Differential-Dynamic-Logic

Rose Bohrer

March 17, 2025

Abstract

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

Published in [1]

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

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

Contents

theory *Ids*
imports *Complex-Main*
begin

1 Identifier locale

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

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

(*fid2* = *fid1*) = *False* (*fid3* = *fid2*) = *False* (*fid3* = *fid1*) = *False*
(*pid2* = *pid1*) = *False* (*pid3* = *pid2*) = *False* (*pid3* = *pid1*) = *False*
(*pid4* = *pid1*) = *False* (*pid4* = *pid2*) = *False* (*pid4* = *pid3*) = *False*
⟨*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*:($\bigwedge i.\ v\$i = w\$i) \Longrightarrow (v = w)$
⟨*proof*⟩

lemma *norm-axis*: *norm* (*axis i x*) = *norm x*
⟨*proof*⟩

lemma *bounded-linear-axis*: *bounded-linear* (*axis i*)
⟨*proof*⟩

lemma *bounded-linear-vec*:
fixes *f*::('*a*::*finite*) ⇒ '*b*::*real-normed-vector* ⇒ '*c*::*real-normed-vector*
assumes *bounds*:⋀*i*. *bounded-linear* (*f i*)
shows *bounded-linear* (λ*x*. χ *i*. *f i x*)
⟨*proof*⟩

lift-definition *blinfun-vec*::('*a*::*finite* ⇒ '*b*::*real-normed-vector* $\Rightarrow_L$ *real*) ⇒ '*b* $\Rightarrow_L$ (*real* ^ '*a*) **is** (λ(*f*::('*a* ⇒ '*b* ⇒ *real*)) (*x*::'*b*). χ (*i*::'*a*). *f i x*)
⟨*proof*⟩

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

lemma *continuous-blinfun-vec*:(⋀*i*. *continuous-on UNIV* (*blinfun-apply* (*g i*))) ⟹ *continuous-on UNIV* (*blinfun-vec g*)
⟨*proof*⟩

lemma *blinfun-elim*:⋀*g*. (*blinfun-apply* (*blinfun-vec g*)) = (λ*x*. χ *i*. *g i x*)
⟨*proof*⟩

lemma *sup-plus*:
fixes *f g*::('*b*::*metric-space*) ⇒ *real*
assumes *nonempty*:*R* ≠ {}
assumes *bddf*:*bdd-above* (*f* ‘ *R*)

assumes *bddg*:*bdd-above* (*g* ‘ *R*)
shows (*SUP x*∈*R*. *f x* + *g x*) ≤ (*SUP x*∈*R*. *f x*) + (*SUP x*∈*R*. *g x*)
⟨*proof*⟩

lemma *continuous-blinfun-vec′*:
fixes *f*::′*a*::{*finite*,*linorder*} ⇒ ′*b*::{*metric-space*, *real-normed-vector*,*abs*} ⇒ ′*b* $\Rightarrow_L$ *real*
fixes *S*::′*b set*
assumes *conts*:⋀*i*. *continuous-on UNIV* (*f i*)
shows *continuous-on UNIV* (λ*x*. *blinfun-vec* (λ *i*. *f i x*))
⟨*proof*⟩

lemma *has-derivative-vec*[*derivative-intros*]:
assumes ⋀*i*. ((λ*x*. *f i x*) *has-derivative* (λ*h*. *f′ i h*)) *F*
shows ((λ*x*. χ *i*. *f i x*) *has-derivative* (λ*h*. χ *i*. *f′ i h*)) *F*
⟨*proof*⟩

lemma *has-derivative-proj*:
fixes *j*::(′*a*::*finite*)
fixes *f*::′*a* ⇒ *real* ⇒ *real*
assumes *assm*:((λ*x*. χ *i*. *f i x*) *has-derivative* (λ*h*. χ *i*. *f′ i h*)) *F*
shows ((λ*x*. *f j x*) *has-derivative* (λ*h*. *f′ j h*)) *F*
⟨*proof*⟩

lemma *has-derivative-proj′*:
fixes *i*::′*a*::*finite*
shows ∀ *x*. ((λ *x*. *x* $ *i*) *has-derivative* (λ*x*::(*real*^′*a*). *x* $ *i*)) (*at x*)
⟨*proof*⟩

lemma *constant-when-zero*:
fixes *v*::*real* ⇒ (*real*, ′*i*::*finite*) *vec*
assumes *x0*: (*v t0*) $ *i* = *x0*
assumes *sol*: (*v solves-ode f*) *T S*
assumes *f0*: ⋀*s x*. *s* ∈ *T* ⟹ *f s x* $ *i* = *0*
assumes *t0*:*t0* ∈ *T*
assumes *t*:*t* ∈ *T*
assumes *convex*:*convex T*
shows *v t* $ *i* = *x0*
⟨*proof*⟩

lemma
solves-ode-subset:
assumes *x*: (*x solves-ode f*) *T X*
assumes *s*: *S* ⊆ *T*
shows (*x solves-ode f*) *S X*
⟨*proof*⟩

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-on-open.flow UNIV* ($\lambda t.\ \lambda x.\ \chi$ (*i*::(*'sz*::*finite*)). *0*) *UNIV 0* (χ *i*. *x*) *t*) $ *i*
⟨*proof*⟩

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

lemma *MVT-ivl'*:
fixes *f*::*'a*::*ordered-euclidean-space*⇒*'b*::*ordered-euclidean-space*
assumes *fderiv*: ($\bigwedge x.\ x \in D \Longrightarrow$ (*f has-derivative J x*) (*at x within D*))
assumes *J-ivl*: $\bigwedge x.\ x \in D \Longrightarrow J\ x\ (a - b) \geq J0$
assumes *line-in*: $\bigwedge x.\ x \in \{0..1\} \Longrightarrow b + x *_R (a - b) \in D$
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 $'c \Rightarrow ('a,\ 'c)$ *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* ⇒ (*'a*, *'c*) *trm* (‹$*f*›)
| *Plus* (*'a*, *'c*) *trm* (*'a*, *'c*) *trm*
| *Times* (*'a*, *'c*) *trm* (*'a*, *'c*) *trm*
— A (real-valued) variable standing for a differential, such as x', given meaning by the state
— and modified by programs.
| *DiffVar 'c* (‹$''›)
— The differential of an arbitrary term $(\vartheta)'$
| *Differential* (*'a*, *'c*) *trm*

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

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

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

— Derived forms
definition *Or* :: (*'a*, *'b*, *'c*) *formula* ⇒ (*'a*, *'b*, *'c*) *formula* ⇒ (*'a*, *'b*, *'c*) *formula* (**infixl** ‹||› *7*)
where *Or P Q* = *Not* (*And* (*Not P*) (*Not Q*))

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

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

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

definition *Equals* :: (*'a*, *'c*) *trm* ⇒ (*'a*, *'c*) *trm* ⇒ (*'a*, *'b*, *'c*) *formula*

where $\mathit{Equals}\ \vartheta\ \vartheta' = ((\mathit{Geq}\ \vartheta\ \vartheta')\ \&\&\ (\mathit{Geq}\ \vartheta'\ \vartheta))$

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

definition $\mathit{Box} :: ('a,\ 'b,\ 'c)\ \mathit{hp} \Rightarrow ('a,\ 'b,\ 'c)\ \mathit{formula} \Rightarrow ('a,\ 'b,\ 'c)\ \mathit{formula}$
(‹([[-]]-)› *10*)
where $\mathit{Box}\ \alpha\ P = \mathit{Not}\ (\mathit{Diamond}\ \alpha\ (\mathit{Not}\ P))$

definition $\mathit{TT} ::('a,'b,'c)\ \mathit{formula}$
where $\mathit{TT} = \mathit{Geq}\ (\mathit{Const}\ 0)\ (\mathit{Const}\ 0)$

definition $\mathit{FF} ::('a,'b,'c)\ \mathit{formula}$
where $\mathit{FF} = \mathit{Geq}\ (\mathit{Const}\ 0)\ (\mathit{Const}\ 1)$

type-synonym $('a,'b,'c)\ \mathit{sequent} = ('a,'b,'c)\ \mathit{formula\ list} * ('a,'b,'c)\ \mathit{formula\ list}$
— Rule: assumptions, then conclusion
type-synonym $('a,'b,'c)\ \mathit{rule} = ('a,'b,'c)\ \mathit{sequent\ list} * ('a,'b,'c)\ \mathit{sequent}$

— silliness to enable proving disequality lemmas
primrec $\mathit{sizeF}::('sf,'sc,\ 'sz)\ \mathit{formula} \Rightarrow \mathit{nat}$
  and $\mathit{sizeP}::('sf,'sc,\ 'sz)\ \mathit{hp} \Rightarrow \mathit{nat}$
where
  $\mathit{sizeP}\ (\mathit{Pvar}\ a) = 1$
| $\mathit{sizeP}\ (\mathit{Assign}\ x\ \vartheta) = 1$
| $\mathit{sizeP}\ (\mathit{DiffAssign}\ x\ \vartheta) = 1$
| $\mathit{sizeP}\ (\mathit{Test}\ \varphi) = \mathit{Suc}\ (\mathit{sizeF}\ \varphi)$
| $\mathit{sizeP}\ (\mathit{EvolveODE}\ \mathit{ODE}\ \varphi) = \mathit{Suc}\ (\mathit{sizeF}\ \varphi)$
| $\mathit{sizeP}\ (\mathit{Choice}\ \alpha\ \beta) = \mathit{Suc}\ (\mathit{sizeP}\ \alpha + \mathit{sizeP}\ \beta)$
| $\mathit{sizeP}\ (\mathit{Sequence}\ \alpha\ \beta) = \mathit{Suc}\ (\mathit{sizeP}\ \alpha + \mathit{sizeP}\ \beta)$
| $\mathit{sizeP}\ (\mathit{Loop}\ \alpha) = \mathit{Suc}\ (\mathit{sizeP}\ \alpha)$
| $\mathit{sizeF}\ (\mathit{Geq}\ p\ q) = 1$
| $\mathit{sizeF}\ (\mathit{Prop}\ p\ \mathit{args}) = 1$
| $\mathit{sizeF}\ (\mathit{Not}\ p) = \mathit{Suc}\ (\mathit{sizeF}\ p)$
| $\mathit{sizeF}\ (\mathit{And}\ p\ q) = \mathit{sizeF}\ p + \mathit{sizeF}\ q$
| $\mathit{sizeF}\ (\mathit{Exists}\ x\ p) = \mathit{Suc}\ (\mathit{sizeF}\ p)$
| $\mathit{sizeF}\ (\mathit{Diamond}\ p\ q) = \mathit{Suc}\ (\mathit{sizeP}\ p + \mathit{sizeF}\ q)$
| $\mathit{sizeF}\ (\mathit{InContext}\ C\ \varphi) = \mathit{Suc}\ (\mathit{sizeF}\ \varphi)$

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

named-theorems *expr-diseq Structural disequality rules for expressions*
lemma $[\mathit{expr\text{-}diseq}]{:}p \neq \mathit{And}\ p\ q$ $\langle \mathit{proof} \rangle$
lemma $[\mathit{expr\text{-}diseq}]{:}q \neq \mathit{And}\ p\ q$ $\langle \mathit{proof} \rangle$
lemma $[\mathit{expr\text{-}diseq}]{:}p \neq \mathit{Not}\ p$ $\langle \mathit{proof} \rangle$
lemma $[\mathit{expr\text{-}diseq}]{:}p \neq \mathit{Or}\ p\ q$ $\langle \mathit{proof} \rangle$
lemma $[\mathit{expr\text{-}diseq}]{:}q \neq \mathit{Or}\ p\ q$ $\langle \mathit{proof} \rangle$
lemma $[\mathit{expr\text{-}diseq}]{:}p \neq \mathit{Implies}\ p\ q$ $\langle \mathit{proof} \rangle$

lemma [*expr-diseq*]:$q \neq$ *Implies p q* ⟨*proof*⟩
lemma [*expr-diseq*]:$p \neq$ *Equiv p q* ⟨*proof*⟩
lemma [*expr-diseq*]:$q \neq$ *Equiv p q* ⟨*proof*⟩
lemma [*expr-diseq*]:$p \neq$ *Exists x p* ⟨*proof*⟩
lemma [*expr-diseq*]:$p \neq$ *Diamond a p* ⟨*proof*⟩
lemma [*expr-diseq*]:$p \neq$ *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* ≡ λ*i*.(*Const 0*)

— Function argument tuple with (effectively) one argument, where all others have a constant value.
fun *singleton* :: (*'a*, *'sz*) *trm* ⇒ (*'sz* ⇒ (*'a*, *'sz*) *trm*)
where *singleton t i* = (*if i* = *vid1 then t else* (*Const 0*))

lemma *expand-singleton*:*singleton t* = (λ*i*. (*if i* = *vid1 then t else* (*Const 0*)))
⟨*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* $\vartheta_1 \Longrightarrow$ *dfree* $\vartheta_2 \Longrightarrow$ *dfree* (*Plus* ϑ_1 ϑ_2)
| *dfree-Times*: *dfree* $\vartheta_1 \Longrightarrow$ *dfree* $\vartheta_2 \Longrightarrow$ *dfree* (*Times* ϑ_1 ϑ_2)

inductive *dsafe* :: (*'a*, *'c*) *trm* ⇒ *bool*
where
 dsafe-Var: *dsafe* (*Var i*)
| *dsafe-Const*: *dsafe* (*Const r*)
| *dsafe-Fun*: ($\bigwedge$*i*. *dsafe* (*args i*)) $\Longrightarrow$ *dsafe* (*Function i args*)
| *dsafe-Plus*: *dsafe* $\vartheta_1 \Longrightarrow$ *dsafe* $\vartheta_2 \Longrightarrow$ *dsafe* (*Plus* ϑ_1 ϑ_2)
| *dsafe-Times*: *dsafe* $\vartheta_1 \Longrightarrow$ *dsafe* $\vartheta_2 \Longrightarrow$ *dsafe* (*Times* ϑ_1 ϑ_2)
| *dsafe-Diff*: *dfree* $\vartheta \Longrightarrow$ *dsafe* (*Differential* ϑ)
| *dsafe-DiffVar*: *dsafe* ($' *i*)

— Explictly-written variables that are bound by the ODE. Needed to compute whether
— ODE's are valid (e.g. whether they bind the same variable twice)
fun *ODE-dom*::(*'a*, *'c*) *ODE* ⇒ *'c set*
where
 ODE-dom (*OVar c*) = {}
| *ODE-dom* (*OSing x* ϑ) = {*x*}
| *ODE-dom* (*OProd ODE1 ODE2*) = *ODE-dom ODE1* ∪ *ODE-dom ODE2*

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

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

| *ffree p* ⟹ *ffree* (*Exists x p*)
| *hpfree a* ⟹ *ffree p* ⟹ *ffree* (*Diamond a p*)
| *ffree* (*Predicational P*)
| *dfree t1* ⟹ *dfree t2* ⟹ *ffree* (*Geq t1 t2*)

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

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

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

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

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

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

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

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

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

— Basic reasoning principles about syntactic constructs, including inductive principles
lemma *dfree-is-dsafe*: *dfree* ϑ ⟹ *dsafe* ϑ
⟨*proof*⟩

lemma *hp-induct* [*case-names Var Assign DiffAssign Test Evolve Choice Compose Star*]:
(⋀*x*. *P* ($α *x*)) ⟹
 (⋀*x1 x2*. *P* (*x1* := *x2*)) ⟹
 (⋀*x1 x2*. *P* (*DiffAssign x1 x2*)) ⟹
 (⋀*x*. *P* (*? x*)) ⟹
 (⋀*x1 x2*. *P* (*EvolveODE x1 x2*)) ⟹
 (⋀*x1 x2*. *P x1* ⟹ *P x2* ⟹ *P* (*x1* ∪∪ *x2*)) ⟹
 (⋀*x1 x2*. *P x1* ⟹ *P x2* ⟹ *P* (*x1* ;; *x2*)) ⟹
 (⋀*x*. *P x* ⟹ *P x*∗∗) ⟹
  P hp
⟨*proof*⟩

lemma *fml-induct*:
(⋀*t1 t2*. *P* (*Geq t1 t2*))
⟹ (⋀*p args*. *P* (*Prop p args*))
⟹ (⋀*p*. *P p* ⟹ *P* (*Not p*))
⟹ (⋀*p q*. *P p* ⟹ *P q* ⟹ *P* (*And p q*))

$\Longrightarrow (\bigwedge x\ p.\ P\ p \Longrightarrow P\ (Exists\ x\ p))$
$\Longrightarrow (\bigwedge a\ p.\ P\ p \Longrightarrow P\ (Diamond\ a\ p))$
$\Longrightarrow (\bigwedge C\ p.\ P\ p \Longrightarrow P\ (InContext\ C\ p))$
$\Longrightarrow P\ \varphi$
⟨*proof*⟩

context *ids* **begin**
lemma *proj-sing1*:(*singleton* ϑ *vid1*) = ϑ
⟨*proof*⟩

lemma *proj-sing2*:*vid1* $\neq$ *y* $\Longrightarrow$ (*singleton* ϑ *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 ν, interpretations I and the semantic functions $[[\psi]]I$, $[[\theta]]I\nu$, $[[\alpha]]I$, which are represented by the Isabelle functions `fml_sem`, `dterm_sem` and `prog_sem`, respectively.

3.3 States

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

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

type-synonym *'a Rvec* = *real*^(*'a*::*finite*)
— A state specifies one vector of values for unprimed variables x and a second vector for x'

type-synonym $'a$ *state* = $'a$ *Rvec* × $'a$ *Rvec*
— $'a$ *simple-state* is half a state - either the xs or the x's
type-synonym $'a$ *simple-state* = $'a$ *Rvec*

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

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

— Agreement lemmas
lemma *agree-nil*: *Vagree* ν ω {}
⟨*proof*⟩

lemma *agree-supset*: *A* ⊇ *B* ⟹ *Vagree* ν ν' *A* ⟹ *Vagree* ν ν' *B*
⟨*proof*⟩

lemma *VSagree-nil*: *VSagree* ν ω {}
⟨*proof*⟩

lemma *VSagree-supset*: *A* ⊇ *B* ⟹ *VSagree* ν ν' *A* ⟹ *VSagree* ν ν' *B*
⟨*proof*⟩

lemma *VSagree-UNIV-eq*: *VSagree A B UNIV* ⟹ *A* = *B*
⟨*proof*⟩

lemma *agree-comm*: ⋀*A B V*. *Vagree A B V* ⟹ *Vagree B A V* ⟨*proof*⟩

lemma *agree-sub*: ⋀ν ω *A B* . *A* ⊆ *B* ⟹ *Vagree* ν ω *B* ⟹ *Vagree* ν ω *A*
⟨*proof*⟩

lemma *agree-UNIV-eq*: ⋀ν ω. *Vagree* ν ω *UNIV* ⟹ ν = ω
⟨*proof*⟩

lemma *agree-UNIV-fst*: ⋀ν ω. *Vagree* ν ω (*Inl* ‘ *UNIV*) ⟹ (*fst* ν) = (*fst* ω)
⟨*proof*⟩

lemma *agree-UNIV-snd*: ⋀ν ω. *Vagree* ν ω (*Inr* ‘ *UNIV*) ⟹ (*snd* ν) = (*snd* ω)
⟨*proof*⟩

lemma *Vagree-univ*: ⋀*a b c d*. *Vagree* (*a*,*b*) (*c*,*d*) *UNIV* ⟹ *a* = *c* ∧ *b* = *d*
⟨*proof*⟩

lemma *agree-union*: ⋀ν ω *A B*. *Vagree* ν ω *A* ⟹ *Vagree* ν ω *B* ⟹ *Vagree* ν ω (*A* ∪ *B*)
⟨*proof*⟩

lemma *agree-trans*: $Vagree\ \nu\ \mu\ A \Longrightarrow Vagree\ \mu\ \omega\ B \Longrightarrow Vagree\ \nu\ \omega\ (A \cap B)$
⟨*proof*⟩

lemma *agree-refl*: $Vagree\ \nu\ \nu\ A$
⟨*proof*⟩

lemma *VSagree-sub*: $\bigwedge \nu\ \omega\ A\ B\ .\ A \subseteq B \Longrightarrow VSagree\ \nu\ \omega\ B \Longrightarrow VSagree\ \nu\ \omega\ A$
⟨*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>)` : $R^m \to R$ for every function in every state. The proposition `(is_interp I)` says that such a derivative actually exists and is continuous (i.e. all functions are C1-continuous) without saying what the exact derivative is.

