstFO$   
 $\vartheta \sigma$ )  $\implies TadmitFO \sigma$  (*Differential*  $\vartheta$ )  
| *TadmitFO-Fun*: ( $\bigwedge i. TadmitFO \sigma (args\ i)$ )  $\implies TadmitFO \sigma$  (*Function*  $f$  *args*)  
| *TadmitFO-Plus*:  $TadmitFO \sigma \vartheta 1 \implies TadmitFO \sigma \vartheta 2 \implies TadmitFO \sigma$  (*Plus*  $\vartheta 1$   
 $\vartheta 2$ )  
| *TadmitFO-Times*:  $TadmitFO \sigma \vartheta 1 \implies TadmitFO \sigma \vartheta 2 \implies TadmitFO \sigma$  (*Times*  
 $\vartheta 1 \vartheta 2$ )  
| *TadmitFO-DiffVar*:  $TadmitFO \sigma$  (*DiffVar*  $x$ )  
| *TadmitFO-Var*:  $TadmitFO \sigma$  (*Var*  $x$ )  
| *TadmitFO-Const*:  $TadmitFO \sigma$  (*Const*  $r$ )

### inductive-simps

*TadmitFO-Plus-simps*[*simp*]:  $TadmitFO \sigma$  (*Plus*  $a$   $b$ )  
**and** *TadmitFO-Times-simps*[*simp*]:  $TadmitFO \sigma$  (*Times*  $a$   $b$ )  
**and** *TadmitFO-Var-simps*[*simp*]:  $TadmitFO \sigma$  (*Var*  $x$ )  
**and** *TadmitFO-DiffVar-simps*[*simp*]:  $TadmitFO \sigma$  (*DiffVar*  $x$ )  
**and** *TadmitFO-Differential-simps*[*simp*]:  $TadmitFO \sigma$  (*Differential*  $\vartheta$ )  
**and** *TadmitFO-Const-simps*[*simp*]:  $TadmitFO \sigma$  (*Const*  $r$ )

**and** *TadmitFO-Fun-simps*[simp]: *TadmitFO*  $\sigma$  (*Function*  $i$  *args*)

**primrec** *Tsubst*::('a, 'c) *trm*  $\Rightarrow$  ('a, 'b, 'c) *subst*  $\Rightarrow$  ('a, 'c) *trm*

**where**

*Tsubst* (*Var*  $x$ )  $\sigma$  = *Var*  $x$   
| *Tsubst* (*DiffVar*  $x$ )  $\sigma$  = *DiffVar*  $x$   
| *Tsubst* (*Const*  $r$ )  $\sigma$  = *Const*  $r$   
| *Tsubst* (*Function*  $f$  *args*)  $\sigma$  = (case *SFunctions*  $\sigma$   $f$  of *Some*  $f'$   $\Rightarrow$  *TsubstFO*  $f'$  | *None*  $\Rightarrow$  *Function*  $f$ ) ( $\lambda$   $i$ . *Tsubst* (*args*  $i$ )  $\sigma$ )  
| *Tsubst* (*Plus*  $\vartheta 1$   $\vartheta 2$ )  $\sigma$  = *Plus* (*Tsubst*  $\vartheta 1$   $\sigma$ ) (*Tsubst*  $\vartheta 2$   $\sigma$ )  
| *Tsubst* (*Times*  $\vartheta 1$   $\vartheta 2$ )  $\sigma$  = *Times* (*Tsubst*  $\vartheta 1$   $\sigma$ ) (*Tsubst*  $\vartheta 2$   $\sigma$ )  
| *Tsubst* (*Differential*  $\vartheta$ )  $\sigma$  = *Differential* (*Tsubst*  $\vartheta$   $\sigma$ )

**primrec** *OsubstFO*::('a + 'b, 'c) *ODE*  $\Rightarrow$  ('b  $\Rightarrow$  ('a, 'c) *trm*)  $\Rightarrow$  ('a, 'c) *ODE*

**where**

*OsubstFO* (*OVar*  $c$ )  $\sigma$  = *OVar*  $c$   
| *OsubstFO* (*OSing*  $x$   $\vartheta$ )  $\sigma$  = *OSing*  $x$  (*TsubstFO*  $\vartheta$   $\sigma$ )  
| *OsubstFO* (*OProd* *ODE1* *ODE2*)  $\sigma$  = *OProd* (*OsubstFO* *ODE1*  $\sigma$ ) (*OsubstFO* *ODE2*  $\sigma$ )

**primrec** *Osubst*::('a, 'c) *ODE*  $\Rightarrow$  ('a, 'b, 'c) *subst*  $\Rightarrow$  ('a, 'c) *ODE*

**where**

*Osubst* (*OVar*  $c$ )  $\sigma$  = (case *SODEs*  $\sigma$   $c$  of *Some*  $c'$   $\Rightarrow$   $c'$  | *None*  $\Rightarrow$  *OVar*  $c$ )  
| *Osubst* (*OSing*  $x$   $\vartheta$ )  $\sigma$  = *OSing*  $x$  (*Tsubst*  $\vartheta$   $\sigma$ )  
| *Osubst* (*OProd* *ODE1* *ODE2*)  $\sigma$  = *OProd* (*Osubst* *ODE1*  $\sigma$ ) (*Osubst* *ODE2*  $\sigma$ )

**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*  $a$ )  $\sigma$  = *Pvar*  $a$   
| *PsubstFO* (*Assign*  $x$   $\vartheta$ )  $\sigma$  = *Assign*  $x$  (*TsubstFO*  $\vartheta$   $\sigma$ )  
| *PsubstFO* (*DiffAssign*  $x$   $\vartheta$ )  $\sigma$  = *DiffAssign*  $x$  (*TsubstFO*  $\vartheta$   $\sigma$ )  
| *PsubstFO* (*Test*  $\varphi$ )  $\sigma$  = *Test* (*FsubstFO*  $\varphi$   $\sigma$ )  
| *PsubstFO* (*EvolveODE* *ODE*  $\varphi$ )  $\sigma$  = *EvolveODE* (*OsubstFO* *ODE*  $\sigma$ ) (*FsubstFO*  $\varphi$   $\sigma$ )  
| *PsubstFO* (*Choice*  $\alpha$   $\beta$ )  $\sigma$  = *Choice* (*PsubstFO*  $\alpha$   $\sigma$ ) (*PsubstFO*  $\beta$   $\sigma$ )  
| *PsubstFO* (*Sequence*  $\alpha$   $\beta$ )  $\sigma$  = *Sequence* (*PsubstFO*  $\alpha$   $\sigma$ ) (*PsubstFO*  $\beta$   $\sigma$ )  
| *PsubstFO* (*Loop*  $\alpha$ )  $\sigma$  = *Loop* (*PsubstFO*  $\alpha$   $\sigma$ )  
  
| *FsubstFO* (*Geq*  $\vartheta 1$   $\vartheta 2$ )  $\sigma$  = *Geq* (*TsubstFO*  $\vartheta 1$   $\sigma$ ) (*TsubstFO*  $\vartheta 2$   $\sigma$ )  
| *FsubstFO* (*Prop*  $p$  *args*)  $\sigma$  = *Prop*  $p$  ( $\lambda$   $i$ . *TsubstFO* (*args*  $i$ )  $\sigma$ )  
| *FsubstFO* (*Not*  $\varphi$ )  $\sigma$  = *Not* (*FsubstFO*  $\varphi$   $\sigma$ )  
| *FsubstFO* (*And*  $\varphi$   $\psi$ )  $\sigma$  = *And* (*FsubstFO*  $\varphi$   $\sigma$ ) (*FsubstFO*  $\psi$   $\sigma$ )  
| *FsubstFO* (*Exists*  $x$   $\varphi$ )  $\sigma$  = *Exists*  $x$  (*FsubstFO*  $\varphi$   $\sigma$ )  
| *FsubstFO* (*Diamond*  $\alpha$   $\varphi$ )  $\sigma$  = *Diamond* (*PsubstFO*  $\alpha$   $\sigma$ ) (*FsubstFO*  $\varphi$   $\sigma$ )  
| *FsubstFO* (*InContext*  $C$   $\varphi$ )  $\sigma$  = *InContext*  $C$  (*FsubstFO*  $\varphi$   $\sigma$ )

**fun** *PPsubst*::('a, 'b + 'd, 'c) *hp*  $\Rightarrow$  ('d  $\Rightarrow$  ('a, 'b, 'c) *formula*)  $\Rightarrow$  ('a, 'b, 'c) *hp*

**and** *PFsubst*::('a, 'b + 'd, 'c) *formula*  $\Rightarrow$  ('d  $\Rightarrow$  ('a, 'b, 'c) *formula*)  $\Rightarrow$  ('a, 'b, 'c)

formula

where

$PPsubst (Pvar a) \sigma = Pvar a$   
|  $PPsubst (Assign x \vartheta) \sigma = Assign x \vartheta$   
|  $PPsubst (DiffAssign x \vartheta) \sigma = DiffAssign x \vartheta$   
|  $PPsubst (Test \varphi) \sigma = Test (PFsubst \varphi \sigma)$   
|  $PPsubst (EvolveODE ODE \varphi) \sigma = EvolveODE ODE (PFsubst \varphi \sigma)$   
|  $PPsubst (Choice \alpha \beta) \sigma = Choice (PPsubst \alpha \sigma) (PPsubst \beta \sigma)$   
|  $PPsubst (Sequence \alpha \beta) \sigma = Sequence (PPsubst \alpha \sigma) (PPsubst \beta \sigma)$   
|  $PPsubst (Loop \alpha) \sigma = Loop (PPsubst \alpha \sigma)$   
  
|  $PFsubst (Geq \vartheta1 \vartheta2) \sigma = (Geq \vartheta1 \vartheta2)$   
|  $PFsubst (Prop p args) \sigma = Prop p args$   
|  $PFsubst (Not \varphi) \sigma = Not (PFsubst \varphi \sigma)$   
|  $PFsubst (And \varphi \psi) \sigma = And (PFsubst \varphi \sigma) (PFsubst \psi \sigma)$   
|  $PFsubst (Exists x \varphi) \sigma = Exists x (PFsubst \varphi \sigma)$   
|  $PFsubst (Diamond \alpha \varphi) \sigma = Diamond (PPsubst \alpha \sigma) (PFsubst \varphi \sigma)$   
|  $PFsubst (InContext C \varphi) \sigma = (case C of Inl C' \Rightarrow InContext C' (PFsubst \varphi \sigma)$   
|  $Inr p' \Rightarrow \sigma p')$

**fun**  $Psubst :: ('a, 'b, 'c) hp \Rightarrow ('a, 'b, 'c) subst \Rightarrow ('a, 'b, 'c) hp$

**and**  $Fsubst :: ('a, 'b, 'c) formula \Rightarrow ('a, 'b, 'c) subst \Rightarrow ('a, 'b, 'c) formula$

where

$Psubst (Pvar a) \sigma = (case SPrograms \sigma a of Some a' \Rightarrow a' | None \Rightarrow Pvar a)$   
|  $Psubst (Assign x \vartheta) \sigma = Assign x (Tsubst \vartheta \sigma)$   
|  $Psubst (DiffAssign x \vartheta) \sigma = DiffAssign x (Tsubst \vartheta \sigma)$   
|  $Psubst (Test \varphi) \sigma = Test (Fsubst \varphi \sigma)$   
|  $Psubst (EvolveODE ODE \varphi) \sigma = EvolveODE (Osubst ODE \sigma) (Fsubst \varphi \sigma)$   
|  $Psubst (Choice \alpha \beta) \sigma = Choice (Psubst \alpha \sigma) (Psubst \beta \sigma)$   
|  $Psubst (Sequence \alpha \beta) \sigma = Sequence (Psubst \alpha \sigma) (Psubst \beta \sigma)$   
|  $Psubst (Loop \alpha) \sigma = Loop (Psubst \alpha \sigma)$   
  
|  $Fsubst (Geq \vartheta1 \vartheta2) \sigma = Geq (Tsubst \vartheta1 \sigma) (Tsubst \vartheta2 \sigma)$   
|  $Fsubst (Prop p args) \sigma = (case SPredicates \sigma p of Some p' \Rightarrow FsubstFO p' (\lambda i. Tsubst (args i) \sigma) | None \Rightarrow Prop p (\lambda i. Tsubst (args i) \sigma))$   
|  $Fsubst (Not \varphi) \sigma = Not (Fsubst \varphi \sigma)$   
|  $Fsubst (And \varphi \psi) \sigma = And (Fsubst \varphi \sigma) (Fsubst \psi \sigma)$   
|  $Fsubst (Exists x \varphi) \sigma = Exists x (Fsubst \varphi \sigma)$   
|  $Fsubst (Diamond \alpha \varphi) \sigma = Diamond (Psubst \alpha \sigma) (Fsubst \varphi \sigma)$   
|  $Fsubst (InContext C \varphi) \sigma = (case SContexts \sigma C of Some C' \Rightarrow PFsubst C' (\lambda . (Fsubst \varphi \sigma)) | None \Rightarrow InContext C (Fsubst \varphi \sigma))$

**definition**  $FVA :: ('a \Rightarrow ('a, 'c) trm) \Rightarrow ('c + 'c) set$

**where**  $FVA args = (\bigcup i. FVT (args i))$

**fun**  $SFV :: ('a, 'b, 'c) subst \Rightarrow ('a + 'b + 'c) \Rightarrow ('c + 'c) set$

**where**  $SFV \sigma (Inl i) = (case SFunctions \sigma i of Some f' \Rightarrow FVT f' | None \Rightarrow \{\})$

|  $SFV \sigma (Inr (Inl i)) = \{\}$



|  $SFV \sigma (Inr (Inr i)) = (case \text{SPredicates } \sigma \ i \text{ of } Some \ p' \Rightarrow FVF \ p' \mid None \Rightarrow \{\})$

**definition**  $FVS :: ('a, 'b, 'c) \text{subst} \Rightarrow ('c + 'c) \text{set}$   
**where**  $FVS \sigma = (\bigcup i. SFV \sigma \ i)$

**definition**  $SDom :: ('a, 'b, 'c) \text{subst} \Rightarrow ('a + 'b + 'c) \text{set}$   
**where**  $SDom \sigma =$   
 $\{Inl \ x \mid x. x \in dom (SFunctions \ \sigma)\}$   
 $\cup \{Inr (Inl \ x) \mid x. x \in dom (SContexts \ \sigma)\}$   
 $\cup \{Inr (Inr \ x) \mid x. x \in dom (SPredicates \ \sigma)\}$   
 $\cup \{Inr (Inr \ x) \mid x. x \in dom (SPrograms \ \sigma)\}$

**definition**  $TUadmit :: ('a, 'b, 'c) \text{subst} \Rightarrow ('a, 'c) \text{trm} \Rightarrow ('c + 'c) \text{set} \Rightarrow bool$   
**where**  $TUadmit \sigma \ \vartheta \ U \longleftrightarrow ((\bigcup i \in SIGT \ \vartheta. (case \text{SFunctions } \sigma \ i \text{ of } Some \ f' \Rightarrow FVT \ f' \mid None \Rightarrow \{\})) \cap U) = \{\}$

**inductive**  $Tadmit :: ('a, 'b, 'c) \text{subst} \Rightarrow ('a, 'c) \text{trm} \Rightarrow bool$   
**where**

$Tadmit\text{-Diff}: Tadmit \ \sigma \ \vartheta \Longrightarrow TUadmit \ \sigma \ \vartheta \ UNIV \Longrightarrow Tadmit \ \sigma \ (\text{Differential } \vartheta)$   
 $Tadmit\text{-Fun1}: (\bigwedge i. Tadmit \ \sigma \ (\text{args } i)) \Longrightarrow SFunctions \ \sigma \ f = Some \ f' \Longrightarrow TadmitFO \ (\lambda \ i. Tsubst \ (\text{args } i) \ \sigma) \ f' \Longrightarrow Tadmit \ \sigma \ (\text{Function } f \ \text{args})$   
 $Tadmit\text{-Fun2}: (\bigwedge i. Tadmit \ \sigma \ (\text{args } i)) \Longrightarrow SFunctions \ \sigma \ f = None \Longrightarrow Tadmit \ \sigma \ (\text{Function } f \ \text{args})$   
 $Tadmit\text{-Plus}: Tadmit \ \sigma \ \vartheta 1 \Longrightarrow Tadmit \ \sigma \ \vartheta 2 \Longrightarrow Tadmit \ \sigma \ (\text{Plus } \vartheta 1 \ \vartheta 2)$   
 $Tadmit\text{-Times}: Tadmit \ \sigma \ \vartheta 1 \Longrightarrow Tadmit \ \sigma \ \vartheta 2 \Longrightarrow Tadmit \ \sigma \ (\text{Times } \vartheta 1 \ \vartheta 2)$   
 $Tadmit\text{-DiffVar}: Tadmit \ \sigma \ (\text{DiffVar } x)$   
 $Tadmit\text{-Var}: Tadmit \ \sigma \ (\text{Var } x)$   
 $Tadmit\text{-Const}: Tadmit \ \sigma \ (\text{Const } r)$

**inductive-simps**

$Tadmit\text{-Plus-simps}[simp]: Tadmit \ \sigma \ (\text{Plus } a \ b)$   
**and**  $Tadmit\text{-Times-simps}[simp]: Tadmit \ \sigma \ (\text{Times } a \ b)$   
**and**  $Tadmit\text{-Var-simps}[simp]: Tadmit \ \sigma \ (\text{Var } x)$   
**and**  $Tadmit\text{-DiffVar-simps}[simp]: Tadmit \ \sigma \ (\text{DiffVar } x)$   
**and**  $Tadmit\text{-Differential-simps}[simp]: Tadmit \ \sigma \ (\text{Differential } \vartheta)$   
**and**  $Tadmit\text{-Const-simps}[simp]: Tadmit \ \sigma \ (\text{Const } r)$   
**and**  $Tadmit\text{-Fun-simps}[simp]: Tadmit \ \sigma \ (\text{Function } i \ \text{args})$

**inductive**  $TadmitF :: ('a, 'b, 'c) \text{subst} \Rightarrow ('a, 'c) \text{trm} \Rightarrow bool$   
**where**

$TadmitF\text{-Diff}: TadmitF \ \sigma \ \vartheta \Longrightarrow TUadmit \ \sigma \ \vartheta \ UNIV \Longrightarrow TadmitF \ \sigma \ (\text{Differential } \vartheta)$   
 $TadmitF\text{-Fun1}: (\bigwedge i. TadmitF \ \sigma \ (\text{args } i)) \Longrightarrow SFunctions \ \sigma \ f = Some \ f' \Longrightarrow (\bigwedge i. dfree \ (Tsubst \ (\text{args } i) \ \sigma)) \Longrightarrow TadmitFFO \ (\lambda \ i. Tsubst \ (\text{args } i) \ \sigma) \ f' \Longrightarrow TadmitF \ \sigma \ (\text{Function } f \ \text{args})$   
 $TadmitF\text{-Fun2}: (\bigwedge i. TadmitF \ \sigma \ (\text{args } i)) \Longrightarrow SFunctions \ \sigma \ f = None \Longrightarrow TadmitF \ \sigma \ (\text{Function } f \ \text{args})$   
 $TadmitF\text{-Plus}: TadmitF \ \sigma \ \vartheta 1 \Longrightarrow TadmitF \ \sigma \ \vartheta 2 \Longrightarrow TadmitF \ \sigma \ (\text{Plus } \vartheta 1 \ \vartheta 2)$   
 $TadmitF\text{-Times}: TadmitF \ \sigma \ \vartheta 1 \Longrightarrow TadmitF \ \sigma \ \vartheta 2 \Longrightarrow TadmitF \ \sigma \ (\text{Times } \vartheta 1 \ \vartheta 2)$

| *TadmitF-DiffVar*: *TadmitF*  $\sigma$  (*DiffVar*  $x$ )  
 | *TadmitF-Var*: *TadmitF*  $\sigma$  (*Var*  $x$ )  
 | *TadmitF-Const*: *TadmitF*  $\sigma$  (*Const*  $r$ )