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

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

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

— For an interpretation to be valid, all functions must be differentiable everywhere.
definition *is-interp* :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* ⇒ *bool*
where *is-interp I* ≡
∀ *x*. ∀ *i*. ((*FDERIV* (*Functions I i*) *x* :> (*FunctionFrechet I i x*)) ∧ *continuous-on UNIV* (λ*x*. *Blinfun* (*FunctionFrechet I i x*)))

lemma *is-interpD*: *is-interp I* ⟹ ∀ *x*. ∀ *i*. (*FDERIV* (*Functions I i*) *x* :> (*FunctionFrechet I i x*))
⟨*proof*⟩
definition *Iagree* :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* ⇒ (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* ⇒ (*'a* + *'b* + *'c*) *set* ⇒ *bool*
where *Iagree I J V* ≡
(∀ *i*∈*V*.

$(\forall x.\ i = Inl\ x \longrightarrow Functions\ I\ x = Functions\ J\ x) \wedge$
$(\forall x.\ i = Inr\ (Inl\ x) \longrightarrow Contexts\ I\ x = Contexts\ J\ x) \wedge$
$(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow Predicates\ I\ x = Predicates\ J\ x) \wedge$
$(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow Programs\ I\ x = Programs\ J\ x) \wedge$
$(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow ODEs\ I\ x = ODEs\ J\ x) \wedge$
$(\forall x.\ i = Inr\ (Inr\ x) \longrightarrow ODEBV\ I\ x = ODEBV\ J\ x))$

lemma *Iagree-Func*:*Iagree I J V* $\Longrightarrow$ *Inl f* $\in$ *V* $\Longrightarrow$ *Functions I f* = *Functions J f*
⟨*proof*⟩

lemma *Iagree-Contexts*:*Iagree I J V* $\Longrightarrow$ *Inr* (*Inl C*) $\in$ *V* $\Longrightarrow$ *Contexts I C* = *Contexts J C*
⟨*proof*⟩

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

lemma *Iagree-Prog*:*Iagree I J V* $\Longrightarrow$ *Inr* (*Inr a*) $\in$ *V* $\Longrightarrow$ *Programs I a* = *Programs J a*
⟨*proof*⟩

lemma *Iagree-ODE*:*Iagree I J V* $\Longrightarrow$ *Inr* (*Inr a*) $\in$ *V* $\Longrightarrow$ *ODEs I a* = *ODEs J a*
⟨*proof*⟩

lemma *Iagree-comm*:$\bigwedge$*A B V*. *Iagree A B V* $\Longrightarrow$ *Iagree B A V*
⟨*proof*⟩

lemma *Iagree-sub*:$\bigwedge$*I J A B* . *A* $\subseteq$ *B* $\Longrightarrow$ *Iagree I J B* $\Longrightarrow$ *Iagree I J A*
⟨*proof*⟩

lemma *Iagree-refl*:*Iagree I I A*
⟨*proof*⟩
primrec *sterm-sem* :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* $\Rightarrow$ (*'a*, *'c*) *trm* $\Rightarrow$ *'c simple-state* $\Rightarrow$ *real*
where
sterm-sem I (*Var x*) *v* = *v* $ *x*
| *sterm-sem I* (*Function f args*) *v* = *Functions I f* (χ *i*. *sterm-sem I* (*args i*) *v*)
| *sterm-sem I* (*Plus t1 t2*) *v* = *sterm-sem I t1 v* + *sterm-sem I t2 v*
| *sterm-sem I* (*Times t1 t2*) *v* = *sterm-sem I t1 v* * *sterm-sem I t2 v*
| *sterm-sem I* (*Const r*) *v* = *r*
| *sterm-sem I* ($' *c*) *v* = *undefined*
| *sterm-sem I* (*Differential d*) *v* = *undefined*

— *frechet I* ϑ ν syntactically computes the frechet derivative of the term ϑ in the interpretation
— *I* at state ν (containing only the unprimed variables). The frechet derivative is a

— linear map from the differential state ν to reals.
primrec *frechet* :: ('a::finite, 'b::finite, 'c::finite) *interp* ⇒ ('a, 'c) *trm* ⇒ 'c *simple-state* ⇒ 'c *simple-state* ⇒ *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* (χ *i*. *sterm-sem I* (*args i*) *v*) (χ *i*. *frechet I* (*args i*) *v 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* =
($\lambda v'$. *sterm-sem I t1 v* ∗ *frechet I t2 v v'* + *frechet I t1 v v'* ∗ *sterm-sem I t2 v*)
| *frechet I* (*Const r*) *v* = ($\lambda v'$. *0*)
| *frechet I* ($' *c*) *v* = *undefined*
| *frechet I* (*Differential d*) *v* = *undefined*

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

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

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

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

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

by the *OVar*
| *ODE-sem-OProd*:*ODE-sem I* (*OProd ODE1 ODE2*) = ($\lambda\nu$. *ODE-sem I ODE1* ν + *ODE-sem I ODE2* ν)

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

fun *semBV* ::('a, 'b,'c) *interp* ⇒ ('a, 'c) *ODE* ⇒ ('c + 'c) *set*
where *semBV I ODE* = *Inl* ‘ (*ODE-vars I ODE*) ∪ *Inr* ‘ (*ODE-vars I ODE*)

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

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

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

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

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

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

— Semantics for formulas, differential formulas, programs.
fun *fml-sem* :: ('a::*finite*, 'b::*finite*, 'c::*finite*) *interp* ⇒ ('a::*finite*, 'b::*finite*, 'c::*finite*) *formula* ⇒ 'c::*finite state set* **and**
prog-sem :: ('a::*finite*, 'b::*finite*, 'c::*finite*) *interp* ⇒ ('a::*finite*, 'b::*finite*, 'c::*finite*) *hp* ⇒ ('c::*finite state* ∗ 'c::*finite state*) *set*
where
fml-sem I (*Geq t1 t2*) = {*v*. *dterm-sem I t1 v* ≥ *dterm-sem I t2 v*}
| *fml-sem I* (*Prop P terms*) = {ν. *Predicates I P* (χ *i*. *dterm-sem I* (*terms i*) ν)}
| *fml-sem I* (*Not* φ) = {*v*. *v* ∉ *fml-sem I* φ}

| *fml-sem I (And φ ψ) = fml-sem I φ ∩ fml-sem I ψ*
| *fml-sem I (Exists x φ) = {v | v r. (repv v x r) ∈ fml-sem I φ}*
| *fml-sem I (Diamond α φ) = {ν | ν ω. (ν, ω) ∈ prog-sem I α ∧ ω ∈ fml-sem I φ}*
| *fml-sem I (InContext c φ) = Contexts I c (fml-sem I φ)*

| *prog-sem I (Pvar p) = Programs I p*
| *prog-sem I (Assign x t) = {(ν, ω). ω = repv ν x (dterm-sem I t ν)}*
| *prog-sem I (DiffAssign x t) = {(ν, ω). ω = repd ν x (dterm-sem I t ν)}*
| *prog-sem I (Test φ) = {(ν, ν) | ν. ν ∈ fml-sem I φ}*
| *prog-sem I (Choice α β) = prog-sem I α ∪ prog-sem I β*
| *prog-sem I (Sequence α β) = prog-sem I α O prog-sem I β*
| *prog-sem I (Loop α) = (prog-sem I α)**
| *prog-sem I (EvolveODE ODE φ) =*
 ({(ν, mk-v I ODE ν (sol t)) | ν sol t.
 t ≥ 0 ∧
 (sol solves-ode (λ-. ODE-sem I ODE)) {0..t} {x. mk-v I ODE ν x ∈ fml-sem I φ} ∧
 sol 0 = fst ν})

context *ids* **begin**
definition *valid* :: *('sf, 'sc, 'sz) formula ⇒ bool*
where *valid φ ≡ (∀ I. ∀ ν. is-interp I ⟶ ν ∈ fml-sem I φ)*
end

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

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

lemma *mk-v-exists:∃ω. Vagree ω ν (− semBV I ODE)*
∧ Vagree ω (mk-xode I ODE sol) (semBV I ODE)
 ⟨*proof*⟩

lemma *mk-v-agree:Vagree (mk-v I ODE ν sol) ν (− semBV I ODE)*
∧ Vagree (mk-v I ODE ν sol) (mk-xode I ODE sol) (semBV I ODE)
 ⟨*proof*⟩

lemma *mk-v-concrete:mk-v I ODE ν sol = ((χ i. (if Inl i ∈ semBV I ODE then sol else (fst ν)) $ i),*
 (χ i. (if Inr i ∈ semBV I ODE then ODE-sem I ODE sol else (snd ν)) $ 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*:
sterm-sem I (*Var x*) = ($\lambda v.\ v$ \$ x)
⟨*proof*⟩

lemma *sconst-case*:
sterm-sem I (*Const r*) = ($\lambda v.\ r$)
⟨*proof*⟩

lemma *sfunction-case*:
sterm-sem I (*Function f args*) = ($\lambda v.$ *Functions I f* (χ *i*. *sterm-sem I* (*args i*) *v*))
⟨*proof*⟩

lemma *splus-case*:
sterm-sem I (*Plus t1 t2*) = ($\lambda v.$ (*sterm-sem I t1 v*) + (*sterm-sem I t2 v*))
⟨*proof*⟩

lemma *stimes-case*:
sterm-sem I (*Times t1 t2*) = ($\lambda v.$ (*sterm-sem I t1 v*) $*$ (*sterm-sem I t2 v*))
⟨*proof*⟩

lemma *or-sem* [*simp*]:
fml-sem I (*Or* φ ψ) = *fml-sem I* φ $\cup$ *fml-sem I* ψ
⟨*proof*⟩

lemma *iff-sem* [*simp*]: ($\nu \in$ *fml-sem I* ($A \leftrightarrow B$))
$\longleftrightarrow$ (($\nu \in$ *fml-sem I A*) $\longleftrightarrow$ ($\nu \in$ *fml-sem I B*))
⟨*proof*⟩

lemma *box-sem* [*simp*]:*fml-sem I* (*Box* α φ) = {ν. $\forall$ ω. (ν, ω) $\in$ *prog-sem I* α $\longrightarrow$ $\omega \in$ *fml-sem I* φ}
⟨*proof*⟩

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

lemma *greater-sem*[*simp*]:*fml-sem I* (*Greater* ϑ ϑ') = {*v*. *dterm-sem I* ϑ *v* > *dterm-sem I* ϑ' *v*}
⟨*proof*⟩

lemma *loop-sem*:*prog-sem I* (*Loop* α) = (*prog-sem I* α)*
⟨*proof*⟩

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

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

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

lemma *tt-sem* [*simp*]:$\textit{fml-sem } I\ TT = UNIV$ ⟨*proof*⟩
lemma *ff-sem* [*simp*]:$\textit{fml-sem } I\ FF = \{\}$ ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

definition *sublist*::′*a list* ⇒ ′*a list* ⇒ *bool*
where $sublist\ A\ B \equiv (\forall x.\ List.member\ A\ x \longrightarrow List.member\ B\ x)$

lemma *sublistI*:$(\bigwedge x.\ List.member\ A\ x \Longrightarrow List.member\ B\ x) \Longrightarrow sublist\ A\ B$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

lemma *soundI-memv'*:($\bigwedge I.$ *is-interp I* $\Longrightarrow$ ($\bigwedge\varphi\ \nu.$ *List.member SG* $\varphi \Longrightarrow \nu \in$ *seq-sem I* φ) $\Longrightarrow$ ($\bigwedge\nu.\ \nu \in$ *seq-sem I C*)) $\Longrightarrow R = (SG,C) \Longrightarrow$ *sound R*
⟨*proof*⟩

lemma *soundD-mem*:*sound* $(SG,C) \Longrightarrow$ ($\bigwedge I.$ *is-interp I* $\Longrightarrow$ ($\bigwedge\varphi.$ *List.member SG* $\varphi \Longrightarrow$ *seq-sem I* φ = *UNIV*) $\Longrightarrow$ *seq-sem I C* = *UNIV*)
⟨*proof*⟩

lemma *soundD-memv*:*sound* $(SG,C) \Longrightarrow$ ($\bigwedge I.$ *is-interp I* $\Longrightarrow$ ($\bigwedge\varphi\ \nu.$ *List.member SG* $\varphi \Longrightarrow \nu \in$ *seq-sem I* φ) $\Longrightarrow$ ($\bigwedge\nu.\ \nu \in$ *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 metavariables.

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

named-theorems *axiom-defs Axiom definitions*

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

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

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

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

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

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

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

([[$\$\alpha$ *vid1*]][[$\$\alpha$ *vid2*]]*Predicational pid1*)

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

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

definition *Vaxiom* :: ('*sf*, '*sc*, '*sz*) *formula*
where [*axiom-defs*]:*Vaxiom* ≡ ($\$\varphi$ *vid1 empty*) → ([[$\$\alpha$ *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* (λy. *if vid1* = *y then Functions I fid1*
(*vec-lambda* (λi. *dterm-sem I* (*empty i*) ν)) *else vec-nth* (*fst* ν) *y*), *snd* ν)
=

dterm-sem I (if i = vid1 then $f fid1 empty else (Const 0)) ν
⟨*proof*⟩

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

⟨*proof*⟩

theorem *assign-valid*: *valid assign-axiom*
⟨*proof*⟩

theorem *diff-assign-valid*: *valid diff-assign-axiom*
⟨*proof*⟩

lemma *mem-to-nonempty*: $\omega \in S \Longrightarrow (S \neq \{\})$
⟨*proof*⟩

lemma *loop-forward*: *ν ∈ fml-sem I ([[$α id1**]]Predicational pid1)*
*⟶ ν ∈ fml-sem I (Predicational pid1&&[[$α id1]][[$α id1**]]Predicational pid1)*
⟨*proof*⟩

lemma *loop-backward*:
*ν ∈ fml-sem I (Predicational pid1 && [[$α id1]][[$α id1**]]Predicational pid1)*
*⟶ ν ∈ fml-sem I ([[$α id1**]]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** *ν*
assumes *is-interp I*
assumes *IS:ν ∈ fml-sem I ([[$α vid1**]](Predicational pid1 →*

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

theorem *I-valid*: *valid Iaxiom*
  ⟨*proof*⟩

theorem *V-valid*: *valid Vaxiom*
  ⟨*proof*⟩

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

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

definition *MP-holds* :: (*'sf*, *'sc*, *'sz*) *formula* ⇒ (*'sf*, *'sc*, *'sz*) *formula* ⇒ *bool*
where *MP-holds* φ ψ ≡ *valid* ($\varphi \rightarrow \psi$) ⟶ *valid* φ ⟶ *valid* ψ

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

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

definition *CE-holds* :: *'sc* ⇒ (*'sf*, *'sc*, *'sz*) *formula* ⇒ (*'sf*, *'sc*, *'sz*) *formula* ⇒ *bool*
where *CE-holds var* φ ψ ≡ *valid* ($\varphi \leftrightarrow \psi$)
  ⟶ *valid* (*InContext var* φ ↔ *InContext var* ψ)

4.2 Soundness proofs for rules

theorem *G-sound*: *G-holds* φ α
  ⟨*proof*⟩

theorem *Skolem-sound*: *Skolem-holds* φ *var*
  ⟨*proof*⟩

theorem *MP-sound*: *MP-holds* φ ψ
  ⟨*proof*⟩

lemma *CT-lemma*:⋀*I*::(*'sf*::*finite*, *'sc*::*finite*, *'sz*::{*finite*,*linorder*}) *interp*. ⋀ *a*::(*real*, *'sz*) *vec*. ⋀ *b*::(*real*, *'sz*) *vec*. ∀ *I*::(*'sf*,*'sc*,*'sz*) *interp*. *is-interp I* ⟶ (∀ *a b*. *dterm-sem I* ϑ (*a*, *b*) = *dterm-sem I* ϑ' (*a*, *b*)) ⟹
    is-interp I ⟹
    Functions I var (*vec-lambda* (λ*i*. *dterm-sem I* (*if i* = *vid1 then* ϑ *else*

$(Const\ 0))\ (a,\ b))) =$

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

⟨*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 $(\lambda(v::'a\ Rvec).\ v \cdot axis\ i\ 1) = (\lambda(v::'a\ Rvec).\ v\ \$\ i)$
⟨*proof*⟩

theorem *svar-deriv*:
fixes $x::\ 'sv::finite$ **and** $\nu::\ 'sv\ Rvec$ **and** $F::real\ filter$
shows $((\lambda v.\ v\ \$\ x)\ has\text{-}derivative\ (\lambda v'.\ v' \cdot (\chi\ i.\ if\ i = x\ then\ 1\ else\ 0)))\ (at\ \nu)$
⟨*proof*⟩

lemma *function-case-inner*:
assumes *good-interp*:
$(\forall x\ i.\ (Functions\ I\ i\ has\text{-}derivative\ FunctionFrechet\ I\ i\ x)\ (at\ x))$
assumes $IH:((\lambda v.\ \chi\ i.\ sterm\text{-}sem\ I\ (args\ i)\ v)$
$has\text{-}derivative\ (\lambda\ v.\ (\chi\ i.\ frechet\ I\ (args\ i)\ \nu\ v)))\ (at\ \nu)$
shows $((\lambda v.\ Functions\ I\ f\ (\chi\ i.\ sterm\text{-}sem\ I\ (args\ i)\ v))$
$has\text{-}derivative\ (\lambda v.\ frechet\ I\ (\$f\ f\ args)\ \nu\ v))\ (at\ \nu)$

⟨*proof*⟩

lemma *func-lemma2*:($\forall x\ i.$ (*Functions I i has-derivative* (*THE* f'. $\forall x.$ (*Functions I i has-derivative* f' *x*) (*at x*)) *x*) (*at x*) ∧
continuous-on UNIV ($\lambda x.$ *Blinfun* ((*THE* f'. $\forall x.$ (*Functions I i has-derivative* f' *x*) (*at x*)) *x*))) $\Longrightarrow$
($\bigwedge\vartheta.$ $\vartheta \in$ *range args* $\Longrightarrow$ (*sterm-sem I* ϑ *has-derivative frechet I* ϑ ν) (*at* ν)) $\Longrightarrow$
(($\lambda v.$ *Functions I f* (*vec-lambda*($\lambda i.$ *sterm-sem I* (*args i*) *v*))) *has-derivative* ($\lambda v'.$ (*THE* f'. $\forall x.$ (*Functions I f has-derivative* f' *x*) (*at x*)) (χ *i. sterm-sem I* (*args i*) ν) (χ *i. frechet I* (*args i*) ν v'))) (*at* ν)
⟨*proof*⟩

lemma *func-lemma*:
is-interp I $\Longrightarrow$
($\bigwedge\vartheta$ :: ('*a*::*finite*, '*c*::*finite*) *trm*. $\vartheta \in$ *range args* $\Longrightarrow$ (*sterm-sem I* ϑ *has-derivative frechet I* ϑ ν) (*at* ν)) $\Longrightarrow$
(*sterm-sem I* ($*f f args*) *has-derivative frechet I* ($*f f args*) ν) (*at* ν)
⟨*proof*⟩

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

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

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

lemma *sterm-determines-frechet*:
fixes *I* ::('*a1*::*finite*, '*b1*::*finite*, '*c*::*finite*) *interp*
and *J* ::('*a2*::*finite*, '*b2*::*finite*, '*c*::*finite*) *interp*
and $\vartheta 1$ :: ('*a1*::*finite*, '*c*::*finite*) *trm*
and $\vartheta 2$ :: ('*a2*::*finite*, '*c*::*finite*) *trm*
and ν
assumes *good-interp1*:*is-interp I*
assumes *good-interp2*:*is-interp J*
assumes *free1*:*dfree* $\vartheta 1$
assumes *free2*:*dfree* $\vartheta 2$
assumes *sem*:*sterm-sem I* $\vartheta 1$ = *sterm-sem J* $\vartheta 2$
shows *frechet I* $\vartheta 1$ (*fst* ν) (*snd* ν) = *frechet J* $\vartheta 2$ (*fst* ν) (*snd* ν)
⟨*proof*⟩

lemma *the-deriv*:
assumes *deriv*:(*f has-derivative F*) (*at x*)
shows (*THE G.* (*f has-derivative G*) (*at x*)) = *F*

⟨*proof*⟩

lemma *the-all-deriv*:
assumes *deriv*:$\forall x$. (*f has-derivative F x*) (*at x*)
shows (*THE G*. $\forall$ *x*. (*f has-derivative G x*) (*at x*)) = *F*
⟨*proof*⟩

typedef ('*a*, '*c*) *strm* = {ϑ:: ('*a*,'*c*) *trm*. *dfree* ϑ}
morphisms *raw-term simple-term*
⟨*proof*⟩

typedef ('*a*, '*b*, '*c*) *good-interp* = {*I*::('*a*::*finite*,'*b*::*finite*,'*c*::*finite*) *interp*. *is-interp I*}
morphisms *raw-interp good-interp*
⟨*proof*⟩

lemma *frechet-linear*:
assumes *good-interp*:*is-interp I*
fixes *v* ϑ
shows *dfree* $\vartheta \Longrightarrow$ *bounded-linear* (*frechet I* ϑ *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* $\Longrightarrow$ *dfree* $\vartheta \Longrightarrow$ (λv. *Blinfun* ($\lambda v'$. *frechet I* ϑ *v v'*)) = *blin-frechet* (*good-interp I*) (*simple-term* ϑ)
⟨*proof*⟩

lemma *sterm-continuous*:
assumes *good-interp*:*is-interp I*
shows *dfree* $\vartheta \Longrightarrow$ *continuous-on UNIV* (*sterm-sem I* ϑ)
⟨*proof*⟩

lemma *sterm-continuous'*:
assumes *good-interp*:*is-interp I*
shows *dfree* $\vartheta \Longrightarrow$ *continuous-on S* (*sterm-sem I* ϑ)
⟨*proof*⟩

lemma *frechet-continuous*:
fixes *I* :: ('*sf*, '*sc*, '*sz*) *interp*
assumes *good-interp*:*is-interp I*

shows *dfree* $\vartheta \Longrightarrow$ *continuous-on UNIV* (*blin-frechet* (*good-interp I*) (*simple-term* ϑ))
⟨*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 *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* ⇒ '*a set*
where
SIGT (*Var var*) = {}
| *SIGT* (*Const r*) = {}
| *SIGT* (*Function var f*) = {*var*} ∪ (⋃*i*. *SIGT* (*f i*))
| *SIGT* (*Plus t1 t2*) = *SIGT t1* ∪ *SIGT t2*
| *SIGT* (*Times t1 t2*) = *SIGT t1* ∪ *SIGT t2*
| *SIGT* (*DiffVar x*) = {}
| *SIGT* (*Differential t*) = *SIGT t*

primrec *SIGO* :: ('*a*, '*c*) *ODE* ⇒ ('*a* + '*c*) *set*

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

primrec $SIGP\ \ :: ('a, 'b, 'c)\ hp \quad \Rightarrow ('a + 'b + 'c)\ set$
and $\quad SIGF\ \ :: ('a, 'b, 'c)\ formula \Rightarrow ('a + 'b + 'c)\ set$
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* ⇒ '*c set*
where
FVO (*OVar c*) = *UNIV*
| *FVO* (*OSing x* ϑ) = {*x*} ∪ {*x* . *Inl x* ∈ *FVT* ϑ}
| *FVO* (*OProd ODE1 ODE2*) = *FVO ODE1* ∪ *FVO ODE2*

Bound variables of ODEs, formulas, programs

fun *BVO* :: ('*a*, '*c*) *ODE* ⇒ ('*c* + '*c*) *set*
where
BVO (*OVar c*) = *UNIV*
| *BVO* (*OSing x* ϑ) = {*Inl x*, *Inr x*}
| *BVO* (*OProd ODE1 ODE2*) = *BVO ODE1* ∪ *BVO ODE2*

fun *BVF* :: ('*a*, '*b*, '*c*) *formula* ⇒ ('*c* + '*c*) *set*
and *BVP* :: ('*a*, '*b*, '*c*) *hp* ⇒ ('*c* + '*c*) *set*
where
BVF (*Geq f g*) = {}
| *BVF* (*Prop p dfun-args*) = {}
| *BVF* (*Not p*) = *BVF p*
| *BVF* (*And p q*) = *BVF p* ∪ *BVF q*
| *BVF* (*Exists x p*) = {*Inl x*} ∪ *BVF p*
| *BVF* (*Diamond* α *p*) = *BVP* α ∪ *BVF p*
| *BVF* (*InContext C p*) = *UNIV*

| *BVP* (*Pvar a*) = *UNIV*
| *BVP* (*Assign x* ϑ) = {*Inl x*}
| *BVP* (*DiffAssign x* ϑ) = {*Inr x*}
| *BVP* (*Test* φ) = {}
| *BVP* (*EvolveODE ODE* φ) = *BVO ODE*
| *BVP* (*Choice* α β) = *BVP* α ∪ *BVP* β
| *BVP* (*Sequence* α β) = *BVP* α ∪ *BVP* β
| *BVP* (*Loop* α) = *BVP* α

Must-bound variables (of a program)

fun *MBV* :: ('*a*, '*b*, '*c*) *hp* ⇒ ('*c* + '*c*) *set*
where
MBV (*Pvar a*) = {}
| *MBV* (*Choice* α β) = *MBV* α ∩ *MBV* β
| *MBV* (*Sequence* α β) = *MBV* α ∪ *MBV* β
| *MBV* (*Loop* α) = {}
| *MBV* (*EvolveODE ODE* -) = (*Inl* ‘ (*ODE-dom ODE*)) ∪ (*Inr* ‘ (*ODE-dom ODE*))
| *MBV* α = *BVP* α

Free variables of a formula, free variables of a program

fun *FVF* :: ('*a*, '*b*, '*c*) *formula* ⇒ ('*c* + '*c*) *set*
and *FVP* :: ('*a*, '*b*, '*c*) *hp* ⇒ ('*c* + '*c*) *set*
where

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

6.3 Lemmas for reasoning about static semantics

lemma *primify-contains*: $x \in primify\ x$
⟨*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*: $Vagree\ \nu\ \nu'\ (FVT\ (Function\ f\ args)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVT\ (args\ i))$
⟨*proof*⟩

lemma *agree-plus1*: $Vagree\ \nu\ \nu'\ (FVDiff\ (Plus\ t1\ t2)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ t1)$
⟨*proof*⟩

lemma *agree-plus2*: $Vagree\ \nu\ \nu'\ (FVDiff\ (Plus\ t1\ t2)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ t2)$
⟨*proof*⟩

lemma *agree-times1*: $Vagree\ \nu\ \nu'\ (FVDiff\ (Times\ t1\ t2)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ t1)$
⟨*proof*⟩

lemma *agree-times2*: $Vagree\ \nu\ \nu'\ (FVDiff\ (Times\ t1\ t2)) \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ t2)$
⟨*proof*⟩

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

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

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

lemma *sum-unique-nonzero*:
fixes $i::'sv::finite$ **and** $f::'sv \Rightarrow real$
assumes $restZero{:}\bigwedge j.\ j{\in}(UNIV::'sv\ set) \Longrightarrow j \neq i \Longrightarrow f\ j = 0$
shows $(\sum j{\in}(UNIV::'sv\ set).\ f\ j) = f\ i$
⟨*proof*⟩

lemma *coincidence-frechet* :
fixes $I :: ('a::finite, 'b::finite, 'c::finite)\ interp$ **and** $\nu :: 'c\ state$ **and** $\nu'::'c\ state$
shows $dfree\ \vartheta \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ \vartheta) \Longrightarrow frechet\ I\ \ \vartheta\ (fst\ \nu)\ (snd\ \nu) = frechet\ I\ \ \vartheta\ (fst\ \nu')\ (snd\ \nu')$
⟨*proof*⟩

lemma *coincidence-frechet′* :
fixes $I\ J :: ('a::finite, 'b::finite, 'c::finite)\ interp$ **and** $\nu :: 'c\ state$ **and** $\nu'::'c\ state$
shows $dfree\ \vartheta \Longrightarrow Vagree\ \nu\ \nu'\ (FVDiff\ \vartheta) \Longrightarrow Iagree\ I\ J\ \{Inl\ x \mid x.\ x \in (SIGT\ \vartheta)\} \Longrightarrow frechet\ I\ \ \vartheta\ (fst\ \nu)\ (snd\ \nu) = frechet\ J\ \ \vartheta\ (fst\ \nu')\ (snd\ \nu')$
⟨*proof*⟩

lemma *coincidence-dterm*:
fixes $I :: ('a::finite, 'b::finite, 'c::finite)\ interp$ **and** $\nu :: 'c\ state$ **and** $\nu'::'c\ state$
shows $dsafe\ \vartheta \Longrightarrow Vagree\ \nu\ \nu'\ (FVT\ \vartheta) \Longrightarrow dterm\text{-}sem\ I\ \vartheta\ \nu = dterm\text{-}sem\ I\ \vartheta\ \nu'$
⟨*proof*⟩

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

7.2 ODE Coincidence Theorems

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

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

SIGO ODE}) $\Longrightarrow$
ODE-sem I ODE ν = *ODE-sem J ODE* ν'
⟨*proof*⟩

lemma *alt-sem-lemma*:$\bigwedge$ *I*::(*'a*::*finite*,*'b*::*finite*,*'c*::*finite*) *interp*. $\bigwedge$ *ODE*::(*'a*::*finite*,*'c*::*finite*) *ODE*. $\bigwedge$*sol*. $\bigwedge$*t*::*real*. $\bigwedge$ *ab*. *osafe ODE* $\Longrightarrow$
ODE-sem I ODE (*sol t*) = *ODE-sem I ODE* (χ *i*. *if i* $\in$ *FVO ODE then sol t* \$ *i else ab* \$ *i*)
⟨*proof*⟩

lemma *bvo-to-fvo*:*Inl x* $\in$ *BVO ODE* $\Longrightarrow$ *x* $\in$ *FVO ODE*
⟨*proof*⟩

lemma *ode-to-fvo*:*x* $\in$ *ODE-vars I ODE* $\Longrightarrow$ *x* $\in$ *FVO ODE*
⟨*proof*⟩

definition *coincide-hp* :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *hp* $\Rightarrow$ (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* $\Rightarrow$ (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* $\Rightarrow$ *bool*
where *coincide-hp* α *I J* $\longleftrightarrow$ ($\forall$ ν ν' μ *V*. *Iagree I J* (*SIGP* α) $\longrightarrow$ *Vagree* ν ν' *V* $\longrightarrow$ *V* $\supseteq$ (*FVP* α) $\longrightarrow$ (ν, μ) $\in$ *prog-sem I* α $\longrightarrow$ ($\exists$ μ'. (ν', μ') $\in$ *prog-sem J* α $\wedge$ *Vagree* μ μ' (*MBV* α $\cup$ *V*)))

definition *ode-sem-equiv* ::(*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *hp* $\Rightarrow$ (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *interp* $\Rightarrow$ *bool*
where *ode-sem-equiv* α *I* $\longleftrightarrow$
($\forall$ *ODE*::(*'a*::*finite*,*'c*::*finite*) *ODE*. $\forall$ φ::(*'a*::*finite*,*'b*::*finite*,*'c*::*finite*)*formula*. *osafe ODE* $\longrightarrow$ *fsafe* φ $\longrightarrow$
(α = *EvolveODE ODE* φ) $\longrightarrow$
{(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
t $\geq$ *0* $\wedge$
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
VSagree (*sol 0*) (*fst* ν) {*x* | *x*. *Inl x* $\in$ *FVP* (*EvolveODE ODE* φ)}} =
{(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
t $\geq$ *0* $\wedge$
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
sol 0 = *fst* ν})

definition *coincide-hp'* :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *hp* $\Rightarrow$ *bool*
where *coincide-hp'* α $\longleftrightarrow$ ($\forall$ *I J*. *coincide-hp* α *I J* $\wedge$ *ode-sem-equiv* α *I*)

definition *coincide-fml* :: (*'a*::*finite*, *'b*::*finite*, *'c*::*finite*) *formula* $\Rightarrow$ *bool*
where *coincide-fml* φ $\longleftrightarrow$ ($\forall$ ν ν' *I J* . *Iagree I J* (*SIGF* φ) $\longrightarrow$ *Vagree* ν ν' (*FVF* φ) $\longrightarrow$ ν $\in$ *fml-sem I* φ $\longleftrightarrow$ ν' $\in$ *fml-sem J* φ)

lemma *coinc-fml* [*simp*]: *coincide-fml* φ = ($\forall$ ν ν' *I J*. *Iagree I J* (*SIGF* φ) $\longrightarrow$ *Vagree* ν ν' (*FVF* φ) $\longrightarrow$ ν $\in$ *fml-sem I* φ $\longleftrightarrow$ ν' $\in$ *fml-sem J* φ)
⟨*proof*⟩

7.3 Coincidence Theorems for Programs and Formulas

lemma *coincidence-hp-fml*:
fixes α::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *hp*
fixes φ::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *formula*
shows (*hpsafe* $\alpha \longrightarrow$ *coincide-hp'* α) $\wedge$ (*fsafe* $\varphi \longrightarrow$ *coincide-fml* φ)
⟨*proof*⟩

lemma *coincidence-formula*:$\bigwedge\nu\ \nu'$ *I J*. *fsafe* (φ::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *formula*) $\Longrightarrow$ *Iagree I J* (*SIGF* φ) $\Longrightarrow$ *Vagree* $\nu\ \nu'$ (*FVF* φ) $\Longrightarrow$ ($\nu \in$ *fml-sem I* φ $\longleftrightarrow \nu' \in$ *fml-sem J* φ)
⟨*proof*⟩

lemma *coincidence-hp*:
fixes $\nu\ \nu'\ \mu$ *V I J*
assumes *safe*:*hpsafe* (α::('*a*::*finite*, '*b*::*finite*, '*c*::*finite*) *hp*)
assumes *IA*:*Iagree I J* (*SIGP* α)
assumes *VA*:*Vagree* $\nu\ \nu'$ *V*
assumes *sub*:$V \supseteq$ (*FVP* α)
assumes *sem*:(ν, μ) $\in$ *prog-sem I* α
shows ($\exists\,\mu'$. (ν', μ') $\in$ *prog-sem J* α $\wedge$ *Vagree* $\mu\ \mu'$ (*MBV* $\alpha \cup V$))
⟨*proof*⟩

7.4 Corollaries: Alternate ODE semantics definition

lemma *ode-sem-eq*:
fixes *I*::('*a*::*finite*,'*b*::*finite*,'*c*::*finite*) *interp* **and** *ODE*::('*a*,'*c*) *ODE* **and** φ::('*a*,'*b*,'*c*) *formula*
assumes *osafe*:*osafe ODE*
assumes *fsafe*:*fsafe* φ
shows
({(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
$t \geq 0 \wedge$
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
VSagree (*sol 0*) (*fst* ν) {*x* | *x*. *Inl x* $\in$ *FVP* (*EvolveODE ODE* φ)}}) =
({(ν, *mk-v I ODE* ν (*sol t*)) | ν *sol t*.
$t \geq 0 \wedge$
(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* ν *x* $\in$ *fml-sem I* φ} $\wedge$
(*sol 0*) = (*fst* ν)})
⟨*proof*⟩

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

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

⟨*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 state. $\bigwedge\omega$ ::'sz state. *hpsafe* $\alpha \Longrightarrow (\nu, \omega) \in$ *prog-sem I* $\alpha \Longrightarrow$ *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* ≡ *Equals* (*Differential* ($*f fid1 empty*)) (*Const 0*)

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

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

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

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

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

9.2 ODE Axioms

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

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

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

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

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

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

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

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

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

([[*EvolveODE* (*OVar vid1*) (*Prop vid1 empty*)]](*Greater* (*f1 fid1 vid1*) (*f1 fid2 vid1*))))

— [{*1′* = *1*(*1*) & *1*(*1*)}]*2*(*1*) <−>
— ∃ *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*) ↔
(*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*:∀ *x i*. (*Functions I i has-derivative FunctionFrechet I i x*) (*at x*)
shows *FunctionFrechet I id1* (*vec-lambda* (*λi*. *sterm-sem I* (*empty i*) (*fst ν*))) (*vec-lambda*(*λi*. *frechet I* (*empty i*) (*fst ν*) (*snd ν*)))= *0*
⟨*proof*⟩

lemma *constant-deriv-zero*:*is-interp I* ⟹ *directional-derivative I* ($*f id1 empty*) *ν* = *0*
⟨*proof*⟩

theorem *diff-const-axiom-valid*: *valid diff-const-axiom*
⟨*proof*⟩

theorem *diff-var-axiom-valid*: *valid diff-var-axiom*
⟨*proof*⟩

theorem *diff-plus-axiom-valid*: *valid diff-plus-axiom*
⟨*proof*⟩

theorem *diff-times-axiom-valid*: *valid diff-times-axiom*
⟨*proof*⟩

9.4 Proofs for ODE Axioms

lemma *DW-valid*:*valid DWaxiom*
⟨*proof*⟩

lemma *DE-lemma*:
fixes *ab bb*::*′sz simple-state*
and *sol*::*real* ⇒ *′sz simple-state*
and *I*::(*′sf*, *′sc*, *′sz*) *interp*
shows
repd (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)) *vid1* (*dterm-sem I* (*f1 fid1 vid1*) (*mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)))
= *mk-v I* (*OSing vid1* (*f1 fid1 vid1*)) (*ab*, *bb*) (*sol t*)

⟨*proof*⟩

lemma *DE-valid*:*valid DEaxiom*
⟨*proof*⟩

lemma *ODE-zero*:$\bigwedge i$. *Inl i* $\notin$ *BVO ODE* $\Longrightarrow$ *Inr i* $\notin$ *BVO ODE* $\Longrightarrow$ *ODE-sem I ODE* ν \$ *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'*:$\bigwedge x$. $x \in \{0..t\} \Longrightarrow$ (*f has-derivative* (*f' x*)) (*at x within* $\{0..t\}$)
assumes *geq'*:$\bigwedge x$. $x \in \{0..t\} \Longrightarrow f'\ x\ s \geq 0$
assumes *int-s*:$s > 0 \wedge s \leq t$
assumes *t*: $0 < t$
shows $f\ s \geq f\ 0$
⟨*proof*⟩

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

lemma *MVT'-gr*:
fixes *f g* ::*real* $\Rightarrow$ *real*

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

lemma *frech-linear*:
fixes $x\ \vartheta\ \nu\ \nu'\ I$
assumes *good-interp*:*is-interp I*
assumes *free*:*dfree* ϑ
shows $x * \textit{frechet}\ I\ \vartheta\ \nu\ \nu' = \textit{frechet}\ I\ \vartheta\ \nu\ (x *_R \nu')$
⟨*proof*⟩

lemma *rift-in-space-time*:
fixes *sol I ODE* $\psi\ \vartheta$ *t s b*
assumes *good-interp*:*is-interp I*
assumes *free*:*dfree* ϑ
assumes *osafe*:*osafe ODE*
assumes *sol*:(*sol solves-ode* (λ- ν'. *ODE-sem I ODE* ν')) $\{0..t\}$
$\{x.\ \textit{mk-v}\ I\ \textit{ODE}\ (\textit{sol}\ 0,\ b)\ x \in \textit{fml-sem}\ I\ \psi\}$
assumes *FVT*:$\textit{FVT}\ \vartheta \subseteq \textit{semBV}\ I\ \textit{ODE}$
assumes *ivl*:$s \in \{0..t\}$
shows ((λt. *sterm-sem I* ϑ (*fst* (*mk-v I ODE* (*sol 0*, *b*) (*sol t*))))
— This is Frechet derivative, so equivalent to:
— *has-real-derivative frechet I* ϑ (*fst*((*mk-v I ODE* (*sol 0*, *b*) (*sol s*)))) (*snd* (*mk-v I ODE* (*sol 0*, *b*) (*sol s*))))) (*at s within* $\{0..t\}$)
has-derivative ($\lambda t'$. $t' *$ *frechet I* ϑ (*fst*((*mk-v I ODE* (*sol 0*, *b*) (*sol s*)))) (*snd* (*mk-v I ODE* (*sol 0*, *b*) (*sol s*))))) (*at s within* $\{0..t\}$)
⟨*proof*⟩

lemma *dterm-sterm-dfree*:
$\textit{dfree}\ \vartheta \Longrightarrow (\bigwedge \nu\ \nu'.\ \textit{sterm-sem}\ I\ \vartheta\ \nu = \textit{dterm-sem}\ I\ \vartheta\ (\nu, \nu'))$
⟨*proof*⟩
lemma *DIGeq-valid*:*valid DIGeqaxiom*
⟨*proof*⟩

lemma *DIGr-valid*:*valid DIGraxiom*
⟨*proof*⟩

lemma *DG-valid*:*valid DGaxiom*
⟨*proof*⟩
end end
theory *USubst*

imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
Static-Semantics
begin

10 Uniform Substitution Definitions

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

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

Explicit data structure for substitutions.