### inductive-simps

*TadmitF-Plus-simps*[*simp*]: *TadmitF*  $\sigma$  (*Plus*  $a$   $b$ )  
**and** *TadmitF-Times-simps*[*simp*]: *TadmitF*  $\sigma$  (*Times*  $a$   $b$ )  
**and** *TadmitF-Var-simps*[*simp*]: *TadmitF*  $\sigma$  (*Var*  $x$ )  
**and** *TadmitF-DiffVar-simps*[*simp*]: *TadmitF*  $\sigma$  (*DiffVar*  $x$ )  
**and** *TadmitF-Differential-simps*[*simp*]: *TadmitF*  $\sigma$  (*Differential*  $\vartheta$ )  
**and** *TadmitF-Const-simps*[*simp*]: *TadmitF*  $\sigma$  (*Const*  $r$ )  
**and** *TadmitF-Fun-simps*[*simp*]: *TadmitF*  $\sigma$  (*Function*  $i$   $args$ )

**inductive** *Oadmit*:: ( $'a, 'b, 'c$ ) *subst*  $\Rightarrow$  ( $'a, 'c$ ) *ODE*  $\Rightarrow$  ( $'c + 'c$ ) *set*  $\Rightarrow$  *bool*  
**where**

*Oadmit-Var*: *Oadmit*  $\sigma$  (*OVar*  $c$ )  $U$   
 | *Oadmit-Sing*: *TUadmit*  $\sigma$   $\vartheta$   $U \Rightarrow$  *TadmitF*  $\sigma$   $\vartheta \Rightarrow$  *Oadmit*  $\sigma$  (*OSing*  $x$   $\vartheta$ )  $U$   
 | *Oadmit-Prod*: *Oadmit*  $\sigma$  *ODE1*  $U \Rightarrow$  *Oadmit*  $\sigma$  *ODE2*  $U \Rightarrow$  *ODE-dom* (*Osubst* *ODE1*  $\sigma$ )  $\cap$  *ODE-dom* (*Osubst* *ODE2*  $\sigma$ ) =  $\{\}$   $\Rightarrow$  *Oadmit*  $\sigma$  (*OProd* *ODE1* *ODE2*)  $U$

### inductive-simps

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

**definition** *PUadmit* :: ( $'a, 'b, 'c$ ) *subst*  $\Rightarrow$  ( $'a, 'b, 'c$ ) *hp*  $\Rightarrow$  ( $'c + 'c$ ) *set*  $\Rightarrow$  *bool*  
**where** *PUadmit*  $\sigma$   $\vartheta$   $U \longleftrightarrow ((\bigcup i \in (SDom \sigma \cap SIGP \vartheta). SFV \sigma i) \cap U) = \{\}$

**definition** *FUadmit* :: ( $'a, 'b, 'c$ ) *subst*  $\Rightarrow$  ( $'a, 'b, 'c$ ) *formula*  $\Rightarrow$  ( $'c + 'c$ ) *set*  $\Rightarrow$  *bool*

**where** *FUadmit*  $\sigma$   $\vartheta$   $U \longleftrightarrow ((\bigcup i \in (SDom \sigma \cap SIGF \vartheta). SFV \sigma i) \cap U) = \{\}$

**definition** *OUadmitFO* :: ( $'d \Rightarrow ('a, 'c)$  *trm*)  $\Rightarrow$  ( $'a + 'd, 'c$ ) *ODE*  $\Rightarrow$  ( $'c + 'c$ ) *set*  $\Rightarrow$  *bool*

**where** *OUadmitFO*  $\sigma$   $\vartheta$   $U \longleftrightarrow ((\bigcup i \in \{i. Inl (Inr i) \in SIGO \vartheta\}. FVT (\sigma i)) \cap U) = \{\}$

**inductive** *OadmitFO* :: ( $'d \Rightarrow ('a, 'c)$  *trm*)  $\Rightarrow$  ( $'a + 'd, 'c$ ) *ODE*  $\Rightarrow$  ( $'c + 'c$ ) *set*  $\Rightarrow$  *bool*

**where**

*OadmitFO-OVar*: *OUadmitFO*  $\sigma$  (*OVar*  $c$ )  $U \Rightarrow$  *OadmitFO*  $\sigma$  (*OVar*  $c$ )  $U$   
 | *OadmitFO-OSing*: *OUadmitFO*  $\sigma$  (*OSing*  $x$   $\vartheta$ )  $U \Rightarrow$  *TadmitFFO*  $\sigma$   $\vartheta \Rightarrow$  *OadmitFO*  $\sigma$  (*OSing*  $x$   $\vartheta$ )  $U$   
 | *OadmitFO-OProd*: *OadmitFO*  $\sigma$  *ODE1*  $U \Rightarrow$  *OadmitFO*  $\sigma$  *ODE2*  $U \Rightarrow$  *OadmitFO*  $\sigma$  (*OProd* *ODE1* *ODE2*)  $U$

### inductive-simps

*OadmitFO-OVar-simps*[*simp*]: *OadmitFO*  $\sigma$  (*OVar*  $a$ )  $U$

**and**  $OadmitFO\text{-}OProd\text{-simps}[simp]$ :  $OadmitFO\ \sigma\ (OProd\ ODE1\ ODE2)\ U$   
**and**  $OadmitFO\text{-}OSing\text{-simps}[simp]$ :  $OadmitFO\ \sigma\ (OSing\ x\ e)\ U$

**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**

$NPadmit\text{-}Pvar$ :  $NPadmit\ \sigma\ (Pvar\ a)$   
 $| NPadmit\text{-}Sequence$ :  $NPadmit\ \sigma\ a \Longrightarrow NPadmit\ \sigma\ b \Longrightarrow PUadmitFO\ \sigma\ b\ (BVP\ (PsubstFO\ a\ \sigma)) \Longrightarrow hpsafe\ (PsubstFO\ a\ \sigma) \Longrightarrow NPadmit\ \sigma\ (Sequence\ a\ b)$   
 $| NPadmit\text{-}Loop$ :  $NPadmit\ \sigma\ a \Longrightarrow PUadmitFO\ \sigma\ a\ (BVP\ (PsubstFO\ a\ \sigma)) \Longrightarrow hpsafe\ (PsubstFO\ a\ \sigma) \Longrightarrow NPadmit\ \sigma\ (Loop\ a)$   
 $| NPadmit\text{-}ODE$ :  $OadmitFO\ \sigma\ ODE\ (BVO\ ODE) \Longrightarrow NFadmit\ \sigma\ \varphi \Longrightarrow FUadmitFO\ \sigma\ \varphi\ (BVO\ ODE) \Longrightarrow fsafe\ (FsubstFO\ \varphi\ \sigma) \Longrightarrow osafe\ (OsubstFO\ ODE\ \sigma) \Longrightarrow NPadmit\ \sigma\ (EvolveODE\ ODE\ \varphi)$   
 $| NPadmit\text{-}Choice$ :  $NPadmit\ \sigma\ a \Longrightarrow NPadmit\ \sigma\ b \Longrightarrow NPadmit\ \sigma\ (Choice\ a\ b)$   
 $| NPadmit\text{-}Assign$ :  $TadmitFO\ \sigma\ \vartheta \Longrightarrow NPadmit\ \sigma\ (Assign\ x\ \vartheta)$   
 $| NPadmit\text{-}DiffAssign$ :  $TadmitFO\ \sigma\ \vartheta \Longrightarrow NPadmit\ \sigma\ (DiffAssign\ x\ \vartheta)$   
 $| NPadmit\text{-}Test$ :  $NFadmit\ \sigma\ \varphi \Longrightarrow NPadmit\ \sigma\ (Test\ \varphi)$

$| NFadmit\text{-}Geg$ :  $TadmitFO\ \sigma\ \vartheta1 \Longrightarrow TadmitFO\ \sigma\ \vartheta2 \Longrightarrow NFadmit\ \sigma\ (Geg\ \vartheta1\ \vartheta2)$   
 $| NFadmit\text{-}Prop$ :  $(\bigwedge i.\ TadmitFO\ \sigma\ (args\ i)) \Longrightarrow NFadmit\ \sigma\ (Prop\ f\ args)$   
 $| NFadmit\text{-}Not$ :  $NFadmit\ \sigma\ \varphi \Longrightarrow NFadmit\ \sigma\ (Not\ \varphi)$   
 $| NFadmit\text{-}And$ :  $NFadmit\ \sigma\ \varphi \Longrightarrow NFadmit\ \sigma\ \psi \Longrightarrow NFadmit\ \sigma\ (And\ \varphi\ \psi)$   
 $| NFadmit\text{-}Exists$ :  $NFadmit\ \sigma\ \varphi \Longrightarrow FUadmitFO\ \sigma\ \varphi\ \{Inl\ x\} \Longrightarrow NFadmit\ \sigma\ (Exists\ x\ \varphi)$   
 $| NFadmit\text{-}Diamond$ :  $NFadmit\ \sigma\ \varphi \Longrightarrow NPadmit\ \sigma\ a \Longrightarrow FUadmitFO\ \sigma\ \varphi\ (BVP\ (PsubstFO\ a\ \sigma)) \Longrightarrow hpsafe\ (PsubstFO\ a\ \sigma) \Longrightarrow NFadmit\ \sigma\ (Diamond\ a\ \varphi)$   
 $| NFadmit\text{-}Context$ :  $NFadmit\ \sigma\ \varphi \Longrightarrow FUadmitFO\ \sigma\ \varphi\ UNIV \Longrightarrow NFadmit\ \sigma\ (InContext\ C\ \varphi)$

**inductive-simps**

$NPadmit\text{-}Pvar\text{-simps}[simp]$ :  $NPadmit\ \sigma\ (Pvar\ a)$   
**and**  $NPadmit\text{-}Sequence\text{-simps}[simp]$ :  $NPadmit\ \sigma\ (a\ ;;\ b)$   
**and**  $NPadmit\text{-}Loop\text{-simps}[simp]$ :  $NPadmit\ \sigma\ (a^{**})$   
**and**  $NPadmit\text{-}ODE\text{-simps}[simp]$ :  $NPadmit\ \sigma\ (EvolveODE\ ODE\ p)$   
**and**  $NPadmit\text{-}Choice\text{-simps}[simp]$ :  $NPadmit\ \sigma\ (a\ \cup\cup\ b)$   
**and**  $NPadmit\text{-}Assign\text{-simps}[simp]$ :  $NPadmit\ \sigma\ (Assign\ x\ e)$   
**and**  $NPadmit\text{-}DiffAssign\text{-simps}[simp]$ :  $NPadmit\ \sigma\ (DiffAssign\ x\ e)$

**and**  $NPAdmit\text{-}Test\text{-}simps[simp]: NPAdmit \sigma (? p)$   
**and**  $NFAdmit\text{-}Geq\text{-}simps[simp]: NFAdmit \sigma (Geq t1 t2)$   
**and**  $NFAdmit\text{-}Prop\text{-}simps[simp]: NFAdmit \sigma (Prop p args)$   
**and**  $NFAdmit\text{-}Not\text{-}simps[simp]: NFAdmit \sigma (Not p)$   
**and**  $NFAdmit\text{-}And\text{-}simps[simp]: NFAdmit \sigma (And p q)$   
**and**  $NFAdmit\text{-}Exists\text{-}simps[simp]: NFAdmit \sigma (Exists x p)$   
**and**  $NFAdmit\text{-}Diamond\text{-}simps[simp]: NFAdmit \sigma (Diamond a p)$   
**and**  $NFAdmit\text{-}Context\text{-}simps[simp]: NFAdmit \sigma (InContext C p)$

**definition**  $PFUadmit :: ('d \Rightarrow ('a, 'b, 'c) formula) \Rightarrow ('a, 'b + 'd, 'c) formula \Rightarrow ('c + 'c) set \Rightarrow bool$   
**where**  $PFUadmit \sigma \vartheta U \longleftrightarrow True$

**definition**  $PPUadmit :: ('d \Rightarrow ('a, 'b, 'c) formula) \Rightarrow ('a, 'b + 'd, 'c) hp \Rightarrow ('c + 'c) set \Rightarrow bool$   
**where**  $PPUadmit \sigma \vartheta U \longleftrightarrow ((\bigcup i. FVF(\sigma i)) \cap U) = \{\}$

**inductive**  $PPadmit :: ('d \Rightarrow ('a, 'b, 'c) formula) \Rightarrow ('a, 'b + 'd, 'c) hp \Rightarrow bool$   
**and**  $PFadmit :: ('d \Rightarrow ('a, 'b, 'c) formula) \Rightarrow ('a, 'b + 'd, 'c) formula \Rightarrow bool$   
**where**

$PPadmit\text{-}Pvar: PPadmit \sigma (Pvar a)$   
 $| PPadmit\text{-}Sequence: PPadmit \sigma a \Longrightarrow PPadmit \sigma b \Longrightarrow PPUadmit \sigma b (BVP (PPsubst a \sigma)) \Longrightarrow hpsafe (PPsubst a \sigma) \Longrightarrow PPadmit \sigma (Sequence a b)$   
 $| PPadmit\text{-}Loop: PPadmit \sigma a \Longrightarrow PPUadmit \sigma a (BVP (PPsubst a \sigma)) \Longrightarrow hpsafe (PPsubst a \sigma) \Longrightarrow PPadmit \sigma (Loop a)$   
 $| PPadmit\text{-}ODE: PFadmit \sigma \varphi \Longrightarrow PFUadmit \sigma \varphi (BVO ODE) \Longrightarrow PPadmit \sigma (EvolveODE ODE \varphi)$   
 $| PPadmit\text{-}Choice: PPadmit \sigma a \Longrightarrow PPadmit \sigma b \Longrightarrow PPadmit \sigma (Choice a b)$   
  
 $| PPadmit\text{-}Assign: PPadmit \sigma (Assign x \vartheta)$   
 $| PPadmit\text{-}DiffAssign: PPadmit \sigma (DiffAssign x \vartheta)$   
 $| PPadmit\text{-}Test: PFadmit \sigma \varphi \Longrightarrow PPadmit \sigma (Test \varphi)$   
  
 $| PFadmit\text{-}Geq: PFadmit \sigma (Geq \vartheta1 \vartheta2)$   
 $| PFadmit\text{-}Prop: PFadmit \sigma (Prop f args)$   
 $| PFadmit\text{-}Not: PFadmit \sigma \varphi \Longrightarrow PFadmit \sigma (Not \varphi)$   
 $| PFadmit\text{-}And: PFadmit \sigma \varphi \Longrightarrow PFadmit \sigma \psi \Longrightarrow PFadmit \sigma (And \varphi \psi)$   
 $| PFadmit\text{-}Exists: PFadmit \sigma \varphi \Longrightarrow PFUadmit \sigma \varphi \{Inl x\} \Longrightarrow PFadmit \sigma (Exists x \varphi)$   
 $| PFadmit\text{-}Diamond: PFadmit \sigma \varphi \Longrightarrow PPadmit \sigma a \Longrightarrow PFUadmit \sigma \varphi (BVP (PPsubst a \sigma)) \Longrightarrow PFadmit \sigma (Diamond a \varphi)$   
 $| PFadmit\text{-}Context: PFadmit \sigma \varphi \Longrightarrow PFUadmit \sigma \varphi UNIV \Longrightarrow PFadmit \sigma (InContext C \varphi)$

**inductive-simps**

$PPadmit\text{-}Pvar\text{-}simps[simp]: PPadmit \sigma (Pvar a)$   
**and**  $PPadmit\text{-}Sequence\text{-}simps[simp]: PPadmit \sigma (a ;; b)$   
**and**  $PPadmit\text{-}Loop\text{-}simps[simp]: PPadmit \sigma (a**)$

**and** *PPadmit-ODE-simps*[simp]: *PPadmit*  $\sigma$  (*EvolveODE* *ODE* *p*)  
**and** *PPadmit-Choice-simps*[simp]: *PPadmit*  $\sigma$  (*a*  $\cup\cup$  *b*)  
**and** *PPadmit-Assign-simps*[simp]: *PPadmit*  $\sigma$  (*Assign* *x* *e*)  
**and** *PPadmit-DiffAssign-simps*[simp]: *PPadmit*  $\sigma$  (*DiffAssign* *x* *e*)  
**and** *PPadmit-Test-simps*[simp]: *PPadmit*  $\sigma$  (*? p*)

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

**inductive** *Padmit*:: ('a, 'b, 'c) *subst*  $\Rightarrow$  ('a, 'b, 'c) *hp*  $\Rightarrow$  *bool*

**and** *Fadmit*:: ('a, 'b, 'c) *subst*  $\Rightarrow$  ('a, 'b, 'c) *formula*  $\Rightarrow$  *bool*

**where**

*Padmit-Pvar*: *Padmit*  $\sigma$  (*Pvar* *a*)  
| *Padmit-Sequence*: *Padmit*  $\sigma$  *a*  $\Rightarrow$  *Padmit*  $\sigma$  *b*  $\Rightarrow$  *PUadmit*  $\sigma$  *b* (*BVP* (*Psubst* *a*  $\sigma$ ))  $\Rightarrow$  *hpsafe* (*Psubst* *a*  $\sigma$ )  $\Rightarrow$  *Padmit*  $\sigma$  (*Sequence* *a* *b*)  
| *Padmit-Loop*: *Padmit*  $\sigma$  *a*  $\Rightarrow$  *PUadmit*  $\sigma$  *a* (*BVP* (*Psubst* *a*  $\sigma$ ))  $\Rightarrow$  *hpsafe* (*Psubst* *a*  $\sigma$ )  $\Rightarrow$  *Padmit*  $\sigma$  (*Loop* *a*)  
| *Padmit-ODE*: *Oadmit*  $\sigma$  *ODE* (*BVO* *ODE*)  $\Rightarrow$  *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *FUadmit*  $\sigma$   $\varphi$  (*BVO* *ODE*)  $\Rightarrow$  *Padmit*  $\sigma$  (*EvolveODE* *ODE*  $\varphi$ )  
| *Padmit-Choice*: *Padmit*  $\sigma$  *a*  $\Rightarrow$  *Padmit*  $\sigma$  *b*  $\Rightarrow$  *Padmit*  $\sigma$  (*Choice* *a* *b*)  
| *Padmit-Assign*: *Tadmit*  $\sigma$   $\vartheta$   $\Rightarrow$  *Padmit*  $\sigma$  (*Assign* *x*  $\vartheta$ )  
| *Padmit-DiffAssign*: *Tadmit*  $\sigma$   $\vartheta$   $\Rightarrow$  *Padmit*  $\sigma$  (*DiffAssign* *x*  $\vartheta$ )  
| *Padmit-Test*: *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *Padmit*  $\sigma$  (*Test*  $\varphi$ )  
  
| *Fadmit-Geq*: *Tadmit*  $\sigma$   $\vartheta1$   $\Rightarrow$  *Tadmit*  $\sigma$   $\vartheta2$   $\Rightarrow$  *Fadmit*  $\sigma$  (*Geq*  $\vartheta1$   $\vartheta2$ )  
| *Fadmit-Prop1*: ( $\bigwedge i. \textit{Tadmit}$   $\sigma$  (*args* *i*))  $\Rightarrow$  *SPredicates*  $\sigma$  *p* = *Some* *p'*  $\Rightarrow$  *NFadmit* ( $\lambda i. \textit{Tsubst}$  (*args* *i*)  $\sigma$ ) *p'*  $\Rightarrow$  ( $\bigwedge i. \textit{dsafe}$  (*Tsubst* (*args* *i*)  $\sigma$ ))  $\Rightarrow$  *Fadmit*  $\sigma$  (*Prop* *p* *args*)  
| *Fadmit-Prop2*: ( $\bigwedge i. \textit{Tadmit}$   $\sigma$  (*args* *i*))  $\Rightarrow$  *SPredicates*  $\sigma$  *p* = *None*  $\Rightarrow$  *Fadmit*  $\sigma$  (*Prop* *p* *args*)  
| *Fadmit-Not*: *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *Fadmit*  $\sigma$  (*Not*  $\varphi$ )  
| *Fadmit-And*: *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *Fadmit*  $\sigma$   $\psi$   $\Rightarrow$  *Fadmit*  $\sigma$  (*And*  $\varphi$   $\psi$ )  
| *Fadmit-Exists*: *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *FUadmit*  $\sigma$   $\varphi$  {*Inl* *x*}  $\Rightarrow$  *Fadmit*  $\sigma$  (*Exists* *x*  $\varphi$ )  
| *Fadmit-Diamond*: *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *Padmit*  $\sigma$  *a*  $\Rightarrow$  *FUadmit*  $\sigma$   $\varphi$  (*BVP* (*Psubst* *a*  $\sigma$ ))  $\Rightarrow$  *hpsafe* (*Psubst* *a*  $\sigma$ )  $\Rightarrow$  *Fadmit*  $\sigma$  (*Diamond* *a*  $\varphi$ )  
| *Fadmit-Context1*: *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *FUadmit*  $\sigma$   $\varphi$  *UNIV*  $\Rightarrow$  *SContexts*  $\sigma$  *C* = *Some* *C'*  $\Rightarrow$  *PFadmit* ( $\lambda -. \textit{Fsubst}$   $\varphi$   $\sigma$ ) *C'*  $\Rightarrow$  *fsafe*(*Fsubst*  $\varphi$   $\sigma$ )  $\Rightarrow$  *Fadmit*  $\sigma$  (*InContext* *C*  $\varphi$ )  
| *Fadmit-Context2*: *Fadmit*  $\sigma$   $\varphi$   $\Rightarrow$  *FUadmit*  $\sigma$   $\varphi$  *UNIV*  $\Rightarrow$  *SContexts*  $\sigma$  *C* = *None*  $\Rightarrow$  *Fadmit*  $\sigma$  (*InContext* *C*  $\varphi$ )

**inductive-simps**

*Padmit-Pvar-simps*[simp]: *Padmit*  $\sigma$  (*Pvar* *a*)  
**and** *Padmit-Sequence-simps*[simp]: *Padmit*  $\sigma$  (*a* ;; *b*)

```

and Padmit-Loop-simps[simp]: Padmit  $\sigma$  (a**)
and Padmit-ODE-simps[simp]: Padmit  $\sigma$  (EvolveODE ODE p)
and Padmit-Choice-simps[simp]: Padmit  $\sigma$  (a  $\cup\cup$  b)
and Padmit-Assign-simps[simp]: Padmit  $\sigma$  (Assign x e)
and Padmit-DiffAssign-simps[simp]: Padmit  $\sigma$  (DiffAssign x e)
and Padmit-Test-simps[simp]: Padmit  $\sigma$  (? p)

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

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

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

definition adjoint :: ('sf, 'sc, 'sz) interp  $\Rightarrow$  ('sf, 'sc, 'sz) subst  $\Rightarrow$  'sz state  $\Rightarrow$  ('sf,
'sc, 'sz) interp
where adjoint I  $\sigma$   $\nu$  =
( $\lambda$ Functions = ( $\lambda$ f. case SFunctions  $\sigma$  f of Some f'  $\Rightarrow$  ( $\lambda$ R. dterm-sem (extendf I R) f'  $\nu$ ) | None  $\Rightarrow$  Functions I f),
 Predicates = ( $\lambda$ p. case SPredicates  $\sigma$  p of Some p'  $\Rightarrow$  ( $\lambda$ R.  $\nu \in$  fml-sem (extendf I R) p') | None  $\Rightarrow$  Predicates I p),
 Contexts = ( $\lambda$ c. case SContexts  $\sigma$  c of Some c'  $\Rightarrow$  ( $\lambda$ R. fml-sem (extendc I R) c') | None  $\Rightarrow$  Contexts I c),
 Programs = ( $\lambda$ a. case SPrograms  $\sigma$  a of Some a'  $\Rightarrow$  prog-sem I a' | None  $\Rightarrow$  Programs I a),
 ODEs = ( $\lambda$ ode. case SODEs  $\sigma$  ode of Some ode'  $\Rightarrow$  ODE-sem I ode' | None  $\Rightarrow$  ODEs I ode),
 ODEBV = ( $\lambda$ ode. case SODEs  $\sigma$  ode of Some ode'  $\Rightarrow$  ODE-vars I ode' | None  $\Rightarrow$  ODEBV I ode)

```

)

**lemma** *dsem-to-ssem:dfree*  $\vartheta \implies dterm\text{-sem } I \vartheta \nu = sterm\text{-sem } I \vartheta (fst \nu)$   
(*proof*)

**definition** *adjointFO*::('sf, 'sc, 'sz) *interp*  $\Rightarrow$  ('d::finite  $\Rightarrow$  ('sf, 'sz) *trm*)  $\Rightarrow$  'sz  
*state*  $\Rightarrow$  ('sf + 'd, 'sc, 'sz) *interp*

**where** *adjointFO* *I*  $\sigma$   $\nu$  =  
( $\lambda$ Functions = ( $\lambda$ f. case f of Inl f'  $\Rightarrow$  Functions I f' | Inr f'  $\Rightarrow$  ( $\lambda$ -. dterm-sem I  
( $\sigma$  f')  $\nu$ )),  
Predicates = Predicates I,  
Contexts = Contexts I,  
Programs = Programs I,  
ODEs = ODEs I,  
ODEBV = ODEBV I  
)

**lemma** *adjoint-free*:

**assumes** *sfree*:( $\bigwedge i$  f'. SFunctions  $\sigma$  i = Some f'  $\implies$  dfree f')  
**shows** *adjoint* *I*  $\sigma$   $\nu$  =  
( $\lambda$ Functions = ( $\lambda$ f. case SFunctions  $\sigma$  f of Some f'  $\Rightarrow$  ( $\lambda$ R. sterm-sem (extendf I  
R) f' (fst  $\nu$ )) | None  $\Rightarrow$  Functions I f),  
Predicates = ( $\lambda$ p. case SPredicates  $\sigma$  p of Some p'  $\Rightarrow$  ( $\lambda$ R.  $\nu \in$  fml-sem (extendf  
I R) p') | None  $\Rightarrow$  Predicates I p),  
Contexts = ( $\lambda$ c. case SContexts  $\sigma$  c of Some c'  $\Rightarrow$  ( $\lambda$ R. fml-sem (extendc I R)  
c') | None  $\Rightarrow$  Contexts I c),  
Programs = ( $\lambda$ a. case SPrograms  $\sigma$  a of Some a'  $\Rightarrow$  prog-sem I a' | None  $\Rightarrow$   
Programs I a),  
ODEs = ( $\lambda$ ode. case SODEs  $\sigma$  ode of Some ode'  $\Rightarrow$  ODE-sem I ode' | None  
 $\Rightarrow$  ODEs I ode),  
ODEBV = ( $\lambda$ ode. case SODEs  $\sigma$  ode of Some ode'  $\Rightarrow$  ODE-vars I ode' | None  
 $\Rightarrow$  ODEBV I ode))  
(*proof*)

**lemma** *adjointFO-free*:( $\bigwedge i$ . dfree ( $\sigma$  i))  $\implies$  (*adjointFO* *I*  $\sigma$   $\nu$  =  
( $\lambda$ Functions = ( $\lambda$ f. case f of Inl f'  $\Rightarrow$  Functions I f' | Inr f'  $\Rightarrow$  ( $\lambda$ -. sterm-sem I  
( $\sigma$  f') (fst  $\nu$ ))),  
Predicates = Predicates I,  
Contexts = Contexts I,  
Programs = Programs I,  
ODEs = ODEs I,  
ODEBV = ODEBV I))  
(*proof*)

**definition** *PFadjoint*::('sf, 'sc, 'sz) *interp*  $\Rightarrow$  ('d::finite  $\Rightarrow$  ('sf, 'sc, 'sz) *formula*)  
 $\Rightarrow$  ('sf, 'sc + 'd, 'sz) *interp*

**where** *PFadjoint* *I*  $\sigma$  =  
( $\lambda$ Functions = Functions I,  
Predicates = Predicates I,

*Contexts* = ( $\lambda f. \text{case } f \text{ of } \text{Inl } f' \Rightarrow \text{Contexts } I f' \mid \text{Inr } f' \Rightarrow (\lambda-. \text{fml-sem } I (\sigma f'))$ ),  
*Programs* = *Programs* *I*,  
*ODEs* = *ODEs* *I*,  
*ODEBV* = *ODEBV* *I*)

**fun** *Ssubst*::('sf, 'sc, 'sz) *sequent*  $\Rightarrow$  ('sf, 'sc, 'sz) *subst*  $\Rightarrow$  ('sf, 'sc, 'sz) *sequent*  
**where** *Ssubst* ( $\Gamma, \Delta$ )  $\sigma$  = (*map* ( $\lambda \varphi. \text{Fsubst } \varphi \sigma$ )  $\Gamma$ , *map* ( $\lambda \varphi. \text{Fsubst } \varphi \sigma$ )  $\Delta$ )

**fun** *Rsubst*::('sf, 'sc, 'sz) *rule*  $\Rightarrow$  ('sf, 'sc, 'sz) *subst*  $\Rightarrow$  ('sf, 'sc, 'sz) *rule*  
**where** *Rsubst* (*SG*, *C*)  $\sigma$  = (*map* ( $\lambda \varphi. \text{Ssubst } \varphi \sigma$ ) *SG*, *Ssubst* *C*  $\sigma$ )

**definition** *Sadmit*::('sf, 'sc, 'sz) *subst*  $\Rightarrow$  ('sf, 'sc, 'sz) *sequent*  $\Rightarrow$  *bool*  
**where** *Sadmit*  $\sigma$  *S*  $\longleftrightarrow$  (( $\forall i. i \geq 0 \longrightarrow i < \text{length } (\text{fst } S) \longrightarrow \text{Fadmit } \sigma (\text{nth } (\text{fst } S) i)$ )  
 $\wedge (\forall i. i \geq 0 \longrightarrow i < \text{length } (\text{snd } S) \longrightarrow \text{Fadmit } \sigma (\text{nth } (\text{snd } S) i)$ ))

**definition** *Radmit*::('sf, 'sc, 'sz) *subst*  $\Rightarrow$  ('sf, 'sc, 'sz) *rule*  $\Rightarrow$  *bool*  
**where** *Radmit*  $\sigma$  *R*  $\longleftrightarrow$  (( $\forall i. i \geq 0 \longrightarrow i < \text{length } (\text{fst } R) \longrightarrow \text{Sadmit } \sigma (\text{nth } (\text{fst } R) i)$ )  
 $\wedge \text{Sadmit } \sigma (\text{snd } R)$ )

**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*:

$f = f' \Longrightarrow p = p' \Longrightarrow c = c' \Longrightarrow PP = PP' \Longrightarrow ode = ode' \Longrightarrow odebv = odebv'$   
 $\Longrightarrow$

( $\langle \text{Functions} = f, \text{Predicates} = p, \text{Contexts} = c, \text{Programs} = PP, \text{ODEs} = ode, \text{ODEBV} = odebv \rangle =$

( $\langle \text{Functions} = f', \text{Predicates} = p', \text{Contexts} = c', \text{Programs} = PP', \text{ODEs} = ode', \text{ODEBV} = odebv' \rangle$ )

*<proof>*)



## 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  $\Rightarrow$  'sf  $\Rightarrow$  ('sf + 'sz,'sz) trm  $\Rightarrow$  'sz state  $\Rightarrow$  'sz Rvec  $\Rightarrow$  ('sz Rvec  $\Rightarrow$  real)

**where**

`extendf-deriv I - (Var i)  $\nu$  x = ( $\lambda$ -. 0)`  
`| extendf-deriv I - (Const r)  $\nu$  x = ( $\lambda$ -. 0)`  
`| extendf-deriv I g (Function f args)  $\nu$  x =`  
`(case f of`  
`Inl ff  $\Rightarrow$  (THE f'.  $\forall$  y. (Functions I ff has-derivative f' y) (at y))`  
`( $\chi$  i. dterm-sem`  
`( $\lambda$ f' -. x $ f'), Predicates =`  
`Predicates I, Contexts = Contexts I, Programs = Programs I,`  
`ODEs = ODEs I, ODEBV = ODEBV I)`  
`(args i)  $\nu$ )  $\circ$`   
`( $\lambda$  $\nu'$ .  $\chi$  ia. extendf-deriv I g (args ia)  $\nu$  x  $\nu'$ )`  
`| Inr ff  $\Rightarrow$  ( $\lambda$   $\nu'$ .  $\nu'$  $ ff))`  
`| extendf-deriv I g (Plus t1 t2)  $\nu$  x = ( $\lambda$  $\nu'$ . (extendf-deriv I g t1  $\nu$  x  $\nu'$ ) +`  
`(extendf-deriv I g t2  $\nu$  x  $\nu'$ ))`  
`| extendf-deriv I g (Times t1 t2)  $\nu$  x =`  
`( $\lambda$  $\nu'$ . ((dterm-sem (extendf I x) t1  $\nu$  * (extendf-deriv I g t2  $\nu$  x  $\nu'$ )))`  
`+ (extendf-deriv I g t1  $\nu$  x  $\nu'$ ) * (dterm-sem (extendf I x) t2  $\nu$ ))`  
`| extendf-deriv I g ($' -)  $\nu$  = undefined`  
`| extendf-deriv I g (Differential -)  $\nu$  = 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 ( $\lambda$ x. dterm-sem (extendf I x) f'  $\nu$ )`

`<proof>`

**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'  $\nu$  x)`

`<proof>`

**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 ( $\lambda$ x. Blinfun (extendf-deriv I i f'  $\nu$  x))`

`<proof>`

**lemma** *extendf-deriv*:

**fixes**  $f'::('sf + 'sz, 'sz) \text{ trm}$  **and**  $I::('sf, 'sc, 'sz) \text{ interp}$   
**assumes** *free:dfree*  $f'$   
**assumes** *good-interp:is-interp*  $I$   
**shows**  $\exists f''. \forall x. ((\lambda R. \text{dterm-sem } (\text{extendf } I R) f' \nu) \text{ has-derivative } (\text{extendf-deriv } I i f f' \nu x)) \text{ (at } x)$   
*<proof>*

**lemma** *adjoint-safe*:

**assumes** *good-interp:is-interp*  $I$   
**assumes** *good-subst*: $(\bigwedge i f'. \text{SFunctions } \sigma i = \text{Some } f' \implies \text{dfree } f')$   
**shows** *is-interp*  $(\text{adjoint } I \sigma \nu)$   
*<proof>*

**lemma** *adjointFO-safe*:

**assumes** *good-interp:is-interp*  $I$   
**assumes** *good-subst*: $(\bigwedge i. \text{dsafe } (\sigma i))$   
**shows** *is-interp*  $(\text{adjointFO } I \sigma \nu)$   
*<proof>*

## 11.2 Lemmas about adjoint interpretations

**lemma** *adjoint-consequence*: $(\bigwedge f'. \text{SFunctions } \sigma f = \text{Some } f' \implies \text{dsafe } f') \implies$   
 $(\bigwedge f'. \text{SPredicates } \sigma f = \text{Some } f' \implies \text{fsafe } f') \implies \text{Vagree } \nu \omega \text{ (FVS } \sigma) \implies$   
 $\text{adjoint } I \sigma \nu = \text{adjoint } I \sigma \omega$   
*<proof>*

**lemma** *SIGT-plus1*: $\text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } (\text{Plus } t1 \ t2). \text{ case } \text{SFunctions } \sigma i \text{ of}$   
 $\text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\})$   
 $\implies \text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } t1. \text{ case } \text{SFunctions } \sigma i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None}$   
 $\Rightarrow \{\})$   
*<proof>*

**lemma** *SIGT-plus2*: $\text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } (\text{Plus } t1 \ t2). \text{ case } \text{SFunctions } \sigma i \text{ of}$   
 $\text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\})$   
 $\implies \text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } t2. \text{ case } \text{SFunctions } \sigma i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None}$   
 $\Rightarrow \{\})$   
*<proof>*

**lemma** *SIGT-times1*: $\text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } (\text{Times } t1 \ t2). \text{ case } \text{SFunctions } \sigma i$   
 $\text{of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\})$   
 $\implies \text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } t1. \text{ case } \text{SFunctions } \sigma i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None}$   
 $\Rightarrow \{\})$   
*<proof>*

**lemma** *SIGT-times2*: $\text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } (\text{Times } t1 \ t2). \text{ case } \text{SFunctions } \sigma i$   
 $\text{of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\})$   
 $\implies \text{Vagree } \nu \omega \text{ (}\bigcup i \in \text{SIGT } t2. \text{ case } \text{SFunctions } \sigma i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None}$

$\Rightarrow \{\}$   
 $\langle proof \rangle$

**lemma** *uadmit-sterm-adjoint'*:

**assumes**  $dsafe: \bigwedge f f'. SFunctions \sigma f = Some f' \implies dsafe f'$   
**assumes**  $fsafe: \bigwedge f f'. SPredicates \sigma f = Some f' \implies fsafe f'$   
**shows**  $Vagree \nu \omega (\bigcup i \in SIGT \vartheta. case SFunctions \sigma i of Some x \Rightarrow FVT x \mid None \Rightarrow \{\}) \implies sterm-sem (adjoint I \sigma \nu) \vartheta = sterm-sem (adjoint I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *uadmit-sterm-adjoint*:

**assumes**  $TUA: TUadmit \sigma \vartheta U$   
**assumes**  $VA: Vagree \nu \omega (-U)$   
**assumes**  $dsafe: \bigwedge f f'. SFunctions \sigma f = Some f' \implies dsafe f'$   
**assumes**  $fsafe: \bigwedge f f'. SPredicates \sigma f = Some f' \implies fsafe f'$   
**shows**  $sterm-sem (adjoint I \sigma \nu) \vartheta = sterm-sem (adjoint I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *uadmit-sterm-ntadjoint'*:

**assumes**  $dsafe: \bigwedge i. dsafe (\sigma i)$   
**shows**  $Vagree \nu \omega ((\bigcup i \in \{i. Inr i \in SIGT \vartheta\}. FVT (\sigma i))) \implies sterm-sem (adjointFO I \sigma \nu) \vartheta = sterm-sem (adjointFO I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *uadmit-sterm-ntadjoint*:

**assumes**  $TUA: NTUadmit \sigma \vartheta U$   
**assumes**  $VA: Vagree \nu \omega (-U)$   
**assumes**  $dsafe: \bigwedge i. dsafe (\sigma i)$   
**assumes**  $good-interp: is-interp I$   
**shows**  $sterm-sem (adjointFO I \sigma \nu) \vartheta = sterm-sem (adjointFO I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *uadmit-dterm-adjoint'*:

**assumes**  $dfree: \bigwedge f f'. SFunctions \sigma f = Some f' \implies dfree f'$   
**assumes**  $fsafe: \bigwedge f f'. SPredicates \sigma f = Some f' \implies fsafe f'$   
**assumes**  $good-interp: is-interp I$   
**shows**  $\bigwedge \nu \omega. Vagree \nu \omega (\bigcup i \in SIGT \vartheta. case SFunctions \sigma i of Some x \Rightarrow FVT x \mid None \Rightarrow \{\}) \implies dsafe \vartheta \implies dterm-sem (adjoint I \sigma \nu) \vartheta = dterm-sem (adjoint I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *uadmit-dterm-adjoint*:

**assumes**  $TUA: TUadmit \sigma \vartheta U$   
**assumes**  $VA: Vagree \nu \omega (-U)$   
**assumes**  $dfree: \bigwedge f f'. SFunctions \sigma f = Some f' \implies dfree f'$   
**assumes**  $fsafe: \bigwedge f f'. SPredicates \sigma f = Some f' \implies fsafe f'$   
**assumes**  $dsafe: dsafe \vartheta$   
**assumes**  $good-interp: is-interp I$   
**shows**  $dterm-sem (adjoint I \sigma \nu) \vartheta = dterm-sem (adjoint I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *uadmit-dterm-ntadjoint'*:

**assumes**  $dfree:\bigwedge i. dsafe (\sigma i)$   
**assumes** *good-interp:is-interp I*  
**shows**  $\bigwedge \nu \omega. Vagree \nu \omega (\bigcup i \in \{i. Inr i \in SIGT \vartheta\}. FVT (\sigma i)) \implies dsafe \vartheta$   
 $\implies dterm-sem (adjointFO I \sigma \nu) \vartheta = dterm-sem (adjointFO I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *uadmit-dterm-ntadjoint*:

**assumes**  $TUA:NTUadmit \sigma \vartheta U$   
**assumes**  $VA:Vagree \nu \omega (-U)$   
**assumes**  $dfree:\bigwedge i. dsafe (\sigma i)$   
**assumes**  $dsafe:dsafe \vartheta$   
**assumes** *good-interp:is-interp I*  
**shows**  $dterm-sem (adjointFO I \sigma \nu) \vartheta = dterm-sem (adjointFO I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**definition**  $ssafe :: ('sf, 'sc, 'sz) subst \Rightarrow bool$

**where**  $ssafe \sigma \equiv$

$(\forall i f'. SFunctions \sigma i = Some f' \longrightarrow dfree f') \wedge$   
 $(\forall f f'. SPredicates \sigma f = Some f' \longrightarrow fsafe f') \wedge$   
 $(\forall f f'. SPrograms \sigma f = Some f' \longrightarrow hpsafe f') \wedge$   
 $(\forall f f'. SODEs \sigma f = Some f' \longrightarrow osafe f') \wedge$   
 $(\forall C C'. SContexts \sigma C = Some C' \longrightarrow fsafe C')$

**lemma** *uadmit-dterm-adjointS*:

**assumes**  $ssafe:ssafe \sigma$   
**assumes** *good-interp:is-interp I*  
**fixes**  $\nu \omega$   
**assumes**  $VA:Vagree \nu \omega (\bigcup i \in SIGT \vartheta. case SFunctions \sigma i of Some x \Rightarrow FVT x \mid None \Rightarrow \{\})$   
**assumes**  $dsafe:dsafe \vartheta$   
**shows**  $dterm-sem (adjoint I \sigma \nu) \vartheta = dterm-sem (adjoint I \sigma \omega) \vartheta$   
 $\langle proof \rangle$

**lemma** *adj-sub-assign-fact*: $\bigwedge i j e. i \in SIGT e \implies j \in (case SFunctions \sigma i of Some x \Rightarrow FVT x \mid None \Rightarrow \{\}) \implies Inl i \in (\{Inl x \mid x. x \in dom (SFunctions \sigma)\} \cup \{Inr (Inl x) \mid x. x \in dom (SContexts \sigma)\} \cup \{Inr (Inr x) \mid x. x \in dom (SPredicates \sigma)\} \cup \{Inr (Inr x) \mid x. x \in dom (SPrograms \sigma)\}) \cap \{Inl x \mid x. x \in SIGT e\}$   
 $\langle proof \rangle$

**lemma** *adj-sub-geq1-fact*: $\bigwedge i j x1 x2. i \in SIGT x1 \implies j \in (case SFunctions \sigma i of Some x \Rightarrow FVT x \mid None \Rightarrow \{\}) \implies Inl i \in (\{Inl x \mid x. x \in dom (SFunctions \sigma)\} \cup \{Inr (Inl x) \mid x. x \in dom (SContexts \sigma)\} \cup \{Inr (Inr x) \mid x. x \in dom (SPredicates \sigma)\} \cup \{Inr (Inr x) \mid x. x \in dom (SPrograms \sigma)\}) \cap \{Inl x \mid x. x \in SIGT x1 \vee x \in SIGT x2\}$   
 $\langle proof \rangle$

**lemma** *adj-sub-geq2-fact*:  $\bigwedge i j x1 x2. i \in \text{SIGT } x2 \implies j \in (\text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \implies \text{Inl } i \in (\{\text{Inl } x \mid x. x \in \text{dom } (\text{SFunctions } \sigma)\} \cup \{\text{Inr } (\text{Inl } x) \mid x. x \in \text{dom } (\text{SContexts } \sigma)\} \cup \{\text{Inr } (\text{Inr } x) \mid x. x \in \text{dom } (\text{SPredicates } \sigma)\} \cup$

$$\{\text{Inr } (\text{Inr } x) \mid x. x \in \text{dom } (\text{SPrograms } \sigma)\}) \cap \{\text{Inl } x \mid x. x \in \text{SIGT } x1 \vee x \in \text{SIGT } x2\}$$

*<proof>*

**lemma** *adj-sub-prop-fact*:  $\bigwedge i j x1 x2 k. i \in \text{SIGT } (x2 k) \implies j \in (\text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \implies \text{Inl } i \in (\{\text{Inl } x \mid x. x \in \text{dom } (\text{SFunctions } \sigma)\} \cup \{\text{Inr } (\text{Inl } x) \mid x. x \in \text{dom } (\text{SContexts } \sigma)\} \cup \{\text{Inr } (\text{Inr } x) \mid x. x \in \text{dom } (\text{SPredicates } \sigma)\} \cup$

$$\{\text{Inr } (\text{Inr } x) \mid x. x \in \text{dom } (\text{SPrograms } \sigma)\}) \cap \text{insert } (\text{Inr } (\text{Inr } x1)) \{\text{Inl } x \mid x. \exists xa. x \in \text{SIGT } (x2 xa)\}$$

*<proof>*

**lemma** *adj-sub-ode-fact*:  $\bigwedge i j x1 x2. \text{Inl } i \in \text{SIGO } x1 \implies j \in (\text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \implies \text{Inl } i \in (\{\text{Inl } x \mid x. x \in \text{dom } (\text{SFunctions } \sigma)\} \cup \{\text{Inr } (\text{Inl } x) \mid x. x \in \text{dom } (\text{SContexts } \sigma)\} \cup \{\text{Inr } (\text{Inr } x) \mid x. x \in \text{dom } (\text{SPredicates } \sigma)\} \cup$

$$\{\text{Inr } (\text{Inr } x) \mid x. x \in \text{dom } (\text{SPrograms } \sigma)\}) \cap (\text{SIGF } x2 \cup \{\text{Inl } x \mid x. \text{Inl } x \in \text{SIGO } x1\} \cup \{\text{Inr } (\text{Inr } x) \mid x. \text{Inr } x \in \text{SIGO } x1\})$$

*<proof>*

**lemma** *adj-sub-assign*:  $\bigwedge e \sigma x. (\bigcup i \in \text{SIGT } e. \text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \subseteq (\bigcup a \in \text{SDom } \sigma \cap \text{SIGP } (x := e). \text{SFV } \sigma a)$

*<proof>*

**lemma** *adj-sub-diff-assign*:  $\bigwedge e \sigma x. (\bigcup i \in \text{SIGT } e. \text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \subseteq (\bigcup a \in \text{SDom } \sigma \cap \text{SIGP } (\text{DiffAssign } x e). \text{SFV } \sigma a)$

*<proof>*

**lemma** *adj-sub-geq1*:  $\bigwedge \sigma x1 x2. (\bigcup i \in \text{SIGT } x1. \text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \subseteq (\bigcup a \in \text{SDom } \sigma \cap \text{SIGF } (\text{Geq } x1 x2). \text{SFV } \sigma a)$

*<proof>*

**lemma** *adj-sub-geq2*:  $\bigwedge \sigma x1 x2. (\bigcup i \in \text{SIGT } x2. \text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \subseteq (\bigcup a \in \text{SDom } \sigma \cap \text{SIGF } (\text{Geq } x1 x2). \text{SFV } \sigma a)$

*<proof>*

**lemma** *adj-sub-prop*:  $\bigwedge \sigma x1 x2 j. (\bigcup i \in \text{SIGT } (x2 j). \text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{Some } x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}) \subseteq (\bigcup a \in \text{SDom } \sigma \cap \text{SIGF } (\$ \varphi x1 x2). \text{SFV } \sigma a)$

*<proof>*

**lemma** *adj-sub-ode*:  $\bigwedge \sigma x1 x2. (\bigcup i \in \{i \mid i. \text{Inl } i \in \text{SIGO } x1\}. \text{case } \text{SFunctions } \sigma \text{ } i \text{ of } \text{None} \Rightarrow \{\} \mid \text{Some } x \Rightarrow \text{FVT } x) \subseteq (\bigcup a \in \text{SDom } \sigma \cap \text{SIGP } (\text{EvolveODE } x1 x2). \text{SFV } \sigma a)$

*<proof>*

**lemma** *uadmit-ode-adjoint'*:

**fixes**  $\sigma I$   
**assumes** *ssafe:ssafe*  $\sigma$   
**assumes** *good-interp:is-interp*  $I$   
**shows**  $\bigwedge \nu \omega. \text{Vagree } \nu \omega (\bigcup i \in \{i \mid i. \text{Inl } i \in \text{SIGO ODE}\}). \text{case SFunctions } \sigma i$   
*of None*  $\Rightarrow \{\}$   $\mid$  *Some*  $x \Rightarrow \text{FVT } x \Rightarrow \text{osafe ODE} \Rightarrow \text{ODE-sem (adjoint } I \sigma \nu)$   
 $\text{ODE} = \text{ODE-sem (adjoint } I \sigma \omega) \text{ ODE}$   
*<proof>*

**lemma** *uadmit-ode-ntadjoint'*:

**fixes**  $\sigma I$   
**assumes** *ssafe:* $\bigwedge i. \text{dsafe } (\sigma i)$   
**assumes** *good-interp:is-interp*  $I$   
**shows**  $\bigwedge \nu \omega. \text{Vagree } \nu \omega (\bigcup y \in \{y. \text{Inl (Inr } y) \in \text{SIGO ODE}\}). \text{FVT } (\sigma y) \Rightarrow$   
 $\text{osafe ODE} \Rightarrow \text{ODE-sem (adjointFO } I \sigma \nu) \text{ ODE} = \text{ODE-sem (adjointFO } I \sigma$   
 $\omega) \text{ ODE}$   
*<proof>*

**lemma** *adjoint-ode-vars*:

**shows**  $\text{ODE-vars (local.adjoint } I \sigma \nu) \text{ ODE} = \text{ODE-vars (local.adjoint } I \sigma \omega)$   
 $\text{ODE}$   
*<proof>*

**lemma** *uadmit-mkv-adjoint*:

**assumes** *ssafe:ssafe*  $\sigma$   
**assumes** *good-interp:is-interp*  $I$   
**assumes**  $\text{VA:Vagree } \nu \omega (\bigcup i \in \{i \mid i. (\text{Inl } i \in \text{SIGO ODE})\}). \text{case SFunctions } \sigma i$   
*of Some*  $x \Rightarrow \text{FVT } x \mid \text{None} \Rightarrow \{\}$   
**assumes** *osafe:osafe*  $\text{ODE}$   
**shows**  $\text{mk-v (adjoint } I \sigma \nu) \text{ ODE} = \text{mk-v (adjoint } I \sigma \omega) \text{ ODE}$   
*<proof>*

**lemma** *adjointFO-ode-vars*:

**shows**  $\text{ODE-vars (adjointFO } I \sigma \nu) \text{ ODE} = \text{ODE-vars (adjointFO } I \sigma \omega) \text{ ODE}$   
*<proof>*

**lemma** *uadmit-mkv-ntadjoint*:

**assumes** *ssafe:* $\bigwedge i. \text{dsafe } (\sigma i)$   
**assumes** *good-interp:is-interp*  $I$   
**assumes**  $\text{VA:Vagree } \nu \omega (\bigcup y \in \{y. \text{Inl (Inr } y) \in \text{SIGO ODE}\}). \text{FVT } (\sigma y)$   
**assumes** *osafe:osafe*  $\text{ODE}$   
**shows**  $\text{mk-v (adjointFO } I \sigma \nu) \text{ ODE} = \text{mk-v (adjointFO } I \sigma \omega) \text{ ODE}$   
*<proof>*

**lemma** *uadmit-prog-fml-adjoint'*:

**fixes**  $\sigma I$   
**assumes** *ssafe:ssafe*  $\sigma$   
**assumes** *good-interp:is-interp*  $I$

**shows**  $\bigwedge \nu \omega. \text{Vagree } \nu \omega (\bigcup x \in \text{SDom } \sigma \cap \text{SIGP } \alpha. \text{SFV } \sigma x) \implies \text{hpsafe } \alpha \implies$   
 $\text{prog-sem } (\text{adjoint } I \sigma \nu) \alpha = \text{prog-sem } (\text{adjoint } I \sigma \omega) \alpha$   
**and**  $\bigwedge \nu \omega. \text{Vagree } \nu \omega (\bigcup x \in \text{SDom } \sigma \cap \text{SIGF } \varphi. \text{SFV } \sigma x) \implies \text{fsafe } \varphi \implies$   
 $\text{fml-sem } (\text{adjoint } I \sigma \nu) \varphi = \text{fml-sem } (\text{adjoint } I \sigma \omega) \varphi$   
 $\langle \text{proof} \rangle$

**lemma** *uadmit-prog-adjoint*:  
**assumes**  $\text{PUA}:\text{PUadmit } \sigma a U$   
**assumes**  $\text{VA}:\text{Vagree } \nu \omega (-U)$   
**assumes**  $\text{hpsafe}:\text{hpsafe } a$   
**assumes**  $\text{ssafe}:\text{ssafe } \sigma$   
**assumes**  $\text{good-interp}:\text{is-interp } I$   
**shows**  $\text{prog-sem } (\text{adjoint } I \sigma \nu) a = \text{prog-sem } (\text{adjoint } I \sigma \omega) a$   
 $\langle \text{proof} \rangle$

**lemma** *uadmit-fml-adjoint*:  
**assumes**  $\text{FUA}:\text{FUadmit } \sigma \varphi U$   
**assumes**  $\text{VA}:\text{Vagree } \nu \omega (-U)$   
**assumes**  $\text{fsafe}:\text{fsafe } \varphi$   
**assumes**  $\text{ssafe}:\text{ssafe } \sigma$   
**assumes**  $\text{good-interp}:\text{is-interp } I$   
**shows**  $\text{fml-sem } (\text{adjoint } I \sigma \nu) \varphi = \text{fml-sem } (\text{adjoint } I \sigma \omega) \varphi$   
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

**lemma** *uadmit-prog-fml-ntadjoint'*:

**fixes**  $\sigma I$   
**assumes**  $ssafe:\bigwedge i. dsafe (\sigma i)$   
**assumes**  $good\text{-interp:is\text{-}interp } I$   
**shows**  $\bigwedge \nu \omega. Vagree \nu \omega (\bigcup x \in \{x. Inl (Inr x) \in SIGP \alpha\}. FVT (\sigma x)) \implies$   
 $hpsafe \alpha \implies prog\text{-sem} (adjointFO I \sigma \nu) \alpha = prog\text{-sem} (adjointFO I \sigma \omega) \alpha$   
**and**  $\bigwedge \nu \omega. Vagree \nu \omega (\bigcup x \in \{x. Inl (Inr x) \in SIGF \varphi\}. FVT (\sigma x)) \implies fsafe$   
 $\varphi \implies fml\text{-sem} (adjointFO I \sigma \nu) \varphi = fml\text{-sem} (adjointFO I \sigma \omega) \varphi$   
 $\langle proof \rangle$

**lemma**  $uadmit\text{-prog\text{-}ntadjoint}$ :  
**assumes**  $TUA:PUadmitFO \sigma \alpha U$   
**assumes**  $VA:Vagree \nu \omega (-U)$   
**assumes**  $dfree:\bigwedge i. dsafe (\sigma i)$   
**assumes**  $hpsafe:hpsafe \alpha$   
**assumes**  $good\text{-interp:is\text{-}interp } I$   
**shows**  $prog\text{-sem} (adjointFO I \sigma \nu) \alpha = prog\text{-sem} (adjointFO I \sigma \omega) \alpha$   
 $\langle proof \rangle$

**lemma**  $uadmit\text{-fml\text{-}ntadjoint}$ :  
**assumes**  $TUA:FUadmitFO \sigma \varphi U$   
**assumes**  $VA:Vagree \nu \omega (-U)$   
**assumes**  $dfree:\bigwedge i. dsafe (\sigma i)$   
**assumes**  $fsafe:fsafe \varphi$   
**assumes**  $good\text{-interp:is\text{-}interp } I$   
**shows**  $fml\text{-sem} (adjointFO I \sigma \nu) \varphi = fml\text{-sem} (adjointFO I \sigma \omega) \varphi$   
 $\langle proof \rangle$

### 11.3 Substitution theorems for terms

**lemma**  $nsubst\text{-stern}$ :  
**fixes**  $I::('sf, 'sc, 'sz) \text{interp}$   
**fixes**  $\nu::'sz \text{state}$   
**shows**  $TadmitFFO \sigma \vartheta \implies (\bigwedge i. dsafe (\sigma i)) \implies stern\text{-sem } I (TsubstFO \vartheta \sigma)$   
 $(fst \nu) = stern\text{-sem} (adjointFO I \sigma \nu) \vartheta (fst \nu)$   
 $\langle proof \rangle$

**lemma**  $nsubst\text{-stern}'$ :  
**fixes**  $I::('sf, 'sc, 'sz) \text{interp}$   
**fixes**  $a b::'sz \text{simple\text{-}state}$   
**shows**  $TadmitFFO \sigma \vartheta \implies (\bigwedge i. dsafe (\sigma i)) \implies stern\text{-sem } I (TsubstFO \vartheta \sigma)$   
 $a = stern\text{-sem} (adjointFO I \sigma (a,b)) \vartheta a$   
 $\langle proof \rangle$

**lemma**  $ntsubst\text{-preserves\text{-}free}$ :  
 $dfree \vartheta \implies (\bigwedge i. dfree (\sigma i)) \implies dfree(TsubstFO \vartheta \sigma)$   
 $\langle proof \rangle$

**lemma**  $tsubst\text{-preserves\text{-}free}$ :  
 $dfree \vartheta \implies (\bigwedge i f'. SFunctions \sigma i = Some f' \implies dfree f') \implies dfree(Tsubst \vartheta \sigma)$



*<proof>*

**lemma** *subst-sterm*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp*

**fixes**  $\nu::'sz$  *state*

**shows**

$TadmitF \sigma \vartheta \implies$

$(\bigwedge i f'. SFunctions \sigma i = Some f' \implies dfree f') \implies$

$sterm-sem I (Tsubst \vartheta \sigma) (fst \nu) = sterm-sem (adjoint I \sigma \nu) \vartheta (fst \nu)$

*<proof>*

**lemma** *nsubst-dterm'*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp*

**fixes**  $\nu::'sz$  *state*

**assumes** *good-interp:is-interp I*

**shows**  $TadmitFO \sigma \vartheta \implies dfree \vartheta \implies (\bigwedge i. dsafe (\sigma i)) \implies dterm-sem I$

$(TsubstFO \vartheta \sigma) \nu = dterm-sem (adjointFO I \sigma \nu) \vartheta \nu$