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

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

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

inductive *TadmitFFO* :: $('d \Rightarrow ('a,\ 'c)\ trm) \Rightarrow ('a + 'd,\ 'c)\ trm \Rightarrow bool$
where
  $TadmitFFO\text{-}Diff: TadmitFFO\ \sigma\ \vartheta \Longrightarrow NTUadmit\ \sigma\ \vartheta\ UNIV \Longrightarrow TadmitFFO\ \sigma\ (Differential\ \vartheta)$
| $TadmitFFO\text{-}Fun1: (\bigwedge i.\ TadmitFFO\ \sigma\ (args\ i)) \Longrightarrow TadmitFFO\ \sigma\ (Function\ (Inl\ f)\ args)$
| $TadmitFFO\text{-}Fun2: (\bigwedge i.\ TadmitFFO\ \sigma\ (args\ i)) \Longrightarrow dfree\ (\sigma\ f) \Longrightarrow TadmitFFO$

σ (*Function* (*Inr f*) *args*)
| *TadmitFFO-Plus*: *TadmitFFO* σ $\vartheta 1$ $\Longrightarrow$ *TadmitFFO* σ $\vartheta 2$ $\Longrightarrow$ *TadmitFFO* σ (*Plus* $\vartheta 1$ $\vartheta 2$)
| *TadmitFFO-Times*: *TadmitFFO* σ $\vartheta 1$ $\Longrightarrow$ *TadmitFFO* σ $\vartheta 2$ $\Longrightarrow$ *TadmitFFO* σ (*Times* $\vartheta 1$ $\vartheta 2$)
| *TadmitFFO-Var*: *TadmitFFO* σ (*Var x*)
| *TadmitFFO-Const*: *TadmitFFO* σ (*Const r*)

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

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

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

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

```
and TadmitFO-Fun-simps[simp]: TadmitFO σ (Function i args)

primrec Tsubst::('a, 'c) trm ⇒ ('a, 'b, 'c) subst ⇒ ('a, 'c) trm
where
  Tsubst (Var x) σ = Var x
| Tsubst (DiffVar x) σ = DiffVar x
| Tsubst (Const r) σ = Const r
| Tsubst (Function f args) σ = (case SFunctions σ f of Some f' ⇒ TsubstFO f' |
None ⇒ Function f) (λ i. Tsubst (args i) σ)
| Tsubst (Plus ϑ1 ϑ2) σ = Plus (Tsubst ϑ1 σ) (Tsubst ϑ2 σ)
| Tsubst (Times ϑ1 ϑ2) σ = Times (Tsubst ϑ1 σ) (Tsubst ϑ2 σ)
| Tsubst (Differential ϑ) σ = Differential (Tsubst ϑ σ)

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

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

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

| FsubstFO (Geq ϑ1 ϑ2) σ = Geq (TsubstFO ϑ1 σ) (TsubstFO ϑ2 σ)
| FsubstFO (Prop p args) σ = Prop p (λi. TsubstFO (args i) σ)
| FsubstFO (Not φ) σ = Not (FsubstFO φ σ)
| FsubstFO (And φ ψ) σ = And (FsubstFO φ σ) (FsubstFO ψ σ)
| FsubstFO (Exists x φ) σ = Exists x (FsubstFO φ σ)
| FsubstFO (Diamond α φ) σ = Diamond (PsubstFO α σ) (FsubstFO φ σ)
| FsubstFO (InContext C φ) σ = InContext C (FsubstFO φ σ)

fun PPsubst::('a, 'b + 'd, 'c) hp ⇒ ('d ⇒ ('a, 'b, 'c) formula) ⇒ ('a, 'b, 'c) hp
and PFsubst::('a, 'b + 'd, 'c) formula ⇒ ('d ⇒ ('a, 'b, 'c) formula) ⇒ ('a, 'b, 'c)
```

formula
where
$PPsubst\ (Pvar\ a)\ \sigma = Pvar\ a$
$|\ PPsubst\ (Assign\ x\ \vartheta)\ \sigma = Assign\ x\ \vartheta$
$|\ PPsubst\ (DiffAssign\ x\ \vartheta)\ \sigma = DiffAssign\ x\ \vartheta$
$|\ PPsubst\ (Test\ \varphi)\ \sigma = Test\ (PFsubst\ \varphi\ \sigma)$
$|\ PPsubst\ (EvolveODE\ ODE\ \varphi)\ \sigma = EvolveODE\ ODE\ (PFsubst\ \varphi\ \sigma)$
$|\ PPsubst\ (Choice\ \alpha\ \beta)\ \sigma = Choice\ (PPsubst\ \alpha\ \sigma)\ (PPsubst\ \beta\ \sigma)$
$|\ PPsubst\ (Sequence\ \alpha\ \beta)\ \sigma = Sequence\ (PPsubst\ \alpha\ \sigma)\ (PPsubst\ \beta\ \sigma)$
$|\ PPsubst\ (Loop\ \alpha)\ \sigma = Loop\ (PPsubst\ \alpha\ \sigma)$

$|\ PFsubst\ (Geq\ \vartheta 1\ \vartheta 2)\ \sigma = (Geq\ \vartheta 1\ \vartheta 2)$
$|\ PFsubst\ (Prop\ p\ args)\ \sigma = Prop\ p\ args$
$|\ PFsubst\ (Not\ \varphi)\ \sigma = Not\ (PFsubst\ \varphi\ \sigma)$
$|\ PFsubst\ (And\ \varphi\ \psi)\ \sigma = And\ (PFsubst\ \varphi\ \sigma)\ (PFsubst\ \psi\ \sigma)$
$|\ PFsubst\ (Exists\ x\ \varphi)\ \sigma = Exists\ x\ (PFsubst\ \varphi\ \sigma)$
$|\ PFsubst\ (Diamond\ \alpha\ \varphi)\ \sigma = Diamond\ (PPsubst\ \alpha\ \sigma)\ (PFsubst\ \varphi\ \sigma)$
$|\ PFsubst\ (InContext\ C\ \varphi)\ \sigma = (case\ C\ of\ Inl\ C' \Rightarrow InContext\ C'\ (PFsubst\ \varphi\ \sigma)$
$|\ Inr\ p' \Rightarrow \sigma\ p')$

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
$Psubst\ (Pvar\ a)\ \sigma = (case\ SPrograms\ \sigma\ a\ of\ Some\ a' \Rightarrow a'\ |\ None \Rightarrow Pvar\ a)$
$|\ Psubst\ (Assign\ x\ \vartheta)\ \sigma = Assign\ x\ (Tsubst\ \vartheta\ \sigma)$
$|\ Psubst\ (DiffAssign\ x\ \vartheta)\ \sigma = DiffAssign\ x\ (Tsubst\ \vartheta\ \sigma)$
$|\ Psubst\ (Test\ \varphi)\ \sigma = Test\ (Fsubst\ \varphi\ \sigma)$
$|\ Psubst\ (EvolveODE\ ODE\ \varphi)\ \sigma = EvolveODE\ (Osubst\ ODE\ \sigma)\ (Fsubst\ \varphi\ \sigma)$
$|\ Psubst\ (Choice\ \alpha\ \beta)\ \sigma = Choice\ (Psubst\ \alpha\ \sigma)\ (Psubst\ \beta\ \sigma)$
$|\ Psubst\ (Sequence\ \alpha\ \beta)\ \sigma = Sequence\ (Psubst\ \alpha\ \sigma)\ (Psubst\ \beta\ \sigma)$
$|\ Psubst\ (Loop\ \alpha)\ \sigma = Loop\ (Psubst\ \alpha\ \sigma)$

$|\ Fsubst\ (Geq\ \vartheta 1\ \vartheta 2)\ \sigma = Geq\ (Tsubst\ \vartheta 1\ \sigma)\ (Tsubst\ \vartheta 2\ \sigma)$
$|\ Fsubst\ (Prop\ p\ args)\ \sigma = (case\ SPredicates\ \sigma\ p\ of\ Some\ p' \Rightarrow FsubstFO\ p'\ (\lambda i.$
$Tsubst\ (args\ i)\ \sigma)\ |\ None \Rightarrow Prop\ p\ (\lambda i.\ Tsubst\ (args\ i)\ \sigma))$
$|\ Fsubst\ (Not\ \varphi)\ \sigma = Not\ (Fsubst\ \varphi\ \sigma)$
$|\ Fsubst\ (And\ \varphi\ \psi)\ \sigma = And\ (Fsubst\ \varphi\ \sigma)\ (Fsubst\ \psi\ \sigma)$
$|\ Fsubst\ (Exists\ x\ \varphi)\ \sigma = Exists\ x\ (Fsubst\ \varphi\ \sigma)$
$|\ Fsubst\ (Diamond\ \alpha\ \varphi)\ \sigma = Diamond\ (Psubst\ \alpha\ \sigma)\ (Fsubst\ \varphi\ \sigma)$
$|\ Fsubst\ (InContext\ C\ \varphi)\ \sigma = (case\ SContexts\ \sigma\ C\ of\ Some\ C' \Rightarrow PFsubst\ C'\ (\lambda$
$\text{-}.\ (Fsubst\ \varphi\ \sigma))\ |\ None \Rightarrow\ InContext\ C\ (Fsubst\ \varphi\ \sigma))$

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

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

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

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

inductive $Tadmit :: ('a, 'b, 'c)\ subst \Rightarrow ('a, 'c)\ trm \Rightarrow bool$
where
$Tadmit\text{-}Diff{:}Tadmit\ \sigma\ \vartheta \Longrightarrow TUadmit\ \sigma\ \vartheta\ UNIV \Longrightarrow Tadmit\ \sigma\ (Differential\ \vartheta)$
| $Tadmit\text{-}Fun1{:}(\bigwedge i.\ Tadmit\ \sigma\ (args\ i)) \Longrightarrow SFunctions\ \sigma\ f = Some\ f' \Longrightarrow TadmitFO\ (\lambda\ i.\ Tsubst\ (args\ i)\ \sigma)\ f' \Longrightarrow Tadmit\ \sigma\ (Function\ f\ args)$
| $Tadmit\text{-}Fun2{:}(\bigwedge i.\ Tadmit\ \sigma\ (args\ i)) \Longrightarrow SFunctions\ \sigma\ f = None \Longrightarrow Tadmit\ \sigma\ (Function\ f\ args)$
| $Tadmit\text{-}Plus{:}Tadmit\ \sigma\ \vartheta 1 \Longrightarrow Tadmit\ \sigma\ \vartheta 2 \Longrightarrow Tadmit\ \sigma\ (Plus\ \vartheta 1\ \vartheta 2)$
| $Tadmit\text{-}Times{:}Tadmit\ \sigma\ \vartheta 1 \Longrightarrow Tadmit\ \sigma\ \vartheta 2 \Longrightarrow Tadmit\ \sigma\ (Times\ \vartheta 1\ \vartheta 2)$
| $Tadmit\text{-}DiffVar{:}Tadmit\ \sigma\ (DiffVar\ x)$
| $Tadmit\text{-}Var{:}Tadmit\ \sigma\ (Var\ x)$
| $Tadmit\text{-}Const{:}Tadmit\ \sigma\ (Const\ r)$

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

inductive $TadmitF :: ('a, 'b, 'c)\ subst \Rightarrow ('a, 'c)\ trm \Rightarrow bool$
where
$TadmitF\text{-}Diff{:}TadmitF\ \sigma\ \vartheta \Longrightarrow TUadmit\ \sigma\ \vartheta\ UNIV \Longrightarrow TadmitF\ \sigma\ (Differential\ \vartheta)$
| $TadmitF\text{-}Fun1{:}(\bigwedge i.\ TadmitF\ \sigma\ (args\ i)) \Longrightarrow SFunctions\ \sigma\ f = Some\ f' \Longrightarrow (\bigwedge i.\ dfree\ (Tsubst\ (args\ i)\ \sigma)) \Longrightarrow TadmitFFO\ (\lambda\ i.\ Tsubst\ (args\ i)\ \sigma)\ f' \Longrightarrow TadmitF\ \sigma\ (Function\ f\ args)$
| $TadmitF\text{-}Fun2{:}(\bigwedge i.\ TadmitF\ \sigma\ (args\ i)) \Longrightarrow SFunctions\ \sigma\ f = None \Longrightarrow TadmitF\ \sigma\ (Function\ f\ args)$
| $TadmitF\text{-}Plus{:}TadmitF\ \sigma\ \vartheta 1 \Longrightarrow TadmitF\ \sigma\ \vartheta 2 \Longrightarrow TadmitF\ \sigma\ (Plus\ \vartheta 1\ \vartheta 2)$
| $TadmitF\text{-}Times{:}TadmitF\ \sigma\ \vartheta 1 \Longrightarrow TadmitF\ \sigma\ \vartheta 2 \Longrightarrow TadmitF\ \sigma\ (Times\ \vartheta 1$

ϑ2)
| *TadmitF-DiffVar*:*TadmitF σ* (*DiffVar x*)
| *TadmitF-Var*:*TadmitF σ* (*Var x*)
| *TadmitF-Const*:*TadmitF σ* (*Const r*)

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

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

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

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

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

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

inductive *OadmitFO* :: ('*d* ⇒ ('*a*, '*c*) *trm*) ⇒ ('*a* + '*d*, '*c*) *ODE* ⇒ ('*c* + '*c*) *set* ⇒ *bool*
where
OadmitFO-OVar:*OUadmitFO σ* (*OVar c*) *U* ⟹ *OadmitFO σ* (*OVar c*) *U*
| *OadmitFO-OSing*:*OUadmitFO σ* (*OSing x ϑ*) *U* ⟹ *TadmitFFO σ ϑ* ⟹ *OadmitFO σ* (*OSing x ϑ*) *U*
| *OadmitFO-OProd*:*OadmitFO σ ODE1 U* ⟹ *OadmitFO σ ODE2 U* ⟹ *OadmitFO σ* (*OProd ODE1 ODE2*) *U*

inductive-simps

OadmitFO-OVar-simps[*simp*]: *OadmitFO σ* (*OVar a*) *U*
and *OadmitFO-OProd-simps*[*simp*]: *OadmitFO σ* (*OProd ODE1 ODE2*) *U*
and *OadmitFO-OSing-simps*[*simp*]: *OadmitFO σ* (*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 σ ϑ 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 σ ϑ U* $\longleftrightarrow$ $((\bigcup\ i \in \{i.\ Inl\ (Inr\ i) \in SIGP\ \vartheta\}.\ FVT\ (\sigma\ i)) \cap U) = \{\}$

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

| *NPadmit-Assign*:*TadmitFO σ ϑ* $\Longrightarrow$ *NPadmit σ* (*Assign x ϑ*)
| *NPadmit-DiffAssign*:*TadmitFO σ ϑ* $\Longrightarrow$ *NPadmit σ* (*DiffAssign x ϑ*)
| *NPadmit-Test*:*NFadmit σ φ* $\Longrightarrow$ *NPadmit σ* (*Test φ*)

| *NFadmit-Geq*:*TadmitFO σ ϑ1* $\Longrightarrow$ *TadmitFO σ ϑ2* $\Longrightarrow$ *NFadmit σ* (*Geq ϑ1 ϑ2*)
| *NFadmit-Prop*:($\bigwedge$*i. TadmitFO σ* (*args i*)) $\Longrightarrow$ *NFadmit σ* (*Prop f args*)
| *NFadmit-Not*:*NFadmit σ φ* $\Longrightarrow$ *NFadmit σ* (*Not φ*)
| *NFadmit-And*:*NFadmit σ φ* $\Longrightarrow$ *NFadmit σ ψ* $\Longrightarrow$ *NFadmit σ* (*And φ ψ*)
| *NFadmit-Exists*:*NFadmit σ φ* $\Longrightarrow$ *FUadmitFO σ φ* {*Inl x*} $\Longrightarrow$ *NFadmit σ* (*Exists x φ*)
| *NFadmit-Diamond*:*NFadmit σ φ* $\Longrightarrow$ *NPadmit σ a* $\Longrightarrow$ *FUadmitFO σ φ* (*BVP* (*PsubstFO a σ*)) $\Longrightarrow$ *hpsafe* (*PsubstFO a σ*) $\Longrightarrow$ *NFadmit σ* (*Diamond a φ*)
| *NFadmit-Context*:*NFadmit σ φ* $\Longrightarrow$ *FUadmitFO σ φ UNIV* $\Longrightarrow$ *NFadmit σ* (*InContext C φ*)

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

and *NPadmit-DiffAssign-simps*[*simp*]: *NPadmit* σ (*DiffAssign x e*)
and *NPadmit-Test-simps*[*simp*]: *NPadmit* σ (*? p*)

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

definition *PFUadmit* :: ($'d \Rightarrow$ ($'a$, $'b$, $'c$) *formula*) $\Rightarrow$ ($'a$, $'b + 'd$, $'c$) *formula* $\Rightarrow$ ($'c + 'c$) *set* $\Rightarrow$ *bool*
where *PFUadmit* σ ϑ *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* σ ϑ *U* $\longleftrightarrow$ (($\bigcup$ *i. FVF* (σ *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-Pvar:*PPadmit* σ (*Pvar a*)
| *PPadmit-Sequence*:*PPadmit* σ *a* $\Longrightarrow$ *PPadmit* σ *b* $\Longrightarrow$ *PPUadmit* σ *b* (*BVP* (*PPsubst a* σ))$\Longrightarrow$ *hpsafe* (*PPsubst a* σ) $\Longrightarrow$ *PPadmit* σ (*Sequence a b*)
| *PPadmit-Loop*:*PPadmit* σ *a* $\Longrightarrow$ *PPUadmit* σ *a* (*BVP* (*PPsubst a* σ)) $\Longrightarrow$ *hpsafe* (*PPsubst a* σ) $\Longrightarrow$ *PPadmit* σ (*Loop a*)
| *PPadmit-ODE*:*PFadmit* σ φ $\Longrightarrow$ *PFUadmit* σ φ (*BVO ODE*) $\Longrightarrow$ *PPadmit* σ (*EvolveODE ODE* φ)
| *PPadmit-Choice*:*PPadmit* σ *a* $\Longrightarrow$ *PPadmit* σ *b* $\Longrightarrow$ *PPadmit* σ (*Choice a b*)

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

| *PFadmit-Geq*:*PFadmit* σ (*Geq* $\vartheta 1$ $\vartheta 2$)
| *PFadmit-Prop*:*PFadmit* σ (*Prop f args*)
| *PFadmit-Not*:*PFadmit* σ φ $\Longrightarrow$ *PFadmit* σ (*Not* φ)
| *PFadmit-And*:*PFadmit* σ φ $\Longrightarrow$ *PFadmit* σ ψ $\Longrightarrow$ *PFadmit* σ (*And* φ ψ)
| *PFadmit-Exists*:*PFadmit* σ φ $\Longrightarrow$ *PFUadmit* σ φ {*Inl x*} $\Longrightarrow$ *PFadmit* σ (*Exists x* φ)
| *PFadmit-Diamond*:*PFadmit* σ φ $\Longrightarrow$ *PPadmit* σ *a* $\Longrightarrow$ *PFUadmit* σ φ (*BVP* (*PPsubst a* σ)) $\Longrightarrow$ *PFadmit* σ (*Diamond a* φ)
| *PFadmit-Context*:*PFadmit* σ φ $\Longrightarrow$ *PFUadmit* σ φ *UNIV* $\Longrightarrow$ *PFadmit* σ (*InContext C* φ)

inductive-simps
PPadmit-Pvar-simps[*simp*]: *PPadmit* σ (*Pvar a*)
and *PPadmit-Sequence-simps*[*simp*]: *PPadmit* σ (*a* ;; *b*)