*<proof>*

**lemma** *ntsubst-free-to-safe*:

$dfree \vartheta \implies (\bigwedge i. dsafe (\sigma i)) \implies dsafe (TsubstFO \vartheta \sigma)$

*<proof>*

**lemma** *ntsubst-preserves-safe*:

$dsafe \vartheta \implies (\bigwedge i. dfree (\sigma i)) \implies dsafe (TsubstFO \vartheta \sigma)$

*<proof>*

**lemma** *tsubst-preserves-safe*:

$dsafe \vartheta \implies (\bigwedge i f'. SFunctions \sigma i = Some f' \implies dfree f') \implies dsafe(Tsubst \vartheta \sigma)$

*<proof>*

**lemma** *subst-dterm*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp*

**assumes** *good-interp:is-interp I*

**shows**

$Tadmit \sigma \vartheta \implies$

$dsafe \vartheta \implies$

$(\bigwedge i f'. SFunctions \sigma i = Some f' \implies dfree f') \implies$

$(\bigwedge f f'. SPredicates \sigma f = Some f' \implies fsafe f') \implies$

$(\bigwedge \nu. dterm-sem I (Tsubst \vartheta \sigma) \nu = dterm-sem (adjoint I \sigma \nu) \vartheta \nu)$

*<proof>*

## 11.4 Substitution theorems for ODEs

**lemma** *osubst-preserves-safe*:

**assumes** *ssafe:ssafe  $\sigma$*

**shows**  $(osafe ODE \implies Oadmit \sigma ODE U \implies osafe (Osubst ODE \sigma))$

*<proof>*

**lemma** *nosubst-preserves-safe*:  
**assumes** *sfree*: $\bigwedge i. \text{dfree } (\sigma \ i)$   
**fixes**  $\alpha :: ('a + 'd, 'b, 'c) \text{ hp}$  **and**  $\varphi :: ('a + 'd, 'b, 'c) \text{ formula}$   
**shows**  $(\text{osafe } \text{ODE} \implies \text{OUadmitFO } \sigma \ \text{ODE } U \implies \text{osafe } (\text{OsubstFO } \text{ODE } \sigma))$   
 $\langle \text{proof} \rangle$

**lemma** *nosubst-dterm*:  
**fixes**  $I :: ('sf, 'sc, 'sz) \text{ interp}$   
**fixes**  $\nu :: 'sz \text{ state}$   
**fixes**  $\nu' :: 'sz \text{ state}$   
**assumes** *good-interp:is-interp*  $I$   
**shows**  $\text{TadmitFO } \sigma \ \vartheta \implies \text{dsafe } \vartheta \implies (\bigwedge i. \text{dsafe } (\sigma \ i)) \implies \text{dterm-sem } I$   
 $(\text{TsubstFO } \vartheta \ \sigma) \ \nu = \text{dterm-sem } (\text{adjoinFO } I \ \sigma \ \nu) \ \vartheta \ \nu$   
 $\langle \text{proof} \rangle$

**lemma** *nosubst-ode*:  
**fixes**  $I :: ('sf, 'sc, 'sz) \text{ interp}$   
**fixes**  $\nu :: 'sz \text{ state}$   
**fixes**  $\nu' :: 'sz \text{ state}$   
**assumes** *good-interp:is-interp*  $I$   
**shows**  $\text{osafe } \text{ODE} \implies \text{OadmitFO } \sigma \ \text{ODE } U \implies (\bigwedge i. \text{dsafe } (\sigma \ i)) \implies \text{ODE-sem}$   
 $I \ (\text{OsubstFO } \text{ODE } \sigma) \ (\text{fst } \nu) = \text{ODE-sem } (\text{adjoinFO } I \ \sigma \ \nu) \ \text{ODE } (\text{fst } \nu)$   
 $\langle \text{proof} \rangle$

**lemma** *osubst-preserves-BVO*:  
**shows**  $\text{BVO } (\text{OsubstFO } \text{ODE } \sigma) = \text{BVO } \text{ODE}$   
 $\langle \text{proof} \rangle$

**lemma** *osubst-preserves-ODE-vars*:  
**shows**  $\text{ODE-vars } I \ (\text{OsubstFO } \text{ODE } \sigma) = \text{ODE-vars } (\text{adjoinFO } I \ \sigma \ \nu) \ \text{ODE}$   
 $\langle \text{proof} \rangle$

**lemma** *osubst-preserves-semBV*:  
**shows**  $\text{semBV } I \ (\text{OsubstFO } \text{ODE } \sigma) = \text{semBV } (\text{adjoinFO } I \ \sigma \ \nu) \ \text{ODE}$   
 $\langle \text{proof} \rangle$

**lemma** *nosubst-mkv*:  
**fixes**  $I :: ('sf, 'sc, 'sz) \text{ interp}$   
**fixes**  $\nu :: 'sz \text{ state}$   
**fixes**  $\nu' :: 'sz \text{ state}$   
**assumes** *good-interp:is-interp*  $I$   
**assumes** *NOU*: $\text{OadmitFO } \sigma \ \text{ODE } U$   
**assumes** *osafe:osafe*  $\text{ODE}$   
**assumes** *frees*: $(\bigwedge i. \text{dsafe } (\sigma \ i))$   
**shows**  $(\text{mk-v } I \ (\text{OsubstFO } \text{ODE } \sigma) \ \nu \ (\text{fst } \nu'))$   
 $= (\text{mk-v } (\text{adjoinFO } I \ \sigma \ \nu') \ \text{ODE } \nu \ (\text{fst } \nu'))$   
 $\langle \text{proof} \rangle$

**lemma** *ODE-unbound-zero*:

**fixes**  $i$   
**shows**  $Inl\ i \notin BVO\ ODE \implies ODE\text{-sem}\ I\ ODE\ x\ \$\ i = 0$   
 $\langle proof \rangle$

**lemma** *ODE-bound-effect*:  
**fixes**  $s\ t\ sol\ ODE\ X\ b$   
**assumes**  $s:s \in \{0..t\}$   
**assumes**  $sol:(sol\ solves\ ode\ (\lambda\ \cdot.\ ODE\text{-sem}\ I\ ODE))\ \{0..t\}\ X$   
**shows**  $Vagree\ (sol\ 0,b)\ (sol\ s,\ b)\ (\neg(BVO\ ODE))$   
 $\langle proof \rangle$

**lemma** *NO-sub:OadmitFO*  $\sigma\ ODE\ A \implies B \subseteq A \implies OadmitFO\ \sigma\ ODE\ B$   
 $\langle proof \rangle$

**lemma** *NO-to-NOU:OadmitFO*  $\sigma\ ODE\ S \implies OUadmitFO\ \sigma\ ODE\ S$   
 $\langle proof \rangle$

## 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**  $(NPadmit\ \sigma\ \alpha \longrightarrow (hpsafe\ \alpha \longrightarrow (\forall i.\ dsafe\ (\sigma\ i)) \longrightarrow (\forall \nu\ \omega.\ ((\nu,\ \omega) \in prog\text{-sem}\ I\ (PsubstFO\ \alpha\ \sigma)) = ((\nu,\ \omega) \in prog\text{-sem}\ (adjointFO\ I\ \sigma\ \nu)\ \alpha)))) \wedge$   
 $(NFadmit\ \sigma\ \varphi \longrightarrow (fsafe\ \varphi \longrightarrow (\forall i.\ dsafe\ (\sigma\ i)) \longrightarrow (\forall \nu.\ (\nu \in fml\text{-sem}\ I\ (FsubstFO\ \varphi\ \sigma)) = (\nu \in fml\text{-sem}\ (adjointFO\ I\ \sigma\ \nu)\ \varphi))))$   
 $\langle proof \rangle$

**lemma** *nsubst-fml*:  
**fixes**  $I::('sf,\ 'sc,\ 'sz)\ interp$   
**fixes**  $\nu::'sz\ state$   
**assumes** *good-interp:is-interp*  $I$   
**assumes** *NFA:NFadmit*  $\sigma\ \varphi$   
**assumes** *fsafe:fsafe*  $\varphi$   
**assumes** *frees*: $(\forall i.\ dsafe\ (\sigma\ i))$   
**shows**  $(\nu \in fml\text{-sem}\ I\ (FsubstFO\ \varphi\ \sigma)) = (\nu \in fml\text{-sem}\ (adjointFO\ I\ \sigma\ \nu)\ \varphi)$   
 $\langle proof \rangle$

**lemma** *nsubst-hp*:  
**fixes**  $I::('sf,\ 'sc,\ 'sz)\ interp$   
**fixes**  $\nu::'sz\ state$   
**assumes** *good-interp:is-interp*  $I$   
**assumes** *NPA:NPadmit*  $\sigma\ \alpha$   
**assumes** *hpsafe:hpsafe*  $\alpha$   
**assumes** *frees*: $\bigwedge i.\ dsafe\ (\sigma\ i)$   
**shows**  $((\nu,\ \omega) \in prog\text{-sem}\ I\ (PsubstFO\ \alpha\ \sigma)) = ((\nu,\ \omega) \in prog\text{-sem}\ (adjointFO\ I\ \sigma\ \nu)\ \alpha)$   
 $\langle proof \rangle$

**lemma** *psubst-sterm*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp*

**assumes** *good-interp:is-interp I*

**shows** (*stern-sem I*  $\vartheta = \text{stern-sem } (PF\text{adjoint } I \sigma) \vartheta$ )

*<proof>*

**lemma** *psubst-dterm*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp*

**assumes** *good-interp:is-interp I*

**shows** (*dsafe*  $\vartheta \implies \text{dterm-sem } I \vartheta = \text{dterm-sem } (PF\text{adjoint } I \sigma) \vartheta$ )

*<proof>*

**lemma** *psubst-ode*:

**assumes** *good-interp:is-interp I*

**shows** *ODE-sem I ODE = ODE-sem (PFadjoint I sigma) ODE*

*<proof>*

**lemma** *psubst-fml*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp*

**assumes** *good-interp:is-interp I*

**shows** ( $PP\text{admit } \sigma \alpha \longrightarrow \text{hpsafe } \alpha \longrightarrow (\forall i. \text{fsafe } (\sigma i)) \longrightarrow (\forall \nu \omega. (\nu, \omega) \in \text{prog-sem } I (PP\text{subst } \alpha \sigma) = ((\nu, \omega) \in \text{prog-sem } (PF\text{adjoint } I \sigma) \alpha)) \wedge$

$(PF\text{admit } \sigma \varphi \longrightarrow \text{fsafe } \varphi \longrightarrow (\forall i. \text{fsafe } (\sigma i)) \longrightarrow (\forall \nu. \nu \in \text{fml-sem } I (PF\text{subst } \varphi \sigma) = (\nu \in \text{fml-sem } (PF\text{adjoint } I \sigma) \varphi))$ )

*<proof>*

**lemma** *subst-ode*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp* **and**  $\nu :: 'sz$  *state*

**assumes** *good-interp:is-interp I*

**shows** *osafe ODE*  $\implies$

*ssafe*  $\sigma \implies$

*Oadmit*  $\sigma$  *ODE (BVO ODE)*  $\implies$

*ODE-sem I (Osubst ODE sigma) (fst nu) = ODE-sem (adjoint I sigma nu) ODE (fst*

*nu)*

*<proof>*

**lemma** *osubst-eq-ODE-vars*: *ODE-vars I (Osubst ODE sigma) = ODE-vars (adjoint I sigma nu) ODE*

*<proof>*

**lemma** *subst-semBV:semBV (adjoint I sigma nu') ODE = (semBV I (Osubst ODE sigma))*

*<proof>*

**lemma** *subst-mkv*:

**fixes**  $I::('sf, 'sc, 'sz)$  *interp*

**fixes**  $\nu :: 'sz$  *state*

**fixes**  $\nu' :: 'sz$  *state*

**assumes** *good-interp:is-interp I*

**assumes**  $NOU:Oadmit\ \sigma\ ODE\ (BVO\ ODE)$   
**assumes**  $osafe:osafe\ ODE$   
**assumes**  $frees:ssafe\ \sigma$   
**shows**  $(mk-v\ I\ (Osubst\ ODE\ \sigma)\ \nu\ (fst\ \nu'))$   
 $= (mk-v\ (adjoint\ I\ \sigma\ \nu')\ ODE\ \nu\ (fst\ \nu'))$   
 $\langle proof \rangle$

**lemma** *subst-fml-hp*:

**fixes**  $I::('sf, 'sc, 'sz)\ interp$   
**assumes**  $good-interp:is-interp\ I$   
**shows**  
 $(Padmit\ \sigma\ \alpha \longrightarrow$   
 $(hpsafe\ \alpha \longrightarrow$   
 $ssafe\ \sigma \longrightarrow$   
 $(\forall\ \nu\ \omega. ((\nu, \omega) \in prog-sem\ I\ (Psubst\ \alpha\ \sigma)) = ((\nu, \omega) \in prog-sem\ (adjoint\ I\ \sigma\ \nu)\ \alpha))))$   
 $\wedge$   
 $(Fadmit\ \sigma\ \varphi \longrightarrow$   
 $(fsafe\ \varphi \longrightarrow$   
 $ssafe\ \sigma \longrightarrow$   
 $(\forall\ \nu. (\nu \in fml-sem\ I\ (Fsubst\ \varphi\ \sigma)) = (\nu \in fml-sem\ (adjoint\ I\ \sigma\ \nu)\ \varphi))))$   
 $\langle proof \rangle$

**lemma** *subst-fml*:

**fixes**  $I::('sf, 'sc, 'sz)\ interp$  **and**  $\nu::'sz\ state$   
**assumes**  $good-interp:is-interp\ I$   
**assumes**  $Fadmit:Fadmit\ \sigma\ \varphi$   
**assumes**  $fsafe:fsafe\ \varphi$   
**assumes**  $ssafe:ssafe\ \sigma$   
**shows**  $(\nu \in fml-sem\ I\ (Fsubst\ \varphi\ \sigma)) = (\nu \in fml-sem\ (adjoint\ I\ \sigma\ \nu)\ \varphi)$   
 $\langle proof \rangle$

**lemma** *subst-fml-valid*:

**fixes**  $I::('sf, 'sc, 'sz)\ interp$  **and**  $\nu::'sz\ state$   
**assumes**  $Fadmit:Fadmit\ \sigma\ \varphi$   
**assumes**  $fsafe:fsafe\ \varphi$   
**assumes**  $ssafe:ssafe\ \sigma$   
**assumes**  $valid:valid\ \varphi$   
**shows**  $valid\ (Fsubst\ \varphi\ \sigma)$   
 $\langle proof \rangle$

**lemma** *subst-sequent*:

**fixes**  $I::('sf, 'sc, 'sz)\ interp$  **and**  $\nu::'sz\ state$   
**assumes**  $good-interp:is-interp\ I$   
**assumes**  $Sadmit:Sadmit\ \sigma\ (\Gamma, \Delta)$   
**assumes**  $Ssafe:Ssafe\ (\Gamma, \Delta)$   
**assumes**  $ssafe:ssafe\ \sigma$   
**shows**  $(\nu \in seq-sem\ I\ (Ssubst\ (\Gamma, \Delta)\ \sigma)) = (\nu \in seq-sem\ (adjoint\ I\ \sigma\ \nu)\ (\Gamma, \Delta))$

*<proof>*

## 11.6 Soundness of substitution rule

**theorem** *subst-rule*:

**assumes** *sound:sound*  $R$   
**assumes** *Radmit:Radmit*  $\sigma$   $R$   
**assumes** *FVS:FVS*  $\sigma = \{\}$   
**assumes** *Rsafe:Rsafe*  $R$   
**assumes** *ssafe:ssafe*  $\sigma$   
**shows** *sound* ( $Rsubst$   $R$   $\sigma$ )

*<proof>*

**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* ::  $'sz \Rightarrow 'sz \Rightarrow 'sz \Rightarrow 'sz$   
**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* ::  $'sz \Rightarrow 'sz \Rightarrow ('sf, 'sz)$  *trm*  $\Rightarrow ('sf, 'sz)$  *trm*  
**where**

$TUrename$   $x$   $y$  ( $Var$   $z$ ) =  $Var$  ( $swap$   $x$   $y$   $z$ )  
 $|$   $TUrename$   $x$   $y$  ( $DiffVar$   $z$ ) =  $DiffVar$  ( $swap$   $x$   $y$   $z$ )  
 $|$   $TUrename$   $x$   $y$  ( $Const$   $r$ ) = ( $Const$   $r$ )  
 $|$   $TUrename$   $x$   $y$  ( $Function$   $f$   $args$ ) =  $Function$   $f$  ( $\lambda i. TUrename$   $x$   $y$  ( $args$   $i$ ))  
 $|$   $TUrename$   $x$   $y$  ( $Plus$   $\vartheta 1$   $\vartheta 2$ ) =  $Plus$  ( $TUrename$   $x$   $y$   $\vartheta 1$ ) ( $TUrename$   $x$   $y$   $\vartheta 2$ )  
 $|$   $TUrename$   $x$   $y$  ( $Times$   $\vartheta 1$   $\vartheta 2$ ) =  $Times$  ( $TUrename$   $x$   $y$   $\vartheta 1$ ) ( $TUrename$   $x$   $y$   $\vartheta 2$ )  
 $|$   $TUrename$   $x$   $y$  ( $Differential$   $\vartheta$ ) =  $Differential$  ( $TUrename$   $x$   $y$   $\vartheta$ )

**primrec** *OUrename* ::  $'sz \Rightarrow 'sz \Rightarrow ('sf, 'sz)$  *ODE*  $\Rightarrow ('sf, 'sz)$  *ODE*

**where**

$OUrename\ x\ y\ (OVar\ c) = undefined$   
|  $OUrename\ x\ y\ (OSing\ z\ \vartheta) = OSing\ (swap\ x\ y\ z)\ (TUrename\ x\ y\ \vartheta)$   
|  $OUrename\ x\ y\ (OProd\ ODE1\ ODE2) = OProd\ (OUrename\ x\ y\ ODE1)\ (OUrename\ x\ y\ ODE2)$

**inductive**  $ORadmit :: ('sf, 'sz)\ ODE \Rightarrow bool$

**where**

$ORadmit-Sing:ORadmit\ (OSing\ x\ \vartheta)$   
|  $ORadmit-Prod:ORadmit\ ODE1 \Longrightarrow ORadmit\ ODE2 \Longrightarrow ORadmit\ (OProd\ ODE1\ ODE2)$

**primrec**  $PUrename :: 'sz \Rightarrow 'sz \Rightarrow ('sf, 'sc, 'sz)\ hp \Rightarrow ('sf, 'sc, 'sz)\ hp$

**and**  $FUrename :: 'sz \Rightarrow 'sz \Rightarrow ('sf, 'sc, 'sz)\ formula \Rightarrow ('sf, 'sc, 'sz)\ formula$

**where**

$PUrename\ x\ y\ (Pvar\ a) = undefined$   
|  $PUrename\ x\ y\ (Assign\ z\ \vartheta) = Assign\ (swap\ x\ y\ z)\ (TUrename\ x\ y\ \vartheta)$   
|  $PUrename\ x\ y\ (DiffAssign\ z\ \vartheta) = DiffAssign\ (swap\ x\ y\ z)\ (TUrename\ x\ y\ \vartheta)$   
|  $PUrename\ x\ y\ (Test\ \varphi) = Test\ (FUrename\ x\ y\ \varphi)$   
|  $PUrename\ x\ y\ (EvolveODE\ ODE\ \varphi) = EvolveODE\ (OUrename\ x\ y\ ODE)\ (FUrename\ x\ y\ \varphi)$   
|  $PUrename\ x\ y\ (Choice\ a\ b) = Choice\ (PUrename\ x\ y\ a)\ (PUrename\ x\ y\ b)$   
|  $PUrename\ x\ y\ (Sequence\ a\ b) = Sequence\ (PUrename\ x\ y\ a)\ (PUrename\ x\ y\ b)$   
|  $PUrename\ x\ y\ (Loop\ a) = Loop\ (PUrename\ x\ y\ a)$   
  
|  $FUrename\ x\ y\ (Geq\ \vartheta1\ \vartheta2) = Geq\ (TUrename\ x\ y\ \vartheta1)\ (TUrename\ x\ y\ \vartheta2)$   
|  $FUrename\ x\ y\ (Prop\ p\ args) = Prop\ p\ (\lambda i.\ TUrename\ x\ y\ (args\ i))$   
|  $FUrename\ x\ y\ (Not\ \varphi) = Not\ (FUrename\ x\ y\ \varphi)$   
|  $FUrename\ x\ y\ (And\ \varphi\ \psi) = And\ (FUrename\ x\ y\ \varphi)\ (FUrename\ x\ y\ \psi)$   
|  $FUrename\ x\ y\ (Exists\ z\ \varphi) = Exists\ (swap\ x\ y\ z)\ (FUrename\ x\ y\ \varphi)$   
|  $FUrename\ x\ y\ (Diamond\ \alpha\ \varphi) = Diamond\ (PUrename\ x\ y\ \alpha)\ (FUrename\ x\ y\ \varphi)$   
|  $FUrename\ x\ y\ (InContext\ C\ \varphi) = undefined$

## 12.2 Uniform Renaming Admissibility

**inductive**  $PRadmit :: ('sf, 'sc, 'sz)\ hp \Rightarrow bool$

**and**  $FRadmit :: ('sf, 'sc, 'sz)\ formula \Rightarrow bool$

**where**

$PRadmit-Assign:PRadmit\ (Assign\ x\ \vartheta)$   
|  $PRadmit-DiffAssign:PRadmit\ (DiffAssign\ x\ \vartheta)$   
|  $PRadmit-Test:FRadmit\ \varphi \Longrightarrow PRadmit\ (Test\ \varphi)$   
|  $PRadmit-EvolveODE:ORadmit\ ODE \Longrightarrow FRadmit\ \varphi \Longrightarrow PRadmit\ (EvolveODE\ ODE\ \varphi)$   
|  $PRadmit-Choice:PRadmit\ a \Longrightarrow PRadmit\ b \Longrightarrow PRadmit\ (Choice\ a\ b)$   
|  $PRadmit-Sequence:PRadmit\ a \Longrightarrow PRadmit\ b \Longrightarrow PRadmit\ (Sequence\ a\ b)$   
|  $PRadmit-Loop:PRadmit\ a \Longrightarrow PRadmit\ (Loop\ a)$   
  
|  $FRadmit-Geq:FRadmit\ (Geq\ \vartheta1\ \vartheta2)$   
|  $FRadmit-Prop:FRadmit\ (Prop\ p\ args)$

| *FRadmit-Not*:  $FRadmit \varphi \implies FRadmit (Not \varphi)$   
| *FRadmit-And*:  $FRadmit \varphi \implies FRadmit \psi \implies FRadmit (And \varphi \psi)$   
| *FRadmit-Exists*:  $FRadmit \varphi \implies FRadmit (Exists x \varphi)$   
| *FRadmit-Diamond*:  $PRadmit \alpha \implies FRadmit \varphi \implies FRadmit (Diamond \alpha \varphi)$

### inductive-simps

*FRadmit-box-simps*[*simp*]:  $FRadmit (Box a f)$   
**and** *PRadmit-box-simps*[*simp*]:  $PRadmit (Assign x e)$

**definition** *RSadj* ::  $'sz \Rightarrow 'sz \Rightarrow 'sz \text{ simple-state} \Rightarrow 'sz \text{ simple-state}$   
**where**  $RSadj\ x\ y\ \nu = (\chi\ z.\ \nu\ \$\ (swap\ x\ y\ z))$

**definition** *Radj* ::  $'sz \Rightarrow 'sz \Rightarrow 'sz \text{ state} \Rightarrow 'sz \text{ state}$   
**where**  $Radj\ x\ y\ \nu = (RSadj\ x\ y\ (fst\ \nu),\ RSadj\ x\ y\ (snd\ \nu))$

**lemma** *SUren*:  $stern-sem\ I\ (TUrename\ x\ y\ \vartheta)\ \nu = stern-sem\ I\ \vartheta\ (RSadj\ x\ y\ \nu)$   
⟨*proof*⟩

**lemma** *ren-preserves-dfree*:  $dfree\ \vartheta \implies dfree\ (TUrename\ x\ y\ \vartheta)$   
⟨*proof*⟩

## 12.3 Uniform Renaming Soundness Proof and Lemmas

**lemma** *TUren-frechet*:

**assumes** *good-interp:is-interp* *I*  
**shows**  $dfree\ \vartheta \implies frechet\ I\ (TUrename\ x\ y\ \vartheta)\ \nu\ \nu' = frechet\ I\ \vartheta\ (RSadj\ x\ y\ \nu)$   
 $(RSadj\ x\ y\ \nu')$   
⟨*proof*⟩

**lemma** *RSadj-fst*:  $RSadj\ x\ y\ (fst\ \nu) = fst\ (Radj\ x\ y\ \nu)$   
⟨*proof*⟩

**lemma** *RSadj-snd*:  $RSadj\ x\ y\ (snd\ \nu) = snd\ (Radj\ x\ y\ \nu)$   
⟨*proof*⟩

**lemma** *TUren*:

**assumes** *good-interp:is-interp* *I*  
**shows**  $d\text{safe}\ \vartheta \implies d\text{term-sem}\ I\ (TUrename\ x\ y\ \vartheta)\ \nu = d\text{term-sem}\ I\ \vartheta\ (Radj\ x\ y\ \nu)$   
⟨*proof*⟩

**lemma** *adj-sum*:  $RSadj\ x\ y\ (\nu1 + \nu2) = (RSadj\ x\ y\ \nu1) + (RSadj\ x\ y\ \nu2)$   
⟨*proof*⟩

**lemma** *OUren*:  $ORadmit\ ODE \implies ODE-sem\ I\ (OUrename\ x\ y\ ODE)\ \nu = RSadj\ x\ y\ (ODE-sem\ I\ ODE\ (RSadj\ x\ y\ \nu))$   
⟨*proof*⟩

**lemma** *state-eq*:



**fixes**  $\nu \nu' :: 'sz \text{ state}$   
**shows**  $(\bigwedge i. (\text{fst } \nu) \$ i = (\text{fst } \nu') \$ i) \implies (\bigwedge i. (\text{snd } \nu) \$ i = (\text{snd } \nu') \$ i) \implies \nu = \nu'$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-repv1*:  
**fixes**  $x y z :: 'sz$   
**shows**  $(\text{Radj } x y (\text{repv } \nu y r)) = \text{repv } (\text{Radj } x y \nu) x r$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-repv2*:  
**fixes**  $x y z :: 'sz$   
**shows**  $(\text{Radj } x y (\text{repv } \nu x r)) = \text{repv } (\text{Radj } x y \nu) y r$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-repv3*:  
**fixes**  $x y z :: 'sz$   
**assumes**  $zx:z \neq x$  **and**  $zy:z \neq y$   
**shows**  $(\text{Radj } x y (\text{repv } \nu z r)) = \text{repv } (\text{Radj } x y \nu) z r$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-repd1*:  
**fixes**  $x y z :: 'sz$   
**shows**  $(\text{Radj } x y (\text{repd } \nu y r)) = \text{repd } (\text{Radj } x y \nu) x r$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-repd2*:  
**fixes**  $x y z :: 'sz$   
**shows**  $(\text{Radj } x y (\text{repd } \nu x r)) = \text{repd } (\text{Radj } x y \nu) y r$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-repd3*:  
**fixes**  $x y z :: 'sz$   
**assumes**  $zx:z \neq x$  **and**  $zy:z \neq y$   
**shows**  $(\text{Radj } x y (\text{repd } \nu z r)) = \text{repd } (\text{Radj } x y \nu) z r$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-eq-iff*:  $(a = b) = ((\text{Radj } x y a) = (\text{Radj } x y b))$   
 $\langle \text{proof} \rangle$

**lemma** *RSadj-cancel*:  $\text{RSadj } x y (\text{RSadj } x y \nu) = \nu$   
 $\langle \text{proof} \rangle$

**lemma** *Radj-cancel*:  $\text{Radj } x y (\text{Radj } x y \nu) = \nu$   
 $\langle \text{proof} \rangle$

**lemma** *OUrename-preserves-ODE-vars*:  $\text{ORadmit } \text{ODE} \implies \{z. (\text{swap } x y z) \in \text{ODE-vars } I \text{ ODE}\} = \text{ODE-vars } I (\text{OUrename } x y \text{ ODE})$   
 $\langle \text{proof} \rangle$

**lemma** *ren-proj*: $(RSadj\ x\ y\ a)\ \$\ z = a\ \$\ (swap\ x\ y\ z)$   
 ⟨proof⟩

**lemma** *swap-cancel*: $swap\ x\ y\ (swap\ x\ y\ z) = z$   
 ⟨proof⟩

**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⟩

**lemma** *sol-lemma*:

**assumes** *ORA:ORadmit ODE*  
**assumes**  $t:0 \leq t$   
**assumes**  $fml:\bigwedge\nu. (\nu \in fml-sem\ I\ (FUrename\ x\ y\ \varphi)) = (Radj\ x\ y\ \nu \in fml-sem\ I\ \varphi)$   
**assumes**  $sol:(sol\ solves-ode\ (\lambda a. ODE-sem\ I\ (OUrename\ x\ y\ ODE)))\ \{0..t\}\ \{xa.\ mk-v\ I\ (OUrename\ x\ y\ ODE)\ (sol\ 0,\ b)\ xa \in fml-sem\ I\ (FUrename\ x\ y\ \varphi)\}$   
**shows**  $((\lambda t. RSadj\ x\ y\ (sol\ t))\ solves-ode\ (\lambda a. ODE-sem\ I\ ODE))\ \{0..t\}\ \{xa.\ mk-v\ I\ ODE\ (RSadj\ x\ y\ (sol\ 0),\ RSadj\ x\ y\ b)\ xa \in fml-sem\ I\ \varphi\}$   
 ⟨proof⟩

**lemma** *sol-lemma2*:

**assumes** *ORA:ORadmit ODE*  
**assumes**  $t:0 \leq t$   
**assumes**  $fml:\bigwedge\nu. (\nu \in fml-sem\ I\ (FUrename\ x\ y\ \varphi)) = (Radj\ x\ y\ \nu \in fml-sem\ I\ \varphi)$   
**assumes**  $sol:(sol\ solves-ode\ (\lambda a. ODE-sem\ I\ ODE))\ \{0..t\}\ \{x.\ mk-v\ I\ ODE\ (sol\ 0,\ b)\ x \in fml-sem\ I\ \varphi\}$   
**shows**  $((\lambda t. RSadj\ x\ y\ (sol\ t))\ solves-ode\ (\lambda a. ODE-sem\ I\ (OUrename\ x\ y\ ODE)))\ \{0..t\}\ \{xa.\ mk-v\ I\ (OUrename\ x\ y\ ODE)\ (RSadj\ x\ y\ (sol\ 0),\ RSadj\ x\ y\ b)\ xa \in fml-sem\ I\ (FUrename\ x\ y\ \varphi)\}$   
 ⟨proof⟩

**lemma** *PUren-FUren*:

**assumes** *good-interp:is-interp I*  
**shows**  
 $(PRadmit\ \alpha \longrightarrow hpsafe\ \alpha \longrightarrow (\forall\ \nu\ \omega. (\nu,\ \omega) \in prog-sem\ I\ (PUrename\ x\ y\ \alpha)) \longleftrightarrow (Radj\ x\ y\ \nu,\ Radj\ x\ y\ \omega) \in prog-sem\ I\ \alpha))$   
 $\wedge (FRadmit\ \varphi \longrightarrow fsafe\ \varphi \longrightarrow (\forall\ \nu. \nu \in fml-sem\ I\ (FUrename\ x\ y\ \varphi)) \longleftrightarrow (Radj\ x\ y\ \nu) \in fml-sem\ I\ \varphi)$   
 ⟨proof⟩

**lemma** *FUren:is-interp I*  $\implies FRadmit\ \varphi \implies fsafe\ \varphi \implies (\bigwedge\nu. (\nu \in fml-sem\ I\ (FUrename\ x\ y\ \varphi)) = (Radj\ x\ y\ \nu \in fml-sem\ I\ \varphi))$   
 ⟨proof⟩

## 12.4 Uniform Renaming Rule Soundness

**lemma** *URename-sound:FRadmit*  $\varphi \implies \text{fsafe } \varphi \implies \text{valid } \varphi \implies \text{valid } (FUrename\ x\ y\ \varphi)$   
*<proof>*

## 12.5 Bound Renaming Rule Soundness

**lemma** *BRename-sound:*  
**assumes** *FRA:FRadmit* ( $[[Assign\ x\ \vartheta]]\varphi$ )  
**assumes** *fsafe:fsafe* ( $[[Assign\ x\ \vartheta]]\varphi$ )  
**assumes** *valid:valid* ( $[[Assign\ x\ \vartheta]]\varphi$ )  
**assumes** *FVF*: $\{Inl\ y, Inr\ y, Inr\ x\} \cap FVF\ \varphi = \{\}$   
**shows** *valid* ( $[[Assign\ y\ \vartheta]]FUrename\ x\ y\ \varphi$ )  
*<proof>*

**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*  $\Rightarrow$  *char list list*  $\Rightarrow$  *char list*  
**where** *join* *S* [] = []  
  | *join* *S* [*S'*] = *S'*  
  | *join* *S* (*S' # SS*) = *S' @ S @ (join S SS)*

**fun** *vid-to-string*::*'sz*  $\Rightarrow$  *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*  $\Rightarrow$  *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:*  $\bigwedge A B C. \text{sublist } A B \implies \text{sound } (A, C) \implies \text{sound } (B, C)$   
*<proof>*

**lemma** *sound-weaken*:  $\bigwedge SG\ SGS\ C. \text{sound } (SGS, C) \implies \text{sound } (SG \# SGS, C)$   
 ⟨*proof*⟩

**lemma** *member-filter*:  $\bigwedge P. \text{List.member } (\text{filter } P\ L)\ x \implies \text{List.member } L\ x$   
 ⟨*proof*⟩

**lemma** *nth-member*:  $n < \text{List.length } L \implies \text{List.member } L\ (\text{List.nth } L\ n)$   
 ⟨*proof*⟩

**lemma** *mem-appL*:  $\text{List.member } A\ x \implies \text{List.member } (A\ @\ B)\ x$   
 ⟨*proof*⟩

**lemma** *sound-weaken-appR*:  $\bigwedge SG\ SGS\ C. \text{sound } (SG, C) \implies \text{sound } (SG\ @\ SGS, C)$   
 ⟨*proof*⟩

**fun** *start-proof*::('sf,'sc,'sz) *sequent*  $\Rightarrow$  ('sf,'sc,'sz) *rule*  
**where** *start-proof*  $S = ([S], S)$

**lemma** *start-proof-sound*:  $\text{sound } (\text{start-proof } S)$   
 ⟨*proof*⟩

## 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*  $\Rightarrow$  ('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)
  ⟨proof⟩
```

```
lemma axiom-valid:valid (get-axiom a)
  ⟨proof⟩
```

```
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 seq-to-string C
```

```
fun close :: 'a list ⇒ 'a ⇒ 'a list
where close L x = filter (λy. y ≠ x) L
```