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

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

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

| *Fadmit-Geq*:*Tadmit* σ $\vartheta 1$ ⟹ *Tadmit* σ $\vartheta 2$ ⟹ *Fadmit* σ (*Geq* $\vartheta 1$ $\vartheta 2$)
| *Fadmit-Prop1*:($\bigwedge$*i*. *Tadmit* σ (*args i*)) ⟹ *SPredicates* σ *p* = *Some p'* ⟹ *NFadmit* (λ *i*. *Tsubst* (*args i*) σ) *p'* ⟹ ($\bigwedge$*i*. *dsafe* (*Tsubst* (*args i*) σ))⟹ *Fadmit* σ (*Prop p args*)
| *Fadmit-Prop2*:($\bigwedge$*i*. *Tadmit* σ (*args i*)) ⟹ *SPredicates* σ *p* = *None* ⟹ *Fadmit* σ (*Prop p args*)
| *Fadmit-Not*:*Fadmit* σ φ ⟹ *Fadmit* σ (*Not* φ)
| *Fadmit-And*:*Fadmit* σ φ ⟹ *Fadmit* σ ψ ⟹ *Fadmit* σ (*And* φ ψ)
| *Fadmit-Exists*:*Fadmit* σ φ ⟹ *FUadmit* σ φ {*Inl x*} ⟹ *Fadmit* σ (*Exists x* φ)
| *Fadmit-Diamond*:*Fadmit* σ φ ⟹ *Padmit* σ *a* ⟹ *FUadmit* σ φ (*BVP* (*Psubst a* σ)) ⟹ *hpsafe* (*Psubst a* σ) ⟹ *Fadmit* σ (*Diamond a* φ)
| *Fadmit-Context1*:*Fadmit* σ φ ⟹ *FUadmit* σ φ *UNIV* ⟹ *SContexts* σ *C* = *Some C'* ⟹ *PFadmit* (λ -. *Fsubst* φ σ) *C'* ⟹ *fsafe*(*Fsubst* φ σ) ⟹ *Fadmit* σ (*InContext C* φ)
| *Fadmit-Context2*:*Fadmit* σ φ ⟹ *FUadmit* σ φ *UNIV* ⟹ *SContexts* σ *C* = *None* ⟹ *Fadmit* σ (*InContext C* φ)

inductive-simps
Padmit-Pvar-simps[*simp*]: *Padmit* σ (*Pvar a*)

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

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

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

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

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

ODEBV I ode)
⦈

lemma *dsem-to-ssem*:*dfree* $\vartheta \Longrightarrow$ *dterm-sem I* ϑ ν = *sterm-sem I* ϑ (*fst* ν)
⟨*proof*⟩

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

lemma *adjoint-free*:
assumes *sfree*:(⋀*i f*′. *SFunctions* σ *i* = *Some f*′ ⟹ *dfree f*′)
shows *adjoint I* σ ν =
⦇*Functions* = (λ*f*. *case SFunctions* σ *f of Some f*′ ⇒ (λ*R*. *sterm-sem* (*extendf I R*) *f*′ (*fst* ν)) | *None* ⇒ *Functions I f*),
Predicates = (λ*p*. *case SPredicates* σ *p of Some p*′ ⇒ (λ*R*. ν ∈ *fml-sem* (*extendf I R*) *p*′) | *None* ⇒ *Predicates I p*),
Contexts = (λ*c*. *case SContexts* σ *c of Some c*′ ⇒ (λ*R*. *fml-sem* (*extendc I R*) *c*′) | *None* ⇒ *Contexts I c*),
Programs = (λ*a*. *case SPrograms* σ *a of Some a*′ ⇒ *prog-sem I a*′ | *None* ⇒ *Programs I a*),
ODEs = (λ*ode*. *case SODEs* σ *ode of Some ode*′ ⇒ *ODE-sem I ode*′ | *None* ⇒ *ODEs I ode*),
ODEBV = (λ*ode*. *case SODEs* σ *ode of Some ode*′ ⇒ *ODE-vars I ode*′ | *None* ⇒ *ODEBV I ode*)⦈
⟨*proof*⟩

lemma *adjointFO-free*:(⋀*i*. *dfree* (σ *i*)) ⟹ (*adjointFO I* σ ν =
⦇*Functions* = (λ*f*. *case f of Inl f*′ ⇒ *Functions I f*′ | *Inr f*′ ⇒ (λ-. *sterm-sem I* (σ *f*′) (*fst* ν))),
Predicates = *Predicates I*,
Contexts = *Contexts I*,
Programs = *Programs I*,
ODEs = *ODEs I*,
ODEBV = *ODEBV I*⦈)
⟨*proof*⟩

definition *PFadjoint*::('*sf*, '*sc*, '*sz*) *interp* ⇒ ('*d*::*finite* ⇒ ('*sf*, '*sc*, '*sz*) *formula*) ⇒ ('*sf*, '*sc* + '*d*, '*sz*) *interp*
where *PFadjoint I* σ =
⦇*Functions* = *Functions I*,

Predicates = *Predicates I*,
Contexts = $(\lambda f.$ *case f of Inl f′* ⇒ *Contexts I f′* | *Inr f′* ⇒ (λ-. *fml-sem I* (σ *f′*))),
Programs = *Programs I*,
ODEs = *ODEs I*,
ODEBV = *ODEBV I*⦈

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

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

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

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

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

11 Soundness proof for uniform substitution rule

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

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

lemma *extendf-deriv*:
fixes f'::('sf + 'sz,'sz) *trm* **and** I::('sf,'sc,'sz) *interp*
assumes *free*:*dfree* f'
assumes *good-interp*:*is-interp I*
shows $\exists f''. \forall x.$ ((λR. *dterm-sem* (*extendf I R*) f' ν) *has-derivative* (*extendf-deriv* *I i-f* f' ν x)) (*at x*)
⟨*proof*⟩

lemma *adjoint-safe*:
assumes *good-interp*:*is-interp I*
assumes *good-subst*:($\bigwedge i\ f'$. *SFunctions* σ i = *Some* f' $\Longrightarrow$ *dfree* f')
shows *is-interp* (*adjoint I* σ ν)
⟨*proof*⟩

lemma *adjointFO-safe*:
assumes *good-interp*:*is-interp I*
assumes *good-subst*:($\bigwedge i$. *dsafe* (σ i))
shows *is-interp* (*adjointFO I* σ ν)
⟨*proof*⟩

11.2 Lemmas about adjoint interpretations

lemma *adjoint-consequence*:($\bigwedge f\ f'$. *SFunctions* σ f = *Some* f' $\Longrightarrow$ *dsafe* f') $\Longrightarrow$ ($\bigwedge f\ f'$. *SPredicates* σ f = *Some* f' $\Longrightarrow$ *fsafe* f') $\Longrightarrow$ *Vagree* ν ω (*FVS* σ) $\Longrightarrow$ *adjoint I* σ ν = *adjoint I* σ ω
⟨*proof*⟩

lemma *SIGT-plus1*:*Vagree* ν ω ($\bigcup i\in SIGT$ (*Plus t1 t2*). *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None* $\Rightarrow$ {})
$\Longrightarrow$ *Vagree* ν ω ($\bigcup i\in SIGT\ t1$. *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None* $\Rightarrow$ {})
⟨*proof*⟩

lemma *SIGT-plus2*:*Vagree* ν ω ($\bigcup i\in SIGT$ (*Plus t1 t2*). *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None* $\Rightarrow$ {})
$\Longrightarrow$ *Vagree* ν ω ($\bigcup i\in SIGT\ t2$. *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None* $\Rightarrow$ {})
⟨*proof*⟩

lemma *SIGT-times1*:*Vagree* ν ω ($\bigcup i\in SIGT$ (*Times t1 t2*). *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None* $\Rightarrow$ {})
$\Longrightarrow$ *Vagree* ν ω ($\bigcup i\in SIGT\ t1$. *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None* $\Rightarrow$ {})
⟨*proof*⟩

lemma *SIGT-times2*:*Vagree* ν ω ($\bigcup i\in SIGT$ (*Times t1 t2*). *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None* $\Rightarrow$ {})
$\Longrightarrow$ *Vagree* ν ω ($\bigcup i\in SIGT\ t2$. *case SFunctions* σ *i of Some* $x \Rightarrow FVT\ x$ | *None*

$\Rightarrow$ {})
⟨proof⟩

lemma *uadmit-sterm-adjoint′*:
assumes *dsafe*:$\bigwedge f\, f'.\ SFunctions\ \sigma\ f = Some\ f' \Longrightarrow dsafe\ f'$
assumes *fsafe*:$\bigwedge f\, f'.\ SPredicates\ \sigma\ f = Some\ f' \Longrightarrow fsafe\ f'$
shows $Vagree\ \nu\ \omega\ (\bigcup i{\in}SIGT\ \vartheta.\ case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x\ |\ None \Rightarrow \{\}) \Longrightarrow sterm\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ \vartheta = sterm\text{-}sem\ (adjoint\ I\ \sigma\ \omega)\ \vartheta$
⟨proof⟩
lemma *uadmit-sterm-adjoint*:
assumes *TUA*:$TUadmit\ \sigma\ \vartheta\ U$
assumes *VA*:$Vagree\ \nu\ \omega\ (-U)$
assumes *dsafe*:$\bigwedge f\, f'.\ SFunctions\ \sigma\ f = Some\ f' \Longrightarrow dsafe\ f'$
assumes *fsafe*:$\bigwedge f\, f'.\ SPredicates\ \sigma\ f = Some\ f' \Longrightarrow fsafe\ f'$
shows $sterm\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ \vartheta = sterm\text{-}sem\ (adjoint\ I\ \sigma\ \omega)\ \vartheta$
⟨proof⟩

lemma *uadmit-sterm-ntadjoint′*:
assumes *dsafe*:$\bigwedge i.\ dsafe\ (\sigma\ i)$
shows $Vagree\ \nu\ \omega\ ((\bigcup\ i{\in}\{i.\ Inr\ i \in SIGT\ \vartheta\}.\ FVT\ (\sigma\ i))) \Longrightarrow sterm\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ \vartheta = sterm\text{-}sem\ (adjointFO\ I\ \sigma\ \omega)\ \vartheta$
⟨proof⟩

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 $sterm\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ \vartheta = sterm\text{-}sem\ (adjointFO\ I\ \sigma\ \omega)\ \vartheta$
⟨proof⟩

lemma *uadmit-dterm-adjoint′*:
assumes *dfree*:$\bigwedge f\, f'.\ SFunctions\ \sigma\ f = Some\ f' \Longrightarrow dfree\ f'$
assumes *fsafe*:$\bigwedge f\, f'.\ SPredicates\ \sigma\ f = Some\ f' \Longrightarrow fsafe\ f'$
assumes *good-interp*:*is-interp I*
shows $\bigwedge \nu\ \omega.\ Vagree\ \nu\ \omega\ (\bigcup i{\in}SIGT\ \vartheta.\ case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x\ |\ None \Rightarrow \{\}) \Longrightarrow dsafe\ \vartheta \Longrightarrow dterm\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ \vartheta = dterm\text{-}sem\ (adjoint\ I\ \sigma\ \omega)\ \vartheta$
⟨proof⟩

lemma *uadmit-dterm-adjoint*:
assumes *TUA*:$TUadmit\ \sigma\ \vartheta\ U$
assumes *VA*:$Vagree\ \nu\ \omega\ (-U)$
assumes *dfree*:$\bigwedge f\, f'.\ SFunctions\ \sigma\ f = Some\ f' \Longrightarrow dfree\ f'$
assumes *fsafe*:$\bigwedge f\, f'.\ SPredicates\ \sigma\ f = Some\ f' \Longrightarrow fsafe\ f'$
assumes *dsafe*:$dsafe\ \vartheta$
assumes *good-interp*:*is-interp I*
shows $dterm\text{-}sem\ (adjoint\ I\ \sigma\ \nu)\ \vartheta = dterm\text{-}sem\ (adjoint\ I\ \sigma\ \omega)\ \vartheta$
⟨proof⟩

lemma *uadmit-dterm-ntadjoint′*:
assumes *dfree*:$\bigwedge i.$ *dsafe* $(\sigma\ i)$
assumes *good-interp*:*is-interp I*
shows $\bigwedge\nu\ \omega.$ *Vagree* $\nu\ \omega$ $(\bigcup\ i\in\{i.\ Inr\ i \in SIGT\ \vartheta\}.\ FVT\ (\sigma\ i)) \Longrightarrow dsafe\ \vartheta$ $\Longrightarrow$ *dterm-sem* $(adjointFO\ I\ \sigma\ \nu)\ \vartheta =$ *dterm-sem* $(adjointFO\ I\ \sigma\ \omega)\ \vartheta$
⟨*proof*⟩

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* ϑ
assumes *good-interp*:*is-interp I*
shows *dterm-sem* $(adjointFO\ I\ \sigma\ \nu)\ \vartheta =$ *dterm-sem* $(adjointFO\ I\ \sigma\ \omega)\ \vartheta$
⟨*proof*⟩

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* σ
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* ϑ
shows *dterm-sem* $(adjoint\ I\ \sigma\ \nu)\ \vartheta =$ *dterm-sem* $(adjoint\ I\ \sigma\ \omega)\ \vartheta$
⟨*proof*⟩

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

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

lemma *adj-sub-geq2-fact*:$\bigwedge i\ j\ x1\ x2.\ i{\in}SIGT\ x2 \Longrightarrow j \in (case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \Longrightarrow Inl\ i \in(\{Inl\ x \mid x.\ x \in dom\ (SFunctions\ \sigma)\} \cup \{Inr\ (Inl\ x) \mid x.\ x \in dom\ (SContexts\ \sigma)\} \cup \{Inr\ (Inr\ x) \mid x.\ x \in dom\ (SPredicates\ \sigma)\} \cup$
$\{Inr\ (Inr\ x) \mid x.\ x \in dom\ (SPrograms\ \sigma)\}) \cap$
$\{Inl\ x \mid x.\ x \in SIGT\ x1 \lor x \in SIGT\ x2\}$
⟨*proof*⟩
lemma *adj-sub-prop-fact*:$\bigwedge i\ j\ x1\ x2\ k.\ i{\in}SIGT\ (x2\ k) \Longrightarrow j \in (case\ SFunctions\ \sigma\ i\ of\ Some\ x \Rightarrow FVT\ x \mid None \Rightarrow \{\}) \Longrightarrow Inl\ i \in(\{Inl\ x \mid x.\ x \in dom\ (SFunctions\ \sigma)\} \cup \{Inr\ (Inl\ x) \mid x.\ x \in dom\ (SContexts\ \sigma)\} \cup \{Inr\ (Inr\ x) \mid x.\ x \in dom\ (SPredicates\ \sigma)\} \cup$
$\{Inr\ (Inr\ x) \mid x.\ x \in dom\ (SPrograms\ \sigma)\}) \cap$
$insert\ (Inr\ (Inr\ x1))\ \{Inl\ x \mid x.\ \exists xa.\ x \in SIGT\ (x2\ xa)\}$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *uadmit-ode-adjoint'*:
  fixes σ I
  assumes *ssafe*:*ssafe* σ
  assumes *good-interp*:*is-interp* I
  shows$\bigwedge\nu\ \omega.$ *Vagree* $\nu\ \omega$ $(\bigcup i{\in}\{i\ |i.\ Inl\ i \in SIGO\ ODE\}.\ case\ SFunctions\ \sigma\ i$ $of\ None \Rightarrow \{\}\ |\ Some\ x \Rightarrow FVT\ x) \Longrightarrow osafe\ ODE \Longrightarrow ODE\text{-}sem\ (adjoint\ I\ \sigma\ \nu)$ $ODE = ODE\text{-}sem\ (adjoint\ I\ \sigma\ \omega)\ ODE$
⟨*proof*⟩

lemma *uadmit-ode-ntadjoint'*:
  fixes σ I
  assumes *ssafe*:$\bigwedge i.\ dsafe\ (\sigma\ i)$
  assumes *good-interp*:*is-interp* I
  shows$\bigwedge\nu\ \omega.$ *Vagree* $\nu\ \omega$ $(\bigcup y{\in}\{y.\ Inl\ (Inr\ y) \in SIGO\ ODE\}.\ FVT\ (\sigma\ y)) \Longrightarrow$ $osafe\ ODE \Longrightarrow ODE\text{-}sem\ (adjointFO\ I\ \sigma\ \nu)\ ODE = ODE\text{-}sem\ (adjointFO\ I\ \sigma$ $\omega)\ ODE$
⟨*proof*⟩

lemma *adjoint-ode-vars*:
  shows $ODE\text{-}vars\ (local.adjoint\ I\ \sigma\ \nu)\ ODE = ODE\text{-}vars\ (local.adjoint\ I\ \sigma\ \omega)$ ODE
  ⟨*proof*⟩

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

lemma *adjointFO-ode-vars*:
  shows $ODE\text{-}vars\ (adjointFO\ I\ \sigma\ \nu)\ ODE = ODE\text{-}vars\ (adjointFO\ I\ \sigma\ \omega)\ ODE$
  ⟨*proof*⟩

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

lemma *uadmit-prog-fml-adjoint'*:
  fixes σ I
  assumes *ssafe*:*ssafe* σ
  assumes *good-interp*:*is-interp* I

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

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

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

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

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

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

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

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

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

lemma *uadmit-prog-fml-ntadjoint′*:

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

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

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

11.3 Substitution theorems for terms

lemma *nsubst-sterm*:
fixes I::(*'sf*, *'sc*, *'sz*) *interp*
fixes ν::*'sz state*
shows *TadmitFFO* $\sigma\ \vartheta \Longrightarrow (\bigwedge i.$ *dsafe* $(\sigma\ i)) \Longrightarrow$ *sterm-sem* I (*TsubstFO* $\vartheta\ \sigma$) (*fst* ν) = *sterm-sem* (*adjointFO* $I\ \sigma\ \nu$) ϑ (*fst* ν)
⟨*proof*⟩

lemma *nsubst-sterm'*:
fixes I::(*'sf*, *'sc*, *'sz*) *interp*
fixes a b::*'sz simple-state*
shows *TadmitFFO* $\sigma\ \vartheta \Longrightarrow (\bigwedge i.$ *dsafe* $(\sigma\ i)) \Longrightarrow$ *sterm-sem* I (*TsubstFO* $\vartheta\ \sigma$) a = *sterm-sem* (*adjointFO* $I\ \sigma\ (a,b)$) $\vartheta\ a$
⟨*proof*⟩

lemma *ntsubst-preserves-free*:
dfree $\vartheta \Longrightarrow (\bigwedge i.$ *dfree* $(\sigma\ i)) \Longrightarrow$ *dfree*(*TsubstFO* $\vartheta\ \sigma$)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *subst-sterm*:
fixes I::(′*sf*, ′*sc*, ′*sz*) *interp*
fixes ν::′*sz state*
shows
TadmitF σ ϑ $\Longrightarrow$
$(\bigwedge i\, f'.$ *SFunctions* σ i = *Some* $f' \Longrightarrow$ *dfree* $f') \Longrightarrow$
sterm-sem I (*Tsubst* ϑ σ) (*fst* ν) = *sterm-sem* (*adjoint* I σ ν) ϑ (*fst* ν)
⟨*proof*⟩

lemma *nsubst-dterm*′:
fixes I::(′*sf*, ′*sc*, ′*sz*) *interp*
fixes ν::′*sz state*
assumes *good-interp*:*is-interp* I
shows *TadmitFO* σ ϑ $\Longrightarrow$ *dfree* ϑ $\Longrightarrow$ $(\bigwedge i.$ *dsafe* $(\sigma\ i)) \Longrightarrow$ *dterm-sem* I (*TsubstFO* ϑ σ) ν = *dterm-sem* (*adjointFO* I σ ν) ϑ ν
⟨*proof*⟩

lemma *ntsubst-free-to-safe*:
dfree ϑ $\Longrightarrow$ $(\bigwedge i.$ *dsafe* $(\sigma\ i)) \Longrightarrow$ *dsafe* (*TsubstFO* ϑ σ)
⟨*proof*⟩

lemma *ntsubst-preserves-safe*:
dsafe ϑ $\Longrightarrow$ $(\bigwedge i.$ *dfree* $(\sigma\ i)) \Longrightarrow$ *dsafe* (*TsubstFO* ϑ σ)
⟨*proof*⟩

lemma *tsubst-preserves-safe*:
dsafe ϑ $\Longrightarrow$ $(\bigwedge i\, f'.$ *SFunctions* σ i = *Some* $f' \Longrightarrow$ *dfree* $f') \Longrightarrow$ *dsafe*(*Tsubst* ϑ σ)
⟨*proof*⟩

lemma *subst-dterm*:
fixes I::(′*sf*, ′*sc*, ′*sz*) *interp*
assumes *good-interp*:*is-interp* I
shows
Tadmit σ ϑ $\Longrightarrow$
dsafe ϑ $\Longrightarrow$
$(\bigwedge i\, f'.$ *SFunctions* σ i = *Some* $f' \Longrightarrow$ *dfree* $f') \Longrightarrow$
$(\bigwedge f\, f'.$ *SPredicates* σ f = *Some* $f' \Longrightarrow$ *fsafe* $f') \Longrightarrow$
$(\bigwedge \nu.$ *dterm-sem* I (*Tsubst* ϑ σ) ν = *dterm-sem* (*adjoint* I σ ν) ϑ $\nu)$
⟨*proof*⟩