```
fun closeI :: 'a list ⇒ nat ⇒ 'a list
where closeI L i = close L (nth L i)
```

**lemma** *close-sub:sublist* (close  $\Gamma$   $\varphi$ )  $\Gamma$   
 ⟨proof⟩

**lemma** *close-app-comm*:close (A @ B) x = close A x @ close B x  
 ⟨proof⟩

**lemma** *close-provable-sound*:sound (SG, C)  $\implies$  sound (close SG  $\varphi$ ,  $\varphi$ )  $\implies$  sound  
 (close SG  $\varphi$ , C)  
 ⟨proof⟩

**fun** *Lrule-result* :: lrule  $\Rightarrow$  nat  $\Rightarrow$  ('sf, 'sc, 'sz) sequent  $\Rightarrow$  ('sf, 'sc, 'sz) sequent  
 list

**where** *Lrule-result AndL* j (A,S) = (case (nth A j) of And p q  $\Rightarrow$  [(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)))  $\Rightarrow$   
 [(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'))))  $\Rightarrow$   
 [(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'))))  $\Rightarrow$   
 [(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  $\Rightarrow$  nat  $\Rightarrow$  ('sf, 'sc, 'sz) sequent  $\Rightarrow$  ('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)))  $\Rightarrow$  [(p # A, q # (closeI S j))] | -  $\Rightarrow$  undefined)  
 | *Rstep-And*:*Rrule-result AndR* j (A,S) = (case (nth S j) of (And p q)  $\Rightarrow$  [(A, p #  
 (closeI S j)), (A, q # (closeI S j))])  
 | *Rstep-Equiv*:*Rrule-result EquivR* j (A,S) =  
 (case (nth S j) of Not(And (Not (And p q)) (Not (And (Not p') (Not q'))))  $\Rightarrow$   
 (if (p = p'  $\wedge$  q = q') then [(p # A, q # (closeI S j)), (q # A, p #  
 (closeI S j))] else undefined))  
 | *Rstep-Cohide*:*Rrule-result CohideR* j (A,S) = [(A, [nth S j])]  
 | *Rstep-CohideRR*:*Rrule-result CohideRR* j (A,S) = [([], [nth S j])]  
 | *Rstep-True*:*Rrule-result TrueR* j (A,S) = []

**fun** *step-result* :: ('sf, 'sc, 'sz) rule  $\Rightarrow$  (nat \* ('sf, 'sc, 'sz) step)  $\Rightarrow$  ('sf, 'sc, 'sz)  
 rule

**where**



$Step\text{-}axiom\text{:}step\text{-}result (SG,C) (i,Axiom a) = (closeI SG i, C)$   
 $| Step\text{-}AxSubst\text{:}step\text{-}result (SG,C) (i,AxSubst a \sigma) = (closeI SG i, C)$   
 $| Step\text{-}Lrule\text{:}step\text{-}result (SG,C) (i,Lrule L j) = (close (append SG (Lrule\text{-}result L j (nth SG i))) (nth SG i), C)$   
 $| Step\text{-}Rrule\text{:}step\text{-}result (SG,C) (i,Rrule L j) = (close (append SG (Rrule\text{-}result L j (nth SG i))) (nth SG i), C)$   
 $| Step\text{-}Cut\text{:}step\text{-}result (SG,C) (i,Cut \varphi) = (let (A,S) = nth SG i in ((\varphi \# A, S) \# ((A, \varphi \# S) \# (closeI SG i), C)))$   
 $| Step\text{-}Vsubst\text{:}step\text{-}result (SG,C) (i,VSubst \varphi \sigma) = (closeI SG i, C)$   
 $| Step\text{-}CloseId\text{:}step\text{-}result (SG,C) (i,CloseId j k) = (closeI SG i, C)$   
 $| Step\text{-}G\text{:}step\text{-}result (SG,C) (i,G) = (case nth SG i of (-, (Not (Diamond q (Not p)))) \# Nil \Rightarrow (([], [p]) \# closeI SG i, C))$   
 $| Step\text{-}DEAxiomSchema\text{:}step\text{-}result (SG,C) (i,DEAxiomSchema ODE \sigma) = (closeI SG i, C)$   
 $| Step\text{-}CE\text{:}step\text{-}result (SG,C) (i, CE \varphi \psi \sigma) = (closeI SG i, C)$   
 $| Step\text{-}CQ\text{:}step\text{-}result (SG,C) (i, CQ \vartheta_1 \vartheta_2 \sigma) = (closeI SG i, C)$   
 $| Step\text{-}default\text{:}step\text{-}result R (i,S) = R$

**fun** *deriv-result* :: ('sf, 'sc, 'sz) rule  $\Rightarrow$  ('sf, 'sc, 'sz) derivation  $\Rightarrow$  ('sf, 'sc, 'sz) rule

**where**

$deriv\text{-}result R [] = R$   
 $| deriv\text{-}result R (s \# ss) = deriv\text{-}result (step\text{-}result R s) (ss)$

**fun** *proof-result* :: ('sf, 'sc, 'sz) pf  $\Rightarrow$  ('sf, 'sc, 'sz) rule

**where** *proof-result* (D,S) = *deriv-result* (start-proof D) S

**inductive** *lrule-ok* :: ('sf, 'sc, 'sz) sequent list  $\Rightarrow$  ('sf, 'sc, 'sz) sequent  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  lrule  $\Rightarrow$  bool

**where**

$Lrule\text{-}And:\bigwedge p q. nth (fst (nth SG i)) j = (p \&\& q) \implies lrule\text{-}ok SG C i j AndL$   
 $| Lrule\text{-}ImPLY:\bigwedge p q. nth (fst (nth SG i)) j = (p \rightarrow q) \implies lrule\text{-}ok SG C i j ImPLYL$   
 $| Lrule\text{-}EquivForward:\bigwedge p q. nth (fst (nth SG i)) j = (p \leftrightarrow q) \implies lrule\text{-}ok SG C i j EquivForwardL$   
 $| Lrule\text{-}EquivBackward:\bigwedge p q. nth (fst (nth SG i)) j = (p \leftrightarrow q) \implies lrule\text{-}ok SG C i j EquivBackwardL$

**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**

$Lrule\text{-}And[prover]: lrule\text{-}ok SG C i j AndL$   
**and**  $Lrule\text{-}ImPLY[prover]: lrule\text{-}ok SG C i j ImPLYL$   
**and**  $Lrule\text{-}Forward[prover]: lrule\text{-}ok SG C i j EquivForwardL$   
**and**  $Lrule\text{-}EquivBackward[prover]: lrule\text{-}ok SG C i j EquivBackwardL$

**inductive** *rrule-ok* :: ('sf, 'sc, 'sz) sequent list  $\Rightarrow$  ('sf, 'sc, 'sz) sequent  $\Rightarrow$  nat  $\Rightarrow$  nat

$\Rightarrow rrule \Rightarrow bool$

**where**

$Rrule\text{-}And:\bigwedge p q. nth (snd (nth SG i)) j = (p \&\& q) \Rightarrow rrule\text{-}ok SG C i j AndR$   
 $| Rrule\text{-}Imply:\bigwedge p q. nth (snd (nth SG i)) j = (p \rightarrow q) \Rightarrow rrule\text{-}ok SG C i j ImplyR$   
 $| Rrule\text{-}Equiv:\bigwedge p q. nth (snd (nth SG i)) j = (p \leftrightarrow q) \Rightarrow rrule\text{-}ok SG C i j EquivR$   
 $| Rrule\text{-}Cohide:\text{length} (snd (nth SG i)) > j \Rightarrow (\bigwedge \Gamma q. (nth SG i) \neq (\Gamma, [q])) \Rightarrow rrule\text{-}ok SG C i j CohideR$   
 $| Rrule\text{-}CohideRR:\text{length} (snd (nth SG i)) > j \Rightarrow (\bigwedge q. (nth SG i) \neq ([], [q])) \Rightarrow rrule\text{-}ok SG C i j CohideRR$   
 $| Rrule\text{-}True:nth (snd (nth SG i)) j = TT \Rightarrow rrule\text{-}ok SG C i j TrueR$

**inductive-simps**

$Rrule\text{-}And\text{-}simps[\text{prover}]: rrule\text{-}ok SG C i j AndR$   
**and**  $Rrule\text{-}Imply\text{-}simps[\text{prover}]: rrule\text{-}ok SG C i j ImplyR$   
**and**  $Rrule\text{-}Equiv\text{-}simps[\text{prover}]: rrule\text{-}ok SG C i j EquivR$   
**and**  $Rrule\text{-}CohideR\text{-}simps[\text{prover}]: rrule\text{-}ok SG C i j CohideR$   
**and**  $Rrule\text{-}CohideRR\text{-}simps[\text{prover}]: rrule\text{-}ok SG C i j CohideRR$   
**and**  $Rrule\text{-}TrueR\text{-}simps[\text{prover}]: rrule\text{-}ok SG C i j TrueR$

**inductive step-ok** :: ('sf, 'sc, 'sz) rule  $\Rightarrow nat \Rightarrow ('sf, 'sc, 'sz) step \Rightarrow bool$

**where**

$Step\text{-}Axiom:(nth SG i) = ([], [get\text{-}axiom a]) \Rightarrow step\text{-}ok (SG, C) i (Axiom a)$   
 $| Step\text{-}AxSubst:(nth SG i) = ([], [Fsubst (get\text{-}axiom a) \sigma]) \Rightarrow F\text{admit} \sigma (get\text{-}axiom a) \Rightarrow \text{ssafe} \sigma \Rightarrow step\text{-}ok (SG, C) i (AxSubst a \sigma)$   
 $| Step\text{-}Lrule:rrule\text{-}ok SG C i j L \Rightarrow j < \text{length} (fst (nth SG i)) \Rightarrow step\text{-}ok (SG, C) i (Lrule L j)$   
 $| Step\text{-}Rrule:rrule\text{-}ok SG C i j L \Rightarrow j < \text{length} (snd (nth SG i)) \Rightarrow step\text{-}ok (SG, C) i (Rrule L j)$   
 $| Step\text{-}Cut:\text{fsafe} \varphi \Rightarrow i < \text{length} SG \Rightarrow step\text{-}ok (SG, C) i (Cut \varphi)$   
 $| Step\text{-}CloseId:nth (fst (nth SG i)) j = nth (snd (nth SG i)) k \Rightarrow j < \text{length} (fst (nth SG i)) \Rightarrow k < \text{length} (snd (nth SG i)) \Rightarrow step\text{-}ok (SG, C) i (CloseId j k)$   
 $| Step\text{-}G:\bigwedge a p. nth SG i = ([], [[[[a]]p]]) \Rightarrow step\text{-}ok (SG, C) i G$   
 $| Step\text{-}DEAxiom\text{-}schema:$   
 $nth SG i =$   
 $([], [Fsubst ((([[[EvolveODE (OProd (OSing vid1 (f1 fid1 vid1)) ODE) (p1 vid2 vid1))]] (P pid1)) \leftrightarrow$   
 $([[[EvolveODE ((OProd (OSing vid1 (f1 fid1 vid1))) ODE) (p1 vid2 vid1))]]$   
 $[[DiffAssign vid1 (f1 fid1 vid1)]] P pid1)))] \sigma)]$   
 $\Rightarrow \text{ssafe} \sigma$   
 $\Rightarrow \text{osafe} ODE$   
 $\Rightarrow \{Inl vid1, Inr vid1\} \cap BVO ODE = \{\}$   
 $\Rightarrow F\text{admit} \sigma ((([[[EvolveODE (OProd (OSing vid1 (f1 fid1 vid1)) ODE) (p1 vid2 vid1))]] (P pid1)) \leftrightarrow$   
 $([[[EvolveODE ((OProd (OSing vid1 (f1 fid1 vid1)) ODE)) (p1 vid2 vid1))]]$   
 $[[DiffAssign vid1 (f1 fid1 vid1)]] P pid1)))]$   
 $\Rightarrow step\text{-}ok (SG, C) i (DEAxiomSchema ODE \sigma)$   
 $| Step\text{-}CE:nth SG i = ([], [Fsubst (Equiv (InContext pid1 \varphi) (InContext pid1 \psi) \sigma)])$   
 $\Rightarrow \text{valid} (Equiv \varphi \psi)$

$\Rightarrow$  *fsafe*  $\varphi$   
 $\Rightarrow$  *fsafe*  $\psi$   
 $\Rightarrow$  *ssafe*  $\sigma$   
 $\Rightarrow$  *Fadmit*  $\sigma$  (*Equiv* (*InContext* *pid1*  $\varphi$ ) (*InContext* *pid1*  $\psi$ ))  
 $\Rightarrow$  *step-ok* (*SG,C*) *i* (*CE*  $\varphi$   $\psi$   $\sigma$ )  
| *Step-CQ:nth* *SG* *i* = ( $\square$ , [*Fsubst* (*Equiv* (*Prop* *p* (*singleton*  $\vartheta$ )) (*Prop* *p* (*singleton*  $\vartheta'$ )))  $\sigma$ ])  
 $\Rightarrow$  *valid* (*Equals*  $\vartheta$   $\vartheta'$ )  
 $\Rightarrow$  *dsafe*  $\vartheta$   
 $\Rightarrow$  *dsafe*  $\vartheta'$   
 $\Rightarrow$  *ssafe*  $\sigma$   
 $\Rightarrow$  *Fadmit*  $\sigma$  (*Equiv* (*Prop* *p* (*singleton*  $\vartheta$ )) (*Prop* *p* (*singleton*  $\vartheta'$ )))  
 $\Rightarrow$  *step-ok* (*SG,C*) *i* (*CQ*  $\vartheta$   $\vartheta'$   $\sigma$ )

### 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*  $\varphi$ )  
**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*  $\sigma$ )  
**and** *Step-DEAxiom-schema-simps*[*prover*]: *step-ok* (*SG,C*) *i* (*DEAxiomSchema* *ODE*  $\sigma$ )  
**and** *Step-CE-simps*[*prover*]: *step-ok* (*SG,C*) *i* (*CE*  $\varphi$   $\psi$   $\sigma$ )  
**and** *Step-CQ-simps*[*prover*]: *step-ok* (*SG,C*) *i* (*CQ*  $\vartheta$   $\vartheta'$   $\sigma$ )

**inductive** *deriv-ok* :: (*'sf*, *'sc*, *'sz*) *rule*  $\Rightarrow$  (*'sf*, *'sc*, *'sz*) *derivation*  $\Rightarrow$  *bool*

**where**

*Deriv-Nil:deriv-ok* *R* *Nil*  
| *Deriv-Cons:step-ok* *R* *i* *S*  $\Rightarrow$   $i \geq 0 \Rightarrow i < \text{length} (\text{fst } R) \Rightarrow \text{deriv-ok} (\text{step-result } R (i,S)) \text{ SS} \Rightarrow \text{deriv-ok } R ((i,S) \# \text{SS})$

### 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*  $\Rightarrow$  *bool*

**where**

*Proof-ok:deriv-ok* (*start-proof* *D*) *S*  $\Rightarrow$  *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*]:  $\bigwedge x. \text{List.member } [x] x$   
 $\langle \text{proof} \rangle$

**lemma** *mem-appR*[*member-intros*]:  $\bigwedge A B x. \text{List.member } B x \implies \text{List.member } (A @ B) x$   
 ⟨*proof*⟩

**lemma** *mem-filter*[*member-intros*]:  $\bigwedge A P x. P x \implies \text{List.member } A x \implies \text{List.member } (\text{filter } P A) x$   
 ⟨*proof*⟩

**lemma** *sound-weaken-appL*:  $\bigwedge SG SGS C. \text{sound } (SGS, C) \implies \text{sound } (SG @ SGS, C)$   
 ⟨*proof*⟩

**lemma** *fml-seq-valid*:  $\text{valid } \varphi \implies \text{seq-valid } ([], [\varphi])$   
 ⟨*proof*⟩

**lemma** *closeI-provable-sound*:  $\bigwedge i. \text{sound } (SG, C) \implies \text{sound } (\text{closeI } SG i, (\text{nth } SG i)) \implies \text{sound } (\text{closeI } SG i, C)$   
 ⟨*proof*⟩

**lemma** *valid-to-sound*:  $\text{seq-valid } A \implies \text{sound } (B, A)$   
 ⟨*proof*⟩

**lemma** *closeI-valid-sound*:  $\bigwedge i. \text{sound } (SG, C) \implies \text{seq-valid } (\text{nth } SG i) \implies \text{sound } (\text{closeI } SG i, C)$   
 ⟨*proof*⟩

**lemma** *close-nonmember-eq*:  $\neg(\text{List.member } A a) \implies \text{close } A a = A$   
 ⟨*proof*⟩

**lemma** *close-noneq-nonempty*:  $\text{List.member } A x \implies x \neq a \implies \text{close } A a \neq []$   
 ⟨*proof*⟩

**lemma** *close-app-neq*:  $\text{List.member } A x \implies x \neq a \implies \text{close } (A @ B) a \neq B$   
 ⟨*proof*⟩

**lemma** *member-singD*:  $\bigwedge x P. P x \implies (\bigwedge y. \text{List.member } [x] y \implies P y)$   
 ⟨*proof*⟩

**lemma** *fst-neq*:  $A \neq B \implies (A, C) \neq (B, D)$   
 ⟨*proof*⟩

**lemma** *lrule-sound*:  $\text{lrule-ok } SG C i j L \implies i < \text{length } SG \implies j < \text{length } (\text{fst } (SG ! i)) \implies \text{sound } (SG, C) \implies \text{sound } (\text{close } (\text{append } SG (\text{Lrule-result } L j (\text{nth } SG i))) (\text{nth } SG i), C)$   
 ⟨*proof*⟩

**lemma** *rrule-sound*:  $\text{rrule-ok } SG C i j L \implies i < \text{length } SG \implies j < \text{length } (\text{snd } (SG ! i)) \implies \text{sound } (SG, C) \implies \text{sound } (\text{close } (\text{append } SG (\text{Rrule-result } L j (\text{nth } SG i))) (\text{nth } SG i), C)$

$SG\ i))$  ( $nth\ SG\ i$ ,  $C$ )  
 $\langle proof \rangle$

**lemma**  $step\ sound: step\ ok\ R\ i\ S \implies i \geq 0 \implies i < length\ (fst\ R) \implies sound\ R$   
 $\implies sound\ (step\ result\ R\ (i,S))$   
 $\langle proof \rangle$

**lemma**  $deriv\ sound: deriv\ ok\ R\ D \implies sound\ R \implies sound\ (deriv\ result\ R\ D)$   
 $\langle proof \rangle$

**lemma**  $proof\ sound: proof\ ok\ Pf \implies sound\ (proof\ result\ Pf)$   
 $\langle proof \rangle$

## 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**  $DIAndCut =$   
 $( ([[ [\alpha\ vid1]]( (And\ (Predicational\ ( pid3))\ (Predicational\ ( pid4)))) \rightarrow (And\ (Predicational\ ( pid1))\ (Predicational\ ( pid2))))$   
 $\rightarrow ([[ [\alpha\ vid1]](And\ (Predicational\ ( pid3))\ (Predicational\ ( pid4)))) \rightarrow ([[ [\alpha\ vid1]](And\ (Predicational\ ( pid1))\ (Predicational\ ( pid2))))$

**definition**  $DIAndSubst::('sf, 'sc, 'sz)\ subst$   
**where**  $DIAndSubst =$   
 $\langle$   $SFunctions = (\lambda-. None),$   
 $SPredicates = (\lambda-. None),$   
 $SContexts = (\lambda C. (if\ C = pid1\ then\ Some(And\ (Predicational\ (Inl\ pid3))\ (Predicational\ (Inl\ pid4)))$   
 $else\ (if\ C = pid2\ then\ Some(And\ (Predicational\ (Inl\ pid1))\ (Predicational\ (Inl\ pid2)))\ else\ None)),$   
 $SPrograms = (\lambda-. None),$   
 $SODEs = (\lambda-. None)$   
 $\rangle$

—  $[a]R \& H \rightarrow R \rightarrow [a]R \& H \rightarrow [a]R$   $DIAndSubst34$

**definition**  $DIAndSubst341::('sf,'sc,'sz)$  subst

**where**  $DIAndSubst341 =$

( $\lfloor$   $SFunctions = (\lambda-. None)$ ,  
 $SPredicates = (\lambda-. None)$ ,  
 $SContexts = (\lambda C. (if C = pid1 then Some(And (Predicational (Inl pid3))$   
 $(Predicational (Inl pid4)))$   
 $else (if C = pid2 then Some(Predicational (Inl pid3)) else None)))$ ,  
 $SPrograms = (\lambda-. None)$ ,  
 $SODEs = (\lambda-. None)$

$\rfloor$

**definition**  $DIAndSubst342::('sf,'sc,'sz)$  subst

**where**  $DIAndSubst342 =$

( $\lfloor$   $SFunctions = (\lambda-. None)$ ,  
 $SPredicates = (\lambda-. None)$ ,  
 $SContexts = (\lambda C. (if C = pid1 then Some(And (Predicational (Inl pid3))$   
 $(Predicational (Inl pid4)))$   
 $else (if C = pid2 then Some(Predicational (Inl pid4)) else None)))$ ,  
 $SPrograms = (\lambda-. None)$ ,  
 $SODEs = (\lambda-. None)$

$\rfloor$

—  $[a]P, [a]R\&H, P, Q \mid - [a]Q \rightarrow P\&Q \rightarrow [a]Q \rightarrow [a]P\&Q, [a]P\&Q;$

**definition**  $DIAndSubst12::('sf,'sc,'sz)$  subst

**where**  $DIAndSubst12 =$

( $\lfloor$   $SFunctions = (\lambda-. None)$ ,  
 $SPredicates = (\lambda-. None)$ ,  
 $SContexts = (\lambda C. (if C = pid1 then Some(Predicational (Inl pid2))$   
 $else (if C = pid2 then Some(Predicational (Inl pid1) \&\& Predicational$   
 $(Inl pid2)) else None)))$ ,  
 $SPrograms = (\lambda-. None)$ ,  
 $SODEs = (\lambda-. None)$

$\rfloor$

—  $P \rightarrow Q \rightarrow P\&Q$

**definition**  $DIAndCurry12::('sf,'sc,'sz)$  subst

**where**  $DIAndCurry12 =$

( $\lfloor$   $SFunctions = (\lambda-. None)$ ,  
 $SPredicates = (\lambda-. None)$ ,  
 $SContexts = (\lambda C. (if C = pid1 then Some(Predicational (Inl pid1))$   
 $else (if C = pid2 then Some(Predicational (Inl pid2) \rightarrow (Predicational$   
 $(Inl pid1) \&\& Predicational (Inl pid2))) else None)))$ ,  
 $SPrograms = (\lambda-. None)$ ,  
 $SODEs = (\lambda-. None)$

$\rfloor$

**definition**  $DIAnd :: ('sf,'sc,'sz)$  rule

**where**  $DIAnd =$

( $(\lfloor (\lfloor, [DIAndSG1]), (\lfloor, [DIAndSG2])$ ,  
 $DIAndConcl$ )

**definition**  $DIAndCutP1 :: ('sf, 'sc, 'sz) \text{ formula}$   
**where**  $DIAndCutP1 = ([[Pvar \text{ vid1}]](Predicational \text{ pid1}))$

**definition**  $DIAndCutP2 :: ('sf, 'sc, 'sz) \text{ formula}$   
**where**  $DIAndCutP2 = ([[Pvar \text{ vid1}]](Predicational \text{ pid2}))$

**definition**  $DIAndCutP12 :: ('sf, 'sc, 'sz) \text{ formula}$   
**where**  $DIAndCutP12 = ((([[Pvar \text{ vid1}]](Pc \text{ pid1}) \rightarrow (Pc \text{ pid2} \rightarrow (And (Pc \text{ pid1}) (Pc \text{ pid2}))))))$   
 $\rightarrow ((([[Pvar \text{ vid1}]](Pc \text{ pid1}) \rightarrow ([[Pvar \text{ vid1}]](Pc \text{ pid2} \rightarrow (And (Pc \text{ pid1}) (Pc \text{ pid2}))))))$

**definition**  $DIAndCut34Elim1 :: ('sf, 'sc, 'sz) \text{ formula}$   
**where**  $DIAndCut34Elim1 = ((([[Pvar \text{ vid1}]](Pc \text{ pid3} \&\& Pc \text{ pid4}) \rightarrow (Pc \text{ pid3})))$   
 $\rightarrow ((([[Pvar \text{ vid1}]](Pc \text{ pid3} \&\& Pc \text{ pid4}) \rightarrow ([[Pvar \text{ vid1}]](Pc \text{ pid3}))))$

**definition**  $DIAndCut34Elim2 :: ('sf, 'sc, 'sz) \text{ formula}$   
**where**  $DIAndCut34Elim2 = ((([[Pvar \text{ vid1}]](Pc \text{ pid3} \&\& Pc \text{ pid4}) \rightarrow (Pc \text{ pid4})))$   
 $\rightarrow ((([[Pvar \text{ vid1}]](Pc \text{ pid3} \&\& Pc \text{ pid4}) \rightarrow ([[Pvar \text{ vid1}]](Pc \text{ pid4}))))$

**definition**  $DIAndCut12Intro :: ('sf, 'sc, 'sz) \text{ formula}$   
**where**  $DIAndCut12Intro = ((([[Pvar \text{ vid1}]](Pc \text{ pid2} \rightarrow (Pc \text{ pid1} \&\& Pc \text{ pid2})))$   
 $\rightarrow ((([[Pvar \text{ vid1}]](Pc \text{ pid2}) \rightarrow ([[Pvar \text{ vid1}]](Pc \text{ pid1} \&\& Pc \text{ pid2}))))$

**definition**  $DIAndProof :: ('sf, 'sc, 'sz) \text{ pf}$   
**where**  $DIAndProof =$   
 $(DIAndConcl, [$   
 $(0, Rrule \text{ ImplyR } 0) \text{ — } 1$   
 $, (0, Lrule \text{ AndL } 0)$   
 $, (0, Rrule \text{ ImplyR } 0)$   
 $, (0, \text{Cut } DIAndCutP1)$   
 $, (1, \text{Cut } DIAndSG1)$   
 $, (0, Rrule \text{ CohideR } 0)$   
 $, (Suc (Suc 0), Lrule \text{ ImplyL } 0)$   
 $, (Suc (Suc (Suc 0)), \text{CloseId } 1 0)$   
 $, (Suc (Suc 0), Lrule \text{ ImplyL } 0)$   
 $, (Suc (Suc 0), \text{CloseId } 0 0)$   
 $, (Suc (Suc 0), \text{Cut } DIAndCut34Elim1) \text{ — } 11$   
 $, (0, Lrule \text{ ImplyL } 0)$   
 $, (Suc (Suc (Suc 0)), Lrule \text{ ImplyL } 0)$   
 $, (0, Rrule \text{ CohideRR } 0)$   
 $, (0, Rrule \text{ CohideRR } 0)$   
 $, (Suc 0, Rrule \text{ CohideRR } 0)$   
 $, (Suc (Suc (Suc (Suc (Suc 0))))), G)$   
 $, (0, Rrule \text{ ImplyR } 0)$   
 $, (Suc (Suc (Suc (Suc (Suc 0))))), Lrule \text{ AndL } 0)$   
 $, (Suc (Suc (Suc (Suc (Suc 0))))), \text{CloseId } 0 0)$   
 $, (Suc (Suc (Suc 0)), \text{AxSubst } AK \text{ DIAndSubst341}) \text{ — } 21$

```

.(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*))  
 ⟨proof⟩

## 17 Example 2: Concrete Hybrid System

—  $v \geq 0 \wedge A() \geq 0 \longrightarrow [v' = A, x' = v]v' \geq 0$

**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

$(\text{Var } vid1)) (\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))]] ((\text{Geq } (\text{Differential } (\text{Var } vid1)) (\text{Differential } (\text{Const } 0)))) \leftrightarrow$   
 $([[\text{EvolveODE } (\text{OProd } (\text{OSing } vid1 (f0 \text{ fid1})) (\text{OSing } vid2 (\text{Var } vid1))) (\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))]]$   
 $[[\text{DiffAssign } vid1 (f0 \text{ fid1})]](\text{Geq } (\text{Differential } (\text{Var } vid1)) (\text{Differential } (\text{Const } 0))))))$

**definition** *SystemKCut::('sf,'sc,'sz) formula*

**where** *SystemKCut =*

$(\text{Implies } ([[ \text{EvolveODE } (\text{OProd } (\text{OSing } vid1 (f0 \text{ fid1})) (\text{OSing } vid2 (\text{Var } vid1))) (\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))]]$   
 $(\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))]$   
 $(\text{Implies } ((\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0))) ([[ \text{DiffAssign } vid1 (f0 \text{ fid1})]](\text{Geq } (\text{Differential } (\text{Var } vid1)) (\text{Differential } (\text{Const } 0))))))$   
 $(\text{Implies } ([[ \text{EvolveODE } (\text{OProd } (\text{OSing } vid1 (f0 \text{ fid1})) (\text{OSing } vid2 (\text{Var } vid1))) (\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))]]$   
 $(\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))] ((\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0))))$   
 $([[ \text{EvolveODE } (\text{OProd } (\text{OSing } vid1 (f0 \text{ fid1})) (\text{OSing } vid2 (\text{Var } vid1))) (\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))]]$   
 $(\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0)))] ([[ \text{DiffAssign } vid1 (f0 \text{ fid1})]](\text{Geq } (\text{Differential } (\text{Var } vid1)) (\text{Differential } (\text{Const } 0))))))$

**definition** *SystemEquivCut::('sf,'sc,'sz) formula*

**where** *SystemEquivCut =*

$(\text{Equiv } (\text{Implies } ((\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0))) ([[ \text{DiffAssign } vid1 (f0 \text{ fid1})]](\text{Geq } (\text{Differential } (\text{Var } vid1)) (\text{Differential } (\text{Const } 0))))))$   
 $(\text{Implies } ((\text{And } TT (\text{Geq } (f0 \text{ fid1}) (\text{Const } 0))) ([[ \text{DiffAssign } vid1 (f0 \text{ fid1})]](\text{Geq } (\text{DiffVar } vid1) (\text{Const } 0))))))$

**definition** *SystemDiffAssignCut::('sf,'sc,'sz) formula*

**where** *SystemDiffAssignCut =*

$([[ \text{DiffAssign } vid1 (\$f \text{ fid1 empty})]] (\text{Geq } (\text{DiffVar } vid1) (\text{Const } 0)))$   
 $\leftrightarrow (\text{Geq } (\$f \text{ fid1 empty}) (\text{Const } 0)))$

**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)))  $\leftrightarrow$  Geq (DiffVar vid1) (Differential (Const 0))*

**definition** *CQ2Concl::('sf,'sc,'sz) formula*

**where** *CQ2Concl = (Geq (DiffVar vid1) (Differential (Const 0)))  $\leftrightarrow$  Geq (\$' vid1) (Const 0)*

**definition**  $CEReq::('sf, 'sc, 'sz)$  formula  
**where**  $CEReq = (Geq (Differential (trm.Var vid1)) (Differential (Const 0)) \leftrightarrow Geq (' vid1) (Const 0))$

**definition**  $CQRightSubst::('sf, 'sc, 'sz)$  subst  
**where**  $CQRightSubst =$   
 (|  $SFunctions = (\lambda-. None)$ ,  
 $SPredicates = (\lambda p. (if p = vid1 then (Some (Geq (DiffVar vid1) (Function (Inr vid1) empty))) else None))$ ,  
 $SContexts = (\lambda-. None)$ ,  
 $SPrograms = (\lambda-. None)$ ,  
 $SODEs = (\lambda-. None)$   
 |)

**definition**  $CQLeftSubst::('sf, 'sc, 'sz)$  subst  
**where**  $CQLeftSubst =$   
 (|  $SFunctions = (\lambda-. None)$ ,  
 $SPredicates = (\lambda p. (if p = vid1 then (Some (Geq (Function (Inr vid1) empty) (Differential (Const 0)))) else None))$ ,  
 $SContexts = (\lambda-. None)$ ,  
 $SPrograms = (\lambda-. None)$ ,  
 $SODEs = (\lambda-. None)$   
 |)

**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**  $CE\text{-result-correct:proof-result}$   $CEProof = ([], ([], [CEReq]))$   
 (proof)

**definition** *DiffConstSubst*::('sf,'sc,'sz) subst

**where** *DiffConstSubst* =  $\langle$   
  *SFunctions* = ( $\lambda f$ . (if  $f = \text{fid1}$  then (Some (Const 0)) else None)),  
  *SPredicates* = ( $\lambda$ -. None),  
  *SContexts* = ( $\lambda$ -. None),  
  *SPrograms* = ( $\lambda$ -. None),  
  *SODEs* = ( $\lambda$ -. None)  
 $\rangle$

**definition** *DiffConstProof*::('sf,'sc,'sz) pf

**where** *DiffConstProof* = ( $\langle$ [], [Equals (Differential (Const 0)) (Const 0))], [  
  0, AxSubst AdConst DiffConstSubst])

**lemma** *diffconst-result-correct:proof-result DiffConstProof* = ( $\langle$ [], ( $\langle$ [], [Equals (Differential (Const 0)) (Const 0))])  
 $\langle$ proof

**lemma** *diffconst-sound-lemma:sound (proof-result DiffConstProof)*  
 $\langle$ proof

**lemma** *valid-of-sound:sound* ( $\langle$ [], ( $\langle$ [], [ $\varphi$ ])  $\implies$  valid  $\varphi$   
 $\langle$ proof

**lemma** *almost-diff-const-sound:sound* ( $\langle$ [], ( $\langle$ [], [Equals (Differential (Const 0)) (Const 0)])])  
 $\langle$ proof

**lemma** *almost-diff-const:valid* (Equals (Differential (Const 0)) (Const 0))  
 $\langle$ proof

**lemma** *almost-diff-var:valid* (Equals (Differential (trm.Var vid1)) ( $\$'$  vid1))  
 $\langle$ proof

**lemma** *CESound-lemma:sound (proof-result CEProof)*  
 $\langle$ proof

**lemma** *sound-to-valid:sound* ( $\langle$ [], ( $\langle$ [], [ $\varphi$ ])  $\implies$  valid  $\varphi$   
 $\langle$ proof

**lemma** *CE1pre:sound* ( $\langle$ [], ( $\langle$ [], [CEReq])])  
 $\langle$ proof

**lemma** *CE1pre-valid:valid CEReq*  
 $\langle$ proof

**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))))))  
 $\langle$ proof

**definition** *SystemDISubst::('sf,'sc,'sz) subst*  
**where** *SystemDISubst =*  
 (| *SFunctions =* ( $\lambda f.$   
 ( if  $f = fid1$  then *Some*(*Function* (*Inr* *vid1*) *empty*)  
 else if  $f = fid2$  then *Some*(*Const* 0)  
 else *None*)),  
*SPredicates =* ( $\lambda p.$  if  $p = vid1$  then *Some* *TT* else *None*),  
*SContexts =* ( $\lambda.$  *None*),  
*SPrograms =* ( $\lambda.$  *None*),  
*SODEs =* ( $\lambda c.$  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 =* ( $\lambda$   
*f.* *None*),  
*SPredicates =* ( $\lambda p.$  *None*),  
*SContexts =* ( $\lambda C.$   
 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 =* ( $\lambda.$  *None*),  
*SODEs =* ( $\lambda c.$  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 =* ( $\lambda f.$  *None*),  
*SPredicates =* ( $\lambda p.$  if  $p = vid1$  then *Some* (*Geq* (*Function* (*Inl* *fid1*) *empty*)  
 (*Const* 0)) else *None*),  
*SContexts =* ( $\lambda.$  *None*),  
*SPrograms =* ( $\lambda a.$  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 =* ( $\lambda.$  *None*)  
 |)

)

**definition** *SystemVSubst2::('sf,'sc,'sz) subst*

**where** *SystemVSubst2 =*

( $\lambda f$ . *SFunctions* = ( $\lambda p$ . if  $p = vid1$  then *Some* (*Geq* (*Function* (*Inl fid1*) *empty*)  
*(Const 0)*) else *None*),  
*SPredicates* = ( $\lambda p$ . if  $p = vid1$  then *Some* (*Geq* (*Function* (*Inl fid1*) *empty*)  
*(Const 0)*) else *None*),  
*SContexts* = ( $\lambda$ . *None*),  
*SPrograms* = ( $\lambda a$ . if  $a = vid1$  then  
*Some* (*EvolveODE* (*OProd*  
(*OSing vid1* (*Function fid1 empty*))  
(*OSing vid2* (*Var vid1*)))  
*TT*)  
else *None*),  
*SODEs* = ( $\lambda$ . *None*)

)

**definition** *SystemDESubst::('sf,'sc,'sz) subst*

**where** *SystemDESubst =*

( $\lambda f$ . if  $f = fid1$  then *Some*(*Function* (*Inl fid1*) *empty*) else *None*),  
*SPredicates* = ( $\lambda p$ . if  $p = vid2$  then *Some*(*And TT* (*Geq* (*Function* (*Inl fid1*)  
*empty*) (*Const 0*))) else *None*),  
*SContexts* = ( $\lambda C$ . if  $C = pid1$  then *Some*(*Geq* (*Differential* (*Var vid1*))  
(*Differential* (*Const 0*))) else *None*),  
*SPrograms* = ( $\lambda$ . *None*),  
*SODEs* = ( $\lambda$ . *None*)

)

**lemma** *systemdesubst-correct*: $\exists$  *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*))))))  $\leftrightarrow$   
(((*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*))  $\leftrightarrow$   
([[*EvolveODE* ((*OProd* (*OSing vid1* (*f1 fid1 vid1*)) *ODE*) (*p1 vid2 vid1*)]  
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*)) *SystemDESubst*

(*proof*)

**definition** *SystemKSubst::('sf,'sc,'sz) subst*

**where** *SystemKSubst =* ( $\lambda f$ . *SFunctions* = ( $\lambda$ . *None*),

*SPredicates* = ( $\lambda$ . *None*),

*SContexts* = ( $\lambda C$ . 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*),

$S\text{Programs} = (\lambda c. \text{if } c = \text{vid1} \text{ then } \text{Some } (\text{EvolveODE } (\text{OProd } (\text{OSing } \text{vid1} \\ (\text{Function } \text{fid1 } \text{empty})) (\text{OSing } \text{vid2 } (\text{Var } \text{vid1})))) (\text{And } (\text{Geq } (\text{Const } 0) (\text{Const } 0)) \\ (\text{Geq } (\text{Function } \text{fid1 } \text{empty}) (\text{Const } 0)))) \text{ else } \text{None}),$   
 $S\text{ODEs} = (\lambda-. \text{None})$

⌋

**lemma**  $\text{subst-imp-simp:Fsubst } (\text{Implies } p \ q) \ \sigma = (\text{Implies } (\text{Fsubst } p \ \sigma) (\text{Fsubst } q \ \sigma))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{subst-equiv-simp:Fsubst } (\text{Equiv } p \ q) \ \sigma = (\text{Equiv } (\text{Fsubst } p \ \sigma) (\text{Fsubst } q \ \sigma))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{subst-box-simp:Fsubst } (\text{Box } p \ q) \ \sigma = (\text{Box } (\text{Psubst } p \ \sigma) (\text{Fsubst } q \ \sigma))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{pfsbst-box-simp:PFsubst } (\text{Box } p \ q) \ \sigma = (\text{Box } (\text{PPsubst } p \ \sigma) (\text{PFsubst } q \ \sigma))$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{pfsbst-imp-simp:PFsubst } (\text{Implies } p \ q) \ \sigma = (\text{Implies } (\text{PFsubst } p \ \sigma) (\text{PFsubst } q \ \sigma))$   
 $\langle \text{proof} \rangle$

**definition**  $\text{SystemDWSbst}::('sf, 'sc, 'sz) \text{ subst}$

**where**  $\text{SystemDWSbst} = \{\ \ S\text{Functions} = (\lambda f. \ \text{None}),$

$\ S\text{Predicates} = (\lambda-. \ \text{None}),$

$\ S\text{Contexts} = (\lambda C. \ \text{if } C = \text{pid1} \ \text{then } \text{Some } (\text{And } (\text{Geq } (\text{Const } 0) (\text{Const } 0)) (\text{Geq } (\text{Function } \text{fid1 } \text{empty}) (\text{Const } 0))) \ \text{else } \text{None}),$

$\ S\text{Programs} = (\lambda-. \ \text{None}),$

$\ S\text{ODEs} = (\lambda c. \ \text{if } c = \text{vid1} \ \text{then } \text{Some } (\text{OProd } (\text{OSing } \text{vid1} (\text{Function } \text{fid1 } \text{empty})) (\text{OSing } \text{vid2} (\text{Var } \text{vid1}))) \ \text{else } \text{None})$

⌋

**definition**  $\text{SystemCESbst}::('sf, 'sc, 'sz) \text{ subst}$

**where**  $\text{SystemCESbst} = \{\ \ S\text{Functions} = (\lambda f. \ \text{None}),$

$\ S\text{Predicates} = (\lambda-. \ \text{None}),$

$\ S\text{Contexts} = (\lambda C. \ \text{if } C = \text{pid1} \ \text{then } \text{Some}(\text{Implies}(\text{And } (\text{Geq } (\text{Const } 0) (\text{Const } 0)) (\text{Geq } (\text{Function } \text{fid1 } \text{empty}) (\text{Const } 0))) ([[\text{DiffAssign } \text{vid1} (\text{Function } \text{fid1 } \text{empty})]](\text{Predicational } (\text{Inr } ()))) \ \text{else } \text{None}),$

$\ S\text{Programs} = (\lambda-. \ \text{None}),$

$\ S\text{ODEs} = (\lambda-. \ \text{None})$

⌋

**lemma**  $\text{SystemCESbstOK}:$

$\text{step-ok}$

$(([\ ]), [\text{Equiv } (\text{Implies}(\text{And } (\text{Geq } (\text{Const } 0) (\text{Const } 0)) (\text{Geq } (\text{Function } \text{fid1 } \text{empty}) (\text{Const } 0))) ([[\text{DiffAssign } \text{vid1} (\text{Function } \text{fid1 } \text{empty})]](\text{SystemCEFml1}))$   
 $(\text{Implies}(\text{And } (\text{Geq } (\text{Const } 0) (\text{Const } 0)) (\text{Geq } (\text{Function } \text{fid1 } \text{empty}) (\text{Const$

```

0))) ([[DiffAssign vid1 (Function fid1 empty)]] ( (SystemCEFml2))))
    ]],
    ([,[]))

    0
    (CE SystemCEFml1 SystemCEFml2 SystemCESubst)
  ⟨proof⟩
definition SystemDiffAssignSubst::('sf, 'sc, 'sz) subst
where SystemDiffAssignSubst = [] SFunctions = (λf. None),
    SPredicates = (λp. 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  
 ⟨proof⟩

```

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)
AX0/CloseId 0 0
    ,(Suc 0, CloseId 1 0)
AX0/Rrule ImPLYR 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 SystemDWSbst) — 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 SystemCESbst) — 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))]]))  
 ⟨proof⟩

**lemma** *SystemSound-lemma:sound (proof-result SystemProof)*  
 ⟨proof⟩

**lemma** *system-sound:sound ([], SystemConcl)*

*<proof>*

**lemma** *DIAnd-result-correct:proof-result* (*proof-take 61 DIAndProof*) = *DIAnd*  
*<proof>*

**theorem** *DIAnd-sound: sound DIAnd*  
*<proof>*

**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] B. Bohrer, V. Rahli, I. Vukotic, M. Völpl, 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.