11.4 Substitution theorems for ODEs

lemma *osubst-preserves-safe*:
assumes *ssafe*:*ssafe* σ
shows (*osafe ODE* $\Longrightarrow$ *Oadmit* σ *ODE U* $\Longrightarrow$ *osafe* (*Osubst ODE* σ))
⟨*proof*⟩

lemma *nosubst-preserves-safe*:
 assumes *sfree*:$\bigwedge i.\ dfree\ (\sigma\ i)$
 fixes α ::($'a$ + $'d$, $'b$, $'c$) *hp* **and** φ ::($'a$ + $'d$, $'b$, $'c$) *formula*
 shows (*osafe ODE* $\Longrightarrow$ *OUadmitFO* σ *ODE U* $\Longrightarrow$ *osafe* (*OsubstFO ODE* σ))
⟨*proof*⟩

lemma *nsubst-dterm*:
 fixes *I*::($'sf$, $'sc$, $'sz$) *interp*
 fixes ν::$'sz$ *state*
 fixes ν'::$'sz$ *state*
 assumes *good-interp*:*is-interp I*
 shows *TadmitFO* σ ϑ $\Longrightarrow$ *dsafe* ϑ $\Longrightarrow$ ($\bigwedge i.$ *dsafe* (σ *i*)) $\Longrightarrow$ *dterm-sem I* (*TsubstFO* ϑ σ) ν = *dterm-sem* (*adjointFO I* σ ν) ϑ ν
⟨*proof*⟩

lemma *nsubst-ode*:
 fixes *I*::($'sf$, $'sc$, $'sz$) *interp*
 fixes ν::$'sz$ *state*
 fixes ν'::$'sz$ *state*
 assumes *good-interp*:*is-interp I*
 shows *osafe ODE* $\Longrightarrow$ *OadmitFO* σ *ODE U* $\Longrightarrow$ ($\bigwedge i.$ *dsafe* (σ *i*)) $\Longrightarrow$ *ODE-sem I* (*OsubstFO ODE* σ) (*fst* ν)= *ODE-sem* (*adjointFO I* σ ν) *ODE* (*fst* ν)
⟨*proof*⟩

lemma *osubst-preserves-BVO*:
 shows *BVO* (*OsubstFO ODE* σ) = *BVO ODE*
⟨*proof*⟩

lemma *osubst-preserves-ODE-vars*:
 shows *ODE-vars I* (*OsubstFO ODE* σ) = *ODE-vars* (*adjointFO I* σ ν) *ODE*
⟨*proof*⟩

lemma *osubst-preserves-semBV*:
 shows *semBV I* (*OsubstFO ODE* σ) = *semBV* (*adjointFO I* σ ν) *ODE*
⟨*proof*⟩

lemma *nsubst-mkv*:
 fixes *I*::($'sf$, $'sc$, $'sz$) *interp*
 fixes ν::$'sz$ *state*
 fixes ν'::$'sz$ *state*
 assumes *good-interp*:*is-interp I*
 assumes *NOU*:*OadmitFO* σ *ODE U*
 assumes *osafe*:*osafe ODE*
 assumes *frees*:($\bigwedge i.$ *dsafe* (σ *i*))
 shows (*mk-v I* (*OsubstFO ODE* σ) ν (*fst* ν'))
  = (*mk-v* (*adjointFO I* σ ν') *ODE* ν (*fst* ν'))
 ⟨*proof*⟩

lemma *ODE-unbound-zero*:

fixes *i*
shows *Inl i* $\notin$ *BVO ODE* $\Longrightarrow$ *ODE-sem I ODE x* $ *i* = *0*
⟨*proof*⟩

lemma *ODE-bound-effect*:
fixes *s t sol ODE X b*
assumes *s*:*s* $\in$ {*0*..*t*}
assumes *sol*:(*sol solves-ode* (λ-. *ODE-sem I ODE*)) {*0*..*t*} *X*
shows *Vagree* (*sol 0*,*b*) (*sol s*, *b*) (−(*BVO ODE*))
⟨*proof*⟩

lemma *NO-sub*:*OadmitFO* σ *ODE A* $\Longrightarrow$ *B* $\subseteq$ *A* $\Longrightarrow$ *OadmitFO* σ *ODE B*
⟨*proof*⟩

lemma *NO-to-NOU*:*OadmitFO* σ *ODE S* $\Longrightarrow$ *OUadmitFO* σ *ODE S*
⟨*proof*⟩

11.5 Substitution theorems for formulas and programs

lemma *nsubst-hp-fml*:
fixes *I*::(′*sf*, ′*sc*, ′*sz*) *interp*
assumes *good-interp*:*is-interp I*
shows (*NPadmit* σ α ⟶ (*hpsafe* α ⟶ (∀ *i*. *dsafe* (σ *i*)) ⟶ (∀ ν ω. ((ν, ω) ∈ *prog-sem I* (*PsubstFO* α σ)) = ((ν, ω) ∈ *prog-sem* (*adjointFO I* σ ν) α)))) ∧
(*NFadmit* σ φ ⟶ (*fsafe* φ ⟶ (∀ *i*. *dsafe* (σ *i*)) ⟶ (∀ ν. (ν ∈ *fml-sem I* (*FsubstFO* φ σ)) = (ν ∈ *fml-sem* (*adjointFO I* σ ν) φ))))
⟨*proof*⟩

lemma *nsubst-fml*:
fixes *I*::(′*sf*, ′*sc*, ′*sz*) *interp*
fixes ν::′*sz state*
assumes *good-interp*:*is-interp I*
assumes *NFA*:*NFadmit* σ φ
assumes *fsafe*:*fsafe* φ
assumes *frees*:(∀ *i*. *dsafe* (σ *i*))
shows (ν ∈ *fml-sem I* (*FsubstFO* φ σ)) = (ν ∈ *fml-sem* (*adjointFO I* σ ν) φ)
⟨*proof*⟩

lemma *nsubst-hp*:
fixes *I*::(′*sf*, ′*sc*, ′*sz*) *interp*
fixes ν::′*sz state*
assumes *good-interp*:*is-interp I*
assumes *NPA*:*NPadmit* σ α
assumes *hpsafe*:*hpsafe* α
assumes *frees*:⋀*i*. *dsafe* (σ *i*)
shows ((ν, ω) ∈ *prog-sem I* (*PsubstFO* α σ)) = ((ν, ω) ∈ *prog-sem* (*adjointFO I* σ ν) α)
⟨*proof*⟩

lemma *psubst-sterm*:
  fixes I::('sf, 'sc, 'sz) *interp*
  assumes *good-interp*:*is-interp I*
  shows (*sterm-sem I* ϑ = *sterm-sem* (*PFadjoint I* σ) ϑ)
⟨*proof*⟩

lemma *psubst-dterm*:
  fixes I::('sf, 'sc, 'sz) *interp*
  assumes *good-interp*:*is-interp I*
  shows (*dsafe* $\vartheta \Longrightarrow$ *dterm-sem I* ϑ = *dterm-sem* (*PFadjoint I* σ) ϑ)
⟨*proof*⟩

lemma *psubst-ode*:
assumes *good-interp*:*is-interp I*
shows *ODE-sem I ODE* = *ODE-sem* (*PFadjoint I* σ) *ODE*
⟨*proof*⟩

lemma *psubst-fml*:
fixes I::('sf, 'sc, 'sz) *interp*
assumes *good-interp*:*is-interp I*
shows (*PPadmit* σ α $\longrightarrow$ *hpsafe* $\alpha \longrightarrow$ ($\forall i$. *fsafe* (σ i)) $\longrightarrow$ ($\forall$ ν ω. (ν,ω) $\in$ *prog-sem I* (*PPsubst* α σ) = ((ν,ω) $\in$ *prog-sem* (*PFadjoint I* σ) α))) $\wedge$
  (*PFadmit* σ $\varphi \longrightarrow$ *fsafe* $\varphi \longrightarrow$ ($\forall i$. *fsafe* (σ i)) $\longrightarrow$ ($\forall$ ν. $\nu \in$ *fml-sem I* (*PFsubst* φ σ) = ($\nu \in$ *fml-sem* (*PFadjoint I* σ) φ)))
⟨*proof*⟩

lemma *subst-ode*:
  fixes I:: ('sf, 'sc, 'sz) *interp* **and** ν :: 'sz *state*
  assumes *good-interp*:*is-interp I*
  shows *osafe ODE* $\Longrightarrow$
        ssafe $\sigma \Longrightarrow$
        Oadmit σ *ODE* (*BVO ODE*) $\Longrightarrow$
        ODE-sem I (*Osubst ODE* σ) (*fst* ν) = *ODE-sem* (*adjoint I* σ ν) *ODE* (*fst* ν)
⟨*proof*⟩

lemma *osubst-eq-ODE-vars*: *ODE-vars I* (*Osubst ODE* σ) = *ODE-vars* (*adjoint I* σ ν) *ODE*
⟨*proof*⟩

lemma *subst-semBV*:*semBV* (*adjoint I* σ ν') *ODE* = (*semBV I* (*Osubst ODE* σ))
⟨*proof*⟩

lemma *subst-mkv*:
  fixes I::('sf, 'sc, 'sz) *interp*
  fixes ν::'sz *state*
  fixes ν'::'sz *state*
  assumes *good-interp*:*is-interp I*

assumes *NOU*:*Oadmit* σ *ODE* (*BVO ODE*)
assumes *osafe*:*osafe ODE*
assumes *frees*:*ssafe* σ
shows (*mk-v I* (*Osubst ODE* σ) ν (*fst* ν'))
= (*mk-v* (*adjoint I* σ ν') *ODE* ν (*fst* ν'))
⟨*proof*⟩

lemma *subst-fml-hp*:
fixes *I*::(*'sf*, *'sc*, *'sz*) *interp*
assumes *good-interp*:*is-interp I*
shows
(*Padmit* σ α $\longrightarrow$
(*hpsafe* α $\longrightarrow$
ssafe σ $\longrightarrow$
($\forall$ ν ω. ((ν, ω) $\in$ *prog-sem I* (*Psubst* α σ)) = ((ν, ω) $\in$ *prog-sem* (*adjoint I* σ ν) α))))
$\wedge$
(*Fadmit* σ φ $\longrightarrow$
(*fsafe* φ $\longrightarrow$
ssafe σ $\longrightarrow$
($\forall$ ν. ($\nu \in$ *fml-sem I* (*Fsubst* φ σ)) = ($\nu \in$ *fml-sem* (*adjoint I* σ ν) φ))))
⟨*proof*⟩

lemma *subst-fml*:
fixes *I*::(*'sf*, *'sc*, *'sz*) *interp* **and** ν::*'sz state*
assumes *good-interp*:*is-interp I*
assumes *Fadmit*:*Fadmit* σ φ
assumes *fsafe*:*fsafe* φ
assumes *ssafe*:*ssafe* σ
shows ($\nu \in$ *fml-sem I* (*Fsubst* φ σ)) = ($\nu \in$ *fml-sem* (*adjoint I* σ ν) φ)
⟨*proof*⟩

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

lemma *subst-sequent*:
fixes *I*::(*'sf*, *'sc*, *'sz*) *interp* **and** ν::*'sz state*
assumes *good-interp*:*is-interp I*
assumes *Sadmit*:*Sadmit* σ (Γ,Δ)
assumes *Ssafe*:*Ssafe* (Γ,Δ)
assumes *ssafe*:*ssafe* σ
shows ($\nu \in$ *seq-sem I* (*Ssubst* (Γ,Δ) σ)) = ($\nu \in$ *seq-sem* (*adjoint I* σ ν) (Γ,Δ))

⟨*proof*⟩

11.6 Soundness of substitution rule

theorem *subst-rule*:
  assumes *sound*:*sound R*
  assumes *Radmit*:*Radmit σ R*
  assumes *FVS*:*FVS σ* = {}
  assumes *Rsafe*:*Rsafe R*
  assumes *ssafe*:*ssafe σ*
  shows *sound* (*Rsubst R σ*)
⟨*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* ⇒ '*sz* ⇒ '*sz* ⇒ '*sz*
where *swap x y z* = (*if z* = *x then y else if z* = *y then x else z*)

12.1 Uniform Renaming Definitions

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

primrec *OUrename* :: '*sz* ⇒ '*sz* ⇒ ('*sf*, '*sz*) *ODE* ⇒ ('*sf*, '*sz*) *ODE*

where
  OUrename x y (*OVar c*) = *undefined*
| *OUrename x y* (*OSing z* ϑ) = *OSing* (*swap x y z*) (*TUrename x y* ϑ)
| *OUrename x y* (*OProd ODE1 ODE2*) = *OProd* (*OUrename x y ODE1*) (*OUrename x y ODE2*)

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

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

| *FUrename x y* (*Geq* $\vartheta 1$ $\vartheta 2$) = *Geq* (*TUrename x y* $\vartheta 1$) (*TUrename x y* $\vartheta 2$)
| *FUrename x y* (*Prop p args*) = *Prop p* (λi. *TUrename x y* (*args i*))
| *FUrename x y* (*Not* φ) = *Not* (*FUrename x y* φ)
| *FUrename x y* (*And* φ ψ) = *And* (*FUrename x y* φ) (*FUrename x y* ψ)
| *FUrename x y* (*Exists z* φ) = *Exists* (*swap x y z*) (*FUrename x y* φ)
| *FUrename x y* (*Diamond* α φ) = *Diamond* (*PUrename x y* α) (*FUrename x y* φ)
| *FUrename x y* (*InContext C* φ) = *undefined*

12.2 Uniform Renaming Admissibility

inductive *PRadmit* :: ('*sf*, '*sc*, '*sz*) *hp* ⇒ *bool*
  and *FRadmit* ::('*sf*, '*sc*, '*sz*) *formula* ⇒ *bool*
where
  PRadmit-Assign:*PRadmit* (*Assign x* ϑ)
| *PRadmit-DiffAssign*:*PRadmit* (*DiffAssign x* ϑ)
| *PRadmit-Test*:*FRadmit* φ ⟹ *PRadmit* (*Test* φ)
| *PRadmit-EvolveODE*:*ORadmit ODE* ⟹ *FRadmit* φ ⟹ *PRadmit* (*EvolveODE ODE* φ)
| *PRadmit-Choice*:*PRadmit a* ⟹ *PRadmit b* ⟹ *PRadmit* (*Choice a b*)
| *PRadmit-Sequence*:*PRadmit a* ⟹ *PRadmit b* ⟹ *PRadmit* (*Sequence a b*)
| *PRadmit-Loop*:*PRadmit a* ⟹ *PRadmit* (*Loop a*)

| *FRadmit-Geq*:*FRadmit* (*Geq* $\vartheta 1$ $\vartheta 2$)
| *FRadmit-Prop*:*FRadmit* (*Prop p args*)

| *FRadmit-Not*:*FRadmit* $\varphi \Longrightarrow$ *FRadmit* (*Not* φ)
| *FRadmit-And*:*FRadmit* $\varphi \Longrightarrow$ *FRadmit* $\psi \Longrightarrow$ *FRadmit* (*And* φ ψ)
| *FRadmit-Exists*:*FRadmit* $\varphi \Longrightarrow$ *FRadmit* (*Exists x* φ)
| *FRadmit-Diamond*:*PRadmit* $\alpha \Longrightarrow$ *FRadmit* $\varphi \Longrightarrow$ *FRadmit* (*Diamond* α φ)

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

definition *RSadj* :: *'sz* ⇒ *'sz* ⇒ *'sz simple-state* ⇒ *'sz simple-state*
where *RSadj x y* ν = (χ *z*. ν $ (*swap x y z*))

definition *Radj* :: *'sz* ⇒ *'sz* ⇒ *'sz state* ⇒ *'sz state*
where *Radj x y* ν = (*RSadj x y* (*fst* ν), *RSadj x y* (*snd* ν))

lemma *SUren*: *sterm-sem I* (*TUrename x y* ϑ) ν = *sterm-sem I* ϑ (*RSadj x y* ν)
⟨*proof*⟩

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

12.3 Uniform Renaming Soundness Proof and Lemmas

lemma *TUren-frechet*:
assumes *good-interp*:*is-interp I*
shows *dfree* $\vartheta \Longrightarrow$ *frechet I* (*TUrename x y* ϑ) ν ν' = *frechet I* ϑ (*RSadj x y* ν) (*RSadj x y* ν')
⟨*proof*⟩

lemma *RSadj-fst*:*RSadj x y* (*fst* ν) = *fst* (*Radj x y* ν)
⟨*proof*⟩

lemma *RSadj-snd*:*RSadj x y* (*snd* ν) = *snd* (*Radj x y* ν)
⟨*proof*⟩

lemma *TUren*:
assumes *good-interp*:*is-interp I*
shows *dsafe* $\vartheta \Longrightarrow$ *dterm-sem I* (*TUrename x y* ϑ) ν = *dterm-sem I* ϑ (*Radj x y* ν)
⟨*proof*⟩

lemma *adj-sum*:*RSadj x y* ($\nu 1$ + $\nu 2$) = (*RSadj x y* $\nu 1$) + (*RSadj x y* $\nu 2$)
⟨*proof*⟩

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

lemma *state-eq*:

fixes ν ν' :: $'sz$ *state*
shows $(\bigwedge i.\ (fst\ \nu)\ \$\ i = (fst\ \nu')\ \$\ i) \Longrightarrow (\bigwedge i.\ (snd\ \nu)\ \$\ i = (snd\ \nu')\ \$\ i) \Longrightarrow \nu\ = \nu'$
⟨*proof*⟩

lemma *Radj-repv1*:
fixes $x\ y\ z :: 'sz$
shows $(Radj\ x\ y\ (repv\ \nu\ y\ r)) = repv\ (Radj\ x\ y\ \nu)\ x\ r$
⟨*proof*⟩

lemma *Radj-repv2*:
fixes $x\ y\ z :: 'sz$
shows $(Radj\ x\ y\ (repv\ \nu\ x\ r)) = repv\ (Radj\ x\ y\ \nu)\ y\ r$
⟨*proof*⟩

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

lemma *Radj-repd1*:
fixes $x\ y\ z :: 'sz$
shows $(Radj\ x\ y\ (repd\ \nu\ y\ r)) = repd\ (Radj\ x\ y\ \nu)\ x\ r$
⟨*proof*⟩

lemma *Radj-repd2*:
fixes $x\ y\ z :: 'sz$
shows $(Radj\ x\ y\ (repd\ \nu\ x\ r)) = repd\ (Radj\ x\ y\ \nu)\ y\ r$
⟨*proof*⟩

lemma *Radj-repd3*:
fixes $x\ y\ z :: 'sz$
assumes $zx{:}z \neq x$ **and** $zy{:}z \neq y$
shows $(Radj\ x\ y\ (repd\ \nu\ z\ r)) = repd\ (Radj\ x\ y\ \nu)\ z\ r$
⟨*proof*⟩
lemma *Radj-eq-iff*:$(a = b) = ((Radj\ x\ y\ a) = (Radj\ x\ y\ b))$
⟨*proof*⟩

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

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

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

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* ≤ *t*
assumes *fml*:⋀*ν*. (*ν* ∈ *fml-sem I* (*FUrename x y φ*)) = (*Radj x y ν* ∈ *fml-sem I φ*)
assumes *sol*:(*sol solves-ode* (*λa*. *ODE-sem I* (*OUrename x y ODE*))) {*0*..*t*} {*xa*. *mk-v I* (*OUrename x y ODE*) (*sol 0*, *b*) *xa* ∈ *fml-sem I* (*FUrename x y φ*)}
shows ((*λt*. *RSadj x y* (*sol t*)) *solves-ode* (*λa*. *ODE-sem I ODE*)) {*0*..*t*} {*xa*. *mk-v I ODE* (*RSadj x y* (*sol 0*), *RSadj x y b*) *xa* ∈ *fml-sem I φ*}
⟨*proof*⟩

lemma *sol-lemma2*:
assumes *ORA*:*ORadmit ODE*
assumes *t*:*0* ≤ *t*
assumes *fml*:⋀*ν*. (*ν* ∈ *fml-sem I* (*FUrename x y φ*)) = (*Radj x y ν* ∈ *fml-sem I φ*)
assumes *sol*:(*sol solves-ode* (*λa*. *ODE-sem I ODE*)) {*0*..*t*} {*x*. *mk-v I ODE* (*sol 0*, *b*) *x* ∈ *fml-sem I φ*}
shows ((*λt*. *RSadj x y* (*sol t*)) *solves-ode* (*λa*. *ODE-sem I* (*OUrename x y ODE*))) {*0*..*t*}
{*xa*. *mk-v I* (*OUrename x y ODE*) (*RSadj x y* (*sol 0*), *RSadj x y b*) *xa* ∈ *fml-sem I* (*FUrename x y φ*)}
⟨*proof*⟩

lemma *PUren-FUren*:
assumes *good-interp*:*is-interp I*
shows
(*PRadmit α* ⟶ *hpsafe α* ⟶ (∀ *ν ω*. (*ν*, *ω*) ∈ *prog-sem I* (*PUrename x y α*) ⟷ (*Radj x y ν*, *Radj x y ω*) ∈ *prog-sem I α*))
∧ (*FRadmit φ* ⟶ *fsafe φ* ⟶ (∀ *ν*. *ν* ∈ *fml-sem I* (*FUrename x y φ*) ⟷ (*Radj x y ν*) ∈ *fml-sem I φ*))
⟨*proof*⟩

lemma *FUren*:*is-interp I* ⟹ *FRadmit φ* ⟹ *fsafe φ* ⟹ (⋀*ν*. (*ν* ∈ *fml-sem I* (*FUrename x y φ*)) = (*Radj x y ν* ∈ *fml-sem I φ*))
⟨*proof*⟩

12.4 Uniform Renaming Rule Soundness

lemma *URename-sound*:*FRadmit* $\varphi \Longrightarrow$ *fsafe* $\varphi \Longrightarrow$ *valid* $\varphi \Longrightarrow$ *valid* (*FUrename* $x\ y\ \varphi$)
⟨*proof*⟩

12.5 Bound Renaming Rule Soundness

lemma *BRename-sound*:
assumes *FRA*:*FRadmit*([[*Assign* $x\ \vartheta$]]φ)
assumes *fsafe*:*fsafe* ([[*Assign* $x\ \vartheta$]]φ)
assumes *valid*:*valid* ([[*Assign* $x\ \vartheta$]]φ)
assumes *FVF*:{*Inl* y, *Inr* y, *Inr* x} $\cap$ *FVF* φ = {}
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* ⇒ *char list list* ⇒ *char list*
where *join* S [] = []
| *join* S [S'] = S'
| *join* S (S' # *SS*) = S' @ S @ (*join* S *SS*)

fun *vid-to-string*::'*sz* ⇒ *char list*
where *vid-to-string vid* = (*if vid* = *vid1 then* ''*x*'' *else if vid* = *vid2 then* ''*y*'' *else if vid* = *vid3 then* ''*z*'' *else* ''*w*'')

fun *oid-to-string*::'*sz* ⇒ *char list*
where *oid-to-string vid* = (*if vid* = *vid1 then* ''*c*'' *else if vid* = *vid2 then* ''*c2*'' *else if vid* = *vid3 then* ''*c3*'' *else* ''*c4*'')

fun cid-to-string::'sc ⇒ char list
where cid-to-string vid = (if vid = pid1 then ''C'' else if vid = pid2 then ''C2'' else if vid = pid3 then ''C3'' else ''C4'')

fun ppid-to-string::'sc ⇒ char list
where ppid-to-string vid = (if vid = pid1 then ''P'' else if vid = pid2 then ''Q'' else if vid = pid3 then ''R'' else ''H'')

fun hpid-to-string::'sz ⇒ char list
where hpid-to-string vid = (if vid = vid1 then ''a'' else if vid = vid2 then ''b'' else if vid = vid3 then ''a1'' else ''b1'')

fun fid-to-string::'sf ⇒ char list
where fid-to-string vid = (if vid = fid1 then ''f'' else if vid = fid2 then ''g'' else if vid = fid3 then ''h'' else ''j'')

primrec trm-to-string::('sf,'sz) trm ⇒ char list
where
trm-to-string (Var x) = vid-to-string x
| trm-to-string (Const r) = ''r''
| trm-to-string (Function f args) = fid-to-string f
| trm-to-string (Plus t1 t2) = trm-to-string t1 @ ''+'' @ trm-to-string t2
| trm-to-string (Times t1 t2) = trm-to-string t1 @ ''*'' @ trm-to-string t2
| trm-to-string (DiffVar x) = ''Dv{'' @ vid-to-string x @ ''}''
| trm-to-string (Differential t) = ''D{'' @ trm-to-string t @ ''}''

primrec ode-to-string::('sf,'sz) ODE ⇒ char list
where
ode-to-string (OVar x) = oid-to-string x
| ode-to-string (OSing x t) = ''d'' @ vid-to-string x @ ''='' @ trm-to-string t
| ode-to-string (OProd ODE1 ODE2) = ode-to-string ODE1 @ '', '' @ ode-to-string ODE2

fun fml-to-string ::('sf, 'sc, 'sz) formula ⇒ char list
and hp-to-string ::('sf, 'sc, 'sz) hp ⇒ char list
where
fml-to-string (Geq t1 t2) = trm-to-string t1 @ ''>='' @ trm-to-string t2
| fml-to-string (Prop p args) = []
| fml-to-string (Not p) =
(case p of (And (Not q) (Not (Not p))) ⇒ fml-to-string p @ ''−>'' @ fml-to-string q
| (Exists x (Not p)) ⇒ ''A''@ vid-to-string x @ ''.'' @ fml-to-string p
| (Diamond a (Not p)) ⇒ ''[''@ hp-to-string a @ '']'' @ fml-to-string p
| (And (Not (And p q)) (Not (And (Not p') (Not q')))) ⇒
(if (p = p' ∧ q = q') then fml-to-string p @ ''<−>'' @ fml-to-string q else ''!'' @ fml-to-string (And (Not (And p q)) (Not (And (Not p') (Not q')))))
| - ⇒ ''!'' @ fml-to-string p)
| fml-to-string (And p q) = fml-to-string p @ ''&'' @ fml-to-string q
| fml-to-string (Exists x p) = ''E'' @ vid-to-string x @ '' . '' @ fml-to-string p

| *fml-to-string* (*Diamond a p*) = ''<'' @ *hp-to-string a* @ ''>'' @ *fml-to-string p*
| *fml-to-string* (*InContext C p*) =
(*case p of*
(*Geq - -*) ⇒ *ppid-to-string C*
| - ⇒ *cid-to-string C* @ ''('' @ *fml-to-string p* @ '')'')

| *hp-to-string* (*Pvar a*) = *hpid-to-string a*
| *hp-to-string* (*Assign x e*) = *vid-to-string x* @ '':='' @ *trm-to-string e*
| *hp-to-string* (*DiffAssign x e*) = ''D{'' @ *vid-to-string x* @ ''}:='' @ *trm-to-string e*
| *hp-to-string* (*Test p*) = ''?'' @ *fml-to-string p*
| *hp-to-string* (*EvolveODE ODE p*) = ''{'' @ *ode-to-string ODE* @ ''&'' @ *fml-to-string p* @ ''}''
| *hp-to-string* (*Choice a b*) = *hp-to-string a* @ ''U'' @ *hp-to-string b*
| *hp-to-string* (*Sequence a b*) = *hp-to-string a* @ '';'' @ *hp-to-string b*
| *hp-to-string* (*Loop a*) = *hp-to-string a* @ ''*''

end end
theory *Proof-Checker*
imports
Ordinary-Differential-Equations.ODE-Analysis
Ids
Lib
Syntax
Denotational-Semantics
Axioms
Differential-Axioms
Frechet-Correctness
Static-Semantics
Coincidence
Bound-Effect
Uniform-Renaming
USubst-Lemma
Pretty-Printer

begin context *ids* **begin**

14 Proof Checker

This proof checker defines a datatype for proof terms in dL and a function for checking proof terms, with a soundness proof that any proof accepted by the checker is a proof of a sound rule or valid formula.

A simple concrete hybrid system and a differential invariant rule for conjunctions are provided as example proofs.

lemma *sound-weaken-gen*: $\bigwedge A\ B\ C.\ sublist\ A\ B \Longrightarrow sound\ (A,\ C) \Longrightarrow sound\ (B,C)$
⟨*proof*⟩

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

lemma *member-filter*:$\bigwedge$*P. List.member* (*filter P L*) *x* $\Longrightarrow$ *List.member L x*
⟨*proof*⟩

lemma *nth-member*:*n* < *List.length L* $\Longrightarrow$ *List.member L* (*List.nth L n*)
⟨*proof*⟩

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

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

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

lemma *start-proof-sound*:*sound* (*start-proof S*)
⟨*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* ⇒ (*'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*)) @ ''   '' @ ~~[*char-of-nat* 10] @~~ *seq-to-string C*

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

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

lemma *close-sub*:*sublist* (*close* Γ φ) Γ
⟨*proof*⟩

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

lemma *close-provable-sound*:*sound* (*SG*, *C*) ⟹ *sound* (*close SG* φ, φ) ⟹ *sound* (*close SG* φ, *C*)
⟨*proof*⟩

fun *Lrule-result* :: *lrule* ⇒ *nat* ⇒ ('*sf*, '*sc*, '*sz*) *sequent* ⇒ ('*sf*, '*sc*, '*sz*) *sequent list*
where *Lrule-result AndL j* (*A*,*S*) = (*case* (*nth A j*) *of And p q* ⇒ [(*close* ([*p*, *q*] @ *A*) (*nth A j*), *S*)])
| *Lrule-result ImplyL j* (*A*,*S*) = (*case* (*nth A j*) *of Not* (*And* (*Not q*) (*Not* (*Not p*))) ⇒
[(*close* (*q* # *A*) (*nth A j*), *S*), (*close A* (*nth A j*), *p* # *S*)])
| *Lrule-result EquivForwardL j* (*A*,*S*) = (*case* (*nth A j*) *of Not*(*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p*′) (*Not q*′)))) ⇒
[(*close* (*q* # *A*) (*nth A j*), *S*), (*close A* (*nth A j*), *p* # *S*)])
| *Lrule-result EquivBackwardL j* (*A*,*S*) = (*case* (*nth A j*) *of Not*(*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p*′) (*Not q*′)))) ⇒
[(*close* (*p* # *A*) (*nth A j*), *S*), (*close A* (*nth A j*), *q* # *S*)])

— Note: Some of the pattern-matching here is... interesting. The reason for this is that we can only
— match on things in the base grammar, when we would quite like to check things in the derived grammar.
— So all the pattern-matches have the definitions expanded, sometimes in a silly way.
fun *Rrule-result* :: *rrule* ⇒ *nat* ⇒ ('*sf*, '*sc*, '*sz*) *sequent* ⇒ ('*sf*, '*sc*, '*sz*) *sequent list*
where
Rstep-Imply:*Rrule-result ImplyR j* (*A*,*S*) = (*case* (*nth S j*) *of Not* (*And* (*Not q*) (*Not* (*Not p*))) ⇒ [(*p* # *A*, *q* # (*closeI S j*))] | - ⇒ *undefined*)
| *Rstep-And*:*Rrule-result AndR j* (*A*,*S*) = (*case* (*nth S j*) *of* (*And p q*) ⇒ [(*A*, *p* # (*closeI S j*)), (*A*, *q* # (*closeI S j*))])
| *Rstep-EquivR*:*Rrule-result EquivR j* (*A*,*S*) =
(*case* (*nth S j*) *of Not*(*And* (*Not* (*And p q*)) (*Not* (*And* (*Not p*′) (*Not q*′)))) ⇒
(*if* (*p* = *p*′ ∧ *q* = *q*′) *then* [(*p* # *A*, *q* # (*closeI S j*)), (*q* # *A*, *p* # (*closeI S j*))]
else undefined))
| *Rstep-CohideR*:*Rrule-result CohideR j* (*A*,*S*) = [(*A*, [*nth S j*])]
| *Rstep-CohideRR*:*Rrule-result CohideRR j* (*A*,*S*) = [([], [*nth S j*])]
| *Rstep-TrueR*:*Rrule-result TrueR j* (*A*,*S*) = []

fun *step-result* :: ('*sf*, '*sc*, '*sz*) *rule* ⇒ (*nat* ∗ ('*sf*, '*sc*, '*sz*) *step*) ⇒ ('*sf*, '*sc*, '*sz*) *rule*
where

Step-axiom:step-result (*SG*,*C*) (*i*,*Axiom a*) = (*closeI SG i*, *C*)
| *Step-AxSubst:step-result* (*SG*,*C*) (*i*,*AxSubst a* σ) = (*closeI SG i*, *C*)
| *Step-Lrule:step-result* (*SG*,*C*) (*i*,*Lrule L j*) = (*close* (*append SG* (*Lrule-result L j* (*nth SG i*))) (*nth SG i*), *C*)
| *Step-Rrule:step-result* (*SG*,*C*) (*i*,*Rrule L j*) = (*close* (*append SG* (*Rrule-result L j* (*nth SG i*))) (*nth SG i*), *C*)
| *Step-Cut:step-result* (*SG*,*C*) (*i*,*Cut* φ) = (*let* (*A*,*S*) = *nth SG i in* ((φ # *A*, *S*) # ((*A*, φ # *S*) # (*closeI SG i*)), *C*))
| *Step-Vsubst:step-result* (*SG*,*C*) (*i*,*VSubst* φ σ) = (*closeI SG i*, *C*)
| *Step-CloseId:step-result* (*SG*,*C*) (*i*,*CloseId j k*) = (*closeI SG i*, *C*)
| *Step-G:step-result* (*SG*,*C*) (*i*,*G*) = (*case nth SG i of* (-, (*Not* (*Diamond q* (*Not p*))) # *Nil*) ⇒ (([], [*p*]) # *closeI SG i*, *C*))
| *Step-DEAxiomSchema:step-result* (*SG*,*C*) (*i*,*DEAxiomSchema ODE* σ) = (*closeI SG i*, *C*)
| *Step-CE:step-result* (*SG*,*C*) (*i*, *CE* φ ψ σ) = (*closeI SG i*, *C*)
| *Step-CQ:step-result* (*SG*,*C*) (*i*, *CQ* ϑ_1 ϑ_2 σ) = (*closeI SG i*, *C*)
| *Step-default:step-result R* (*i*,*S*) = *R*

fun *deriv-result* :: ('*sf*, '*sc*, '*sz*) *rule* ⇒ ('*sf*, '*sc*, '*sz*) *derivation* ⇒ ('*sf*, '*sc*, '*sz*) *rule*
where
deriv-result R [] = *R*
| *deriv-result R* (*s* # *ss*) = *deriv-result* (*step-result R s*) (*ss*)

fun *proof-result* :: ('*sf*, '*sc*, '*sz*) *pf* ⇒ ('*sf*, '*sc*, '*sz*) *rule*
where *proof-result* (*D*,*S*) = *deriv-result* (*start-proof D*) *S*

inductive *lrule-ok* ::('*sf*,'*sc*,'*sz*) *sequent list* ⇒ ('*sf*,'*sc*,'*sz*) *sequent* ⇒ *nat* ⇒ *nat* ⇒ *lrule* ⇒ *bool*
where
Lrule-And:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* && *q*) ⟹ *lrule-ok SG C i j AndL*
| *Lrule-Imply*:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* → *q*) ⟹ *lrule-ok SG C i j ImplyL*
| *Lrule-EquivForward*:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* ↔ *q*) ⟹ *lrule-ok SG C i j EquivForwardL*
| *Lrule-EquivBackward*:⋀*p q*. *nth* (*fst* (*nth SG i*)) *j* = (*p* ↔ *q*) ⟹ *lrule-ok SG C i j EquivBackwardL*

named-theorems *prover Simplification rules for checking validity of proof certificates*
lemmas [*prover*] = *axiom-defs Box-def Or-def Implies-def filter-append ssafe-def SDom-def FUadmit-def PFUadmit-def id-simps*

inductive-simps
Lrule-And[*prover*]: *lrule-ok SG C i j AndL*
and *Lrule-Imply*[*prover*]: *lrule-ok SG C i j ImplyL*
and *Lrule-Forward*[*prover*]: *lrule-ok SG C i j EquivForwardL*
and *Lrule-EquivBackward*[*prover*]: *lrule-ok SG C i j EquivBackwardL*

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

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

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

inductive *step-ok* :: ('*sf*, '*sc*, '*sz*) *rule* $\Rightarrow$ *nat* $\Rightarrow$ ('*sf*, '*sc*, '*sz*) *step* $\Rightarrow$ *bool*
where
Step-Axiom:(*nth SG i*) = ([], [*get-axiom a*]) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*Axiom a*)
| *Step-AxSubst*:(*nth SG i*) = ([], [*Fsubst* (*get-axiom a*) σ]) $\Longrightarrow$ *Fadmit* σ (*get-axiom a*) $\Longrightarrow$ *ssafe* σ $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*AxSubst a* σ)
| *Step-Lrule*:*lrule-ok SG C i j L* $\Longrightarrow$ *j* < *length* (*fst* (*nth SG i*)) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*Lrule L j*)
| *Step-Rrule*:*rrule-ok SG C i j L* $\Longrightarrow$ *j* < *length* (*snd* (*nth SG i*)) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*Rrule L j*)
| *Step-Cut*:*fsafe* φ $\Longrightarrow$ *i* < *length SG* $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*Cut* φ)
| *Step-CloseId*:*nth* (*fst* (*nth SG i*)) *j* = *nth* (*snd* (*nth SG i*)) *k* $\Longrightarrow$ *j* < *length* (*fst* (*nth SG i*)) $\Longrightarrow$ *k* < *length* (*snd* (*nth SG i*)) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*CloseId j k*)
| *Step-G*:$\bigwedge a\ p.$ *nth SG i* = ([], [(([[*a*]]*p*)]) $\Longrightarrow$ *step-ok* (*SG*,*C*) *i G*
| *Step-DEAxiom-schema*:
nth SG i =
([], [*Fsubst* (((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*)) *ODE*) (*p1 vid2 vid1*)]] (*P pid1*)) $\leftrightarrow$
([[*EvolveODE* ((*OProd* (*OSing vid1* (*f1 fid1 vid1*))) *ODE*) (*p1 vid2 vid1*)]]
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*))) σ])
$\Longrightarrow$ *ssafe* σ
$\Longrightarrow$ *osafe ODE*
$\Longrightarrow$ {*Inl vid1*, *Inr vid1*} $\cap$ *BVO ODE* = {}
$\Longrightarrow$ *Fadmit* σ (((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*) (*p1 vid2 vid1*)]] (*P pid1*)) $\leftrightarrow$
([[*EvolveODE* ((*OProd* (*OSing vid1* (*f1 fid1 vid1*))*ODE*)) (*p1 vid2 vid1*)]]
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*)))
$\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*DEAxiomSchema ODE* σ)
| *Step-CE*:*nth SG i* = ([], [*Fsubst* (*Equiv* (*InContext pid1* φ) (*InContext pid1* ψ)) σ])
$\Longrightarrow$ *valid* (*Equiv* φ ψ)

$\Longrightarrow$ *fsafe* φ
$\Longrightarrow$ *fsafe* ψ
$\Longrightarrow$ *ssafe* σ
$\Longrightarrow$ *Fadmit* σ (*Equiv* (*InContext pid1* φ) (*InContext pid1* ψ))
$\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*CE* φ ψ σ)
| *Step-CQ*:*nth SG i* = ([], [*Fsubst* (*Equiv* (*Prop p* (*singleton* ϑ)) (*Prop p* (*singleton* ϑ'))) σ])
$\Longrightarrow$ *valid* (*Equals* ϑ ϑ')
$\Longrightarrow$ *dsafe* ϑ
$\Longrightarrow$ *dsafe* ϑ'
$\Longrightarrow$ *ssafe* σ
$\Longrightarrow$ *Fadmit* σ (*Equiv* (*Prop p* (*singleton* ϑ)) (*Prop p* (*singleton* ϑ')))
$\Longrightarrow$ *step-ok* (*SG*,*C*) *i* (*CQ* ϑ ϑ' σ)

inductive-simps
Step-G-simps[*prover*]: *step-ok* (*SG*,*C*) *i G*
and *Step-CloseId-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*CloseId j k*)
and *Step-Cut-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Cut* φ)
and *Step-Rrule-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Rrule j L*)
and *Step-Lrule-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Lrule j L*)
and *Step-Axiom-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*Axiom a*)
and *Step-AxSubst-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*AxSubst a* σ)
and *Step-DEAxiom-schema-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*DEAxiomSchema ODE* σ)
and *Step-CE-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*CE* φ ψ σ)
and *Step-CQ-simps*[*prover*]: *step-ok* (*SG*,*C*) *i* (*CQ* ϑ ϑ' σ)

inductive *deriv-ok* :: (*'sf*, *'sc*, *'sz*) *rule* ⇒ (*'sf*, *'sc*, *'sz*) *derivation* ⇒ *bool*
where
Deriv-Nil:*deriv-ok R Nil*
| *Deriv-Cons*:*step-ok R i S* $\Longrightarrow$ $i \geq 0$ $\Longrightarrow$ $i <$ *length* (*fst R*) $\Longrightarrow$ *deriv-ok* (*step-result R* (*i*,*S*)) *SS* $\Longrightarrow$ *deriv-ok R* ((*i*,*S*) # *SS*)

inductive-simps
Deriv-nil-simps[*prover*]: *deriv-ok R Nil*
and *Deriv-cons-simps*[*prover*]: *deriv-ok R* ((*i*,*S*)#*SS*)

inductive *proof-ok* :: (*'sf*, *'sc*, *'sz*) *pf* ⇒ *bool*
where
Proof-ok:*deriv-ok* (*start-proof D*) *S* $\Longrightarrow$ *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*]:$\bigwedge x$. *List.member* [*x*] *x*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SG i))) (*nth SG i*), *C*)
⟨*proof*⟩

lemma *step-sound*:*step-ok R i S* ⟹ *i* ≥ *0* ⟹ *i* < *length* (*fst R*) ⟹ *sound R* ⟹ *sound* (*step-result R* (*i*,*S*))
⟨*proof*⟩

lemma *deriv-sound*:*deriv-ok R D* ⟹ *sound R* ⟹ *sound* (*deriv-result R D*)
⟨*proof*⟩

lemma *proof-sound*:*proof-ok Pf* ⟹ *sound* (*proof-result Pf*)
⟨*proof*⟩

16 Example 1: Differential Invariants

definition *DIAndConcl*::('*sf*,'*sc*,'*sz*) *sequent*
where *DIAndConcl* = ([], [*Implies* (*And* (*Predicational pid1*) (*Predicational pid2*))

(*Implies* ([[*Pvar vid1*]](*And* (*Predicational pid3*) (*Predicational pid4*)))
([[*Pvar vid1*]](*And* (*Predicational pid1*) (*Predicational pid2*))))])

definition *DIAndSG1*::('*sf*,'*sc*,'*sz*) *formula*
where *DIAndSG1* = (*Implies* (*Predicational pid1*) (*Implies* ([[*Pvar vid1*]](*Predicational pid3*)) ([[*Pvar vid1*]](*Predicational pid1*))))

definition *DIAndSG2*::('*sf*,'*sc*,'*sz*) *formula*
where *DIAndSG2* = (*Implies* (*Predicational pid2*) (*Implies* ([[*Pvar vid1*]](*Predicational pid4*)) ([[*Pvar vid1*]](*Predicational pid2*))))

definition *DIAndCut*::('*sf*,'*sc*,'*sz*) *formula*
where *DIAndCut* =
(([[\$α *vid1*]]((*And* (*Predicational* (*pid3*)) (*Predicational* (*pid4*)))) → (*And* (*Predicational* (*pid1*)) (*Predicational* (*pid2*))))
→ ([[\$α *vid1*]](*And* (*Predicational* (*pid3*)) (*Predicational* (*pid4*)))) → ([[\$α *vid1*]](*And* (*Predicational* (*pid1*)) (*Predicational* (*pid2*)))))

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

— [*a*]*R*&*H*−>*R*−>[*a*]*R*&*H*−>[*a*]*R DIAndSubst34*

definition *DIAndSubst341*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndSubst341* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*And* (*Predicational* (*Inl pid3*)) (*Predicational* (*Inl pid4*)))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid3*)) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)
definition *DIAndSubst342*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndSubst342* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*And* (*Predicational* (*Inl pid3*)) (*Predicational* (*Inl pid4*)))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid4*)) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)

— [*a*]*P*, [*a*]*R*&*H*, *P*, *Q* |− [*a*]*Q*−>*P*&*Q*−>[*a*]*Q*−>[*a*]*P*&*Q*, [*a*]*P*&*Q*;;
definition *DIAndSubst12*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndSubst12* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*Predicational* (*Inl pid2*))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid1*) && *Predicational* (*Inl pid2*)) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)

— *P* −> *Q*−>*P*&*Q*
definition *DIAndCurry12*::('*sf*,'*sc*,'*sz*) *subst*
where *DIAndCurry12* =
(| *SFunctions* = (λ-. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. (*if C* = *pid1 then Some*(*Predicational* (*Inl pid1*))
else (*if C* = *pid2 then Some*(*Predicational* (*Inl pid2*) → (*Predicational* (*Inl pid1*) && *Predicational* (*Inl pid2*))) *else None*))),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
|)

definition *DIAnd* :: ('*sf*,'*sc*,'*sz*) *rule*
where *DIAnd* =
([([],[*DIAndSG1*]),([],[*DIAndSG2*])],
DIAndConcl)

```
definition DIAndCutP1 :: ('sf,'sc,'sz) formula
where DIAndCutP1 = ([[Pvar vid1]](Predicational pid1))
```

```
definition DIAndCutP2 :: ('sf,'sc,'sz) formula
where DIAndCutP2 = ([[Pvar vid1]](Predicational pid2))
```

```
definition DIAndCutP12 :: ('sf,'sc,'sz) formula
where DIAndCutP12 = ((([[Pvar vid1]](Pc pid1) → (Pc pid2 → (And (Pc pid1)
(Pc pid2))))
  → ((([[Pvar vid1]]Pc pid1) → ([[Pvar vid1]](Pc pid2 → (And (Pc pid1) (Pc
pid2))))))
```

```
definition DIAndCut34Elim1 :: ('sf,'sc,'sz) formula
where DIAndCut34Elim1 = ((([[Pvar vid1]](Pc pid3 && Pc pid4) → (Pc pid3))
 → ((([[Pvar vid1]](Pc pid3 && Pc pid4)) → ([[Pvar vid1]](Pc pid3))))
```

```
definition DIAndCut34Elim2 :: ('sf,'sc,'sz) formula
where DIAndCut34Elim2 = ((([[Pvar vid1]](Pc pid3 && Pc pid4) → (Pc pid4))
 → ((([[Pvar vid1]](Pc pid3 && Pc pid4)) → ([[Pvar vid1]](Pc pid4))))
```

```
definition DIAndCut12Intro :: ('sf,'sc,'sz) formula
where DIAndCut12Intro = ((([[Pvar vid1]](Pc pid2 → (Pc pid1 && Pc pid2)))
 → ((([[Pvar vid1]](Pc pid2)) → ([[Pvar vid1]](Pc pid1 && Pc pid2))))
```

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

```
,(Suc (Suc 0), CloseId 0 0)
,(Suc 0, CloseId 0 0)
,(0, Cut DIAndCut12Intro)
,(Suc 0, Rrule CohideRR 0)
,(Suc (Suc 0), AxSubst AK DIAndSubst12)
,(0, Lrule ImplyL 0)
,(1, Lrule ImplyL 0)
,(Suc (Suc 0), CloseId 0 0)
,(Suc 0, Cut DIAndCutP12)
,(0, Lrule ImplyL 0) — 31
,(0, Rrule CohideRR 0)
,(Suc (Suc (Suc (Suc 0))), AxSubst AK DIAndCurry12)
,(Suc (Suc (Suc 0)), Rrule CohideRR 0)
,(Suc (Suc 0), Lrule ImplyL 0)
,(Suc (Suc 0), G)
,(0, Rrule ImplyR 0)
,(Suc (Suc (Suc (Suc 0))), Rrule ImplyR 0)
,(Suc (Suc (Suc (Suc 0))), Rrule AndR 0)
,(Suc (Suc (Suc (Suc (Suc 0)))), CloseId 0 0)
,(Suc (Suc (Suc (Suc 0))), CloseId 1 0) — 41
,(Suc (Suc  0), CloseId 0 0)
,(Suc 0, Cut DIAndCut34Elim2)
,(0, Lrule ImplyL 0)
,(0, Rrule CohideRR 0)
,(Suc (Suc (Suc (Suc 0))), AxSubst AK DIAndSubst342) — 46
,(Suc (Suc (Suc 0)), Rrule CohideRR 0)
,(Suc (Suc (Suc 0)), G) — 48
,(0, Rrule ImplyR 0)
,(Suc (Suc (Suc 0)), Lrule AndL 0) — 50
,(Suc (Suc (Suc 0)), CloseId 1 0)
,(Suc (Suc 0), Lrule ImplyL 0)
,(Suc 0, CloseId 0 0)
,(1, Cut DIAndSG2)
,(0, Lrule ImplyL 0)
,(0, Rrule CohideRR 0)
,(Suc (Suc (Suc 0)), CloseId 4 0)
,(Suc (Suc 0), Lrule ImplyL 0)
,(Suc (Suc (Suc 0)), CloseId 0 0)
,(Suc (Suc (Suc 0)), CloseId 0 0)
,(1, CloseId 1 0)
])
```

fun *proof-take* :: *nat* ⇒ (*'sf*,*'sc*,*'sz*) *pf* ⇒ (*'sf*,*'sc*,*'sz*) *pf*
where *proof-take n* (*C*,*D*) = (*C*,*List.take n D*)

fun *last-step*::(*'sf*,*'sc*,*'sz*) *pf* ⇒ *nat* ⇒ *nat* ∗ (*'sf*,*'sc*,*'sz*) *step*
where *last-step* (*C*,*D*) *n* = *List.last* (*take n D*)

lemma *DIAndSound-lemma*:*sound* (*proof-result* (*proof-take 61 DIAndProof*))
⟨*proof*⟩

17 Example 2: Concrete Hybrid System

definition *SystemConcl*::(′*sf*,′*sc*,′*sz*) *sequent*
where *SystemConcl* =
([],[
Implies (*And* (*Geq* (*Var vid1*) (*Const 0*)) (*Geq* (*f0 fid1*) (*Const 0*)))
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*TT*)]]*Geq* (*Var vid1*) (*Const 0*))
])

definition *SystemDICut* :: (′*sf*,′*sc*,′*sz*) *formula*
where *SystemDICut* =
Implies
(*Implies TT* ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]]
(*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))
(*Implies*
(*Implies TT* (*Geq* (*Var vid1*) (*Const 0*)))
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]](*Geq* (*Var vid1*) (*Const 0*))))

definition *SystemDCCut*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemDCCut* =
((([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]](*Geq* (*f0 fid1*) (*Const 0*))) →
(([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]]((*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))
↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))))

definition *SystemVCut*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemVCut* =
Implies (*Geq* (*f0 fid1*) (*Const 0*)) ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]](*Geq* (*f0 fid1*) (*Const 0*)))

definition *SystemVCut2*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemVCut2* =
Implies (*Geq* (*f0 fid1*) (*Const 0*)) ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) *TT*]](*Geq* (*f0 fid1*) (*Const 0*)))

definition *SystemDECut*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemDECut* = (([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ((*Geq* (*Differential* (*Var vid1*))

(*Differential* (*Const 0*))))) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]]
[[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))

definition *SystemKCut*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemKCut* =
(*Implies* ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]]
(*Implies* ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))) ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))))
(*Implies* ([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))))
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))))))

definition *SystemEquivCut*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemEquivCut* =
(*Equiv* (*Implies* ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))) ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))))
(*Implies* ((*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))) ([[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*DiffVar vid1*) (*Const 0*)))))

definition *SystemDiffAssignCut*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemDiffAssignCut* =
((([[*DiffAssign vid1* ($*f fid1 empty*)]] (*Geq* (*DiffVar vid1*) (*Const 0*)))
↔ (*Geq* ($*f fid1 empty*) (*Const 0*)))

definition *SystemCEFml1*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemCEFml1* = *Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))

definition *SystemCEFml2*::(′*sf*,′*sc*,′*sz*) *formula*
where *SystemCEFml2* = *Geq* (*DiffVar vid1*) (*Const 0*)

definition *CQ1Concl*::(′*sf*,′*sc*,′*sz*) *formula*
where *CQ1Concl* = (*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)) ↔ *Geq* (*DiffVar vid1*) (*Differential* (*Const 0*)))

definition *CQ2Concl*::(′*sf*,′*sc*,′*sz*) *formula*
where *CQ2Concl* = (*Geq* (*DiffVar vid1*) (*Differential* (*Const 0*)) ↔ *Geq* ($′ *vid1*) (*Const 0*))

definition *CEReq*::(′*sf*,′*sc*,′*sz*) *formula*

where *CEReq* = (*Geq* (*Differential* (*trm.Var vid1*)) (*Differential* (*Const 0*)) ↔ *Geq* ($′ *vid1*) (*Const 0*))

definition *CQRightSubst*::(′*sf*,′*sc*,′*sz*) *subst*
where *CQRightSubst* =
⦇ *SFunctions* = (λ-. *None*),
SPredicates = (λ*p*. (*if p* = *vid1 then* (*Some* (*Geq* (*DiffVar vid1*) (*Function* (*Inr vid1*) *empty*))) *else None*)),
SContexts = (λ-. *None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⦈

definition *CQLeftSubst*::(′*sf*,′*sc*,′*sz*) *subst*
where *CQLeftSubst* =
⦇ *SFunctions* = (λ-. *None*),
SPredicates = (λ*p*. (*if p* = *vid1 then* (*Some* (*Geq* (*Function* (*Inr vid1*) *empty*) (*Differential* (*Const 0*)))) *else None*)),
SContexts = (λ-. *None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⦈

definition *CEProof*::(′*sf*,′*sc*,′*sz*) *pf*
where *CEProof* = (([],[*CEReq*]), [
(*0*, *Cut CQ1Concl*)
,(*0*, *Cut CQ2Concl*)
,(*1*, *Rrule CohideRR 0*)
,(*Suc* (*Suc 0*), *CQ* (*Differential* (*Const 0*)) (*Const 0*) *CQRightSubst*)
,(*1*, *Rrule CohideRR 0*)
,(*1*, *CQ* (*Differential* (*Var vid1*)) (*DiffVar vid1*) *CQLeftSubst*)
,(*0*, *Rrule EquivR 0*)
,(*0*, *Lrule EquivForwardL 1*)
,(*Suc* (*Suc 0*), *Lrule EquivForwardL 1*)
,(*Suc* (*Suc* (*Suc 0*)), *CloseId 0 0*)
,(*Suc* (*Suc 0*), *CloseId 0 0*)
,(*Suc 0*, *CloseId 0 0*)
,(*0*, *Lrule EquivBackwardL* (*Suc* (*Suc 0*)))
,(*0*, *CloseId 0 0*)
,(*0*, *Lrule EquivBackwardL* (*Suc 0*))
,(*0*, *CloseId 0 0*)
,(*0*, *CloseId 0 0*)
])

lemma *CE-result-correct*:*proof-result CEProof* = ([],([],[*CEReq*]))
⟨*proof*⟩

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

where *DiffConstSubst* = ⦇
  SFunctions = (λ*f*. (*if f* = *fid1 then* (*Some* (*Const 0*)) *else None*)),
  SPredicates = (λ-. *None*),
  SContexts = (λ-. *None*),
  SPrograms = (λ-. *None*),
  SODEs = (λ-. *None*)
⦈

definition *DiffConstProof*::('*sf*,'*sc*,'*sz*) *pf*
where *DiffConstProof* = (([],[(*Equals* (*Differential* (*Const 0*)) (*Const 0*))]), [
(*0*, *AxSubst AdConst DiffConstSubst*)])

lemma *diffconst-result-correct*:*proof-result DiffConstProof* = ([], ([],[*Equals* (*Differential* (*Const 0*)) (*Const 0*)]))
⟨*proof*⟩

lemma *diffconst-sound-lemma*:*sound* (*proof-result DiffConstProof*)
⟨*proof*⟩

lemma *valid-of-sound*:*sound* ([], ([],[φ])) ⟹ *valid* φ
⟨*proof*⟩

lemma *almost-diff-const-sound*:*sound* ([], ([], [*Equals* (*Differential* (*Const 0*)) (*Const 0*)]))
⟨*proof*⟩

lemma *almost-diff-const*:*valid* (*Equals* (*Differential* (*Const 0*)) (*Const 0*))
⟨*proof*⟩
lemma *almost-diff-var*:*valid* (*Equals* (*Differential* (*trm.Var vid1*)) ($' *vid1*))
⟨*proof*⟩

lemma *CESound-lemma*:*sound* (*proof-result CEProof*)
⟨*proof*⟩

lemma *sound-to-valid*:*sound* ([], ([], [φ])) ⟹ *valid* φ
⟨*proof*⟩

lemma *CE1pre*:*sound* ([], ([], [*CEReq*]))
⟨*proof*⟩

lemma *CE1pre-valid*:*valid CEReq*
⟨*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*)))))
⟨*proof*⟩

definition *SystemDISubst*::('sf,'sc,'sz) *subst*
where *SystemDISubst* =

```
  (| SFunctions = (λf.
    (      if f = fid1 then Some(Function (Inr vid1) empty)
     else if f = fid2 then Some(Const 0)
     else None)),
    SPredicates = (λp. if p = vid1 then Some TT else None),
    SContexts = (λ-. None),
    SPrograms = (λ-. None),
    SODEs = (λc. if c = vid1 then Some (OProd (OSing vid1 (f0 fid1)) (OSing
vid2 (trm.Var vid1))) else None)
  |)
```

definition *SystemDCSubst*::('sf,'sc,'sz) *subst*
where *SystemDCSubst* =

```
  (| SFunctions = (λ
  f. None),
    SPredicates = (λp.  None),
    SContexts = (λC.
    if C = pid1 then
      Some TT
    else if C = pid2 then
      Some (Geq (Differential (Var vid1)) (Differential (Const 0)))
    else if C = pid3 then
      Some (Geq (Function fid1 empty) (Const 0))
    else
     None),
    SPrograms = (λ-. None),
     SODEs = (λc. 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 = (λf.  None),
     SPredicates = (λp. if p = vid1 then Some (Geq (Function (Inl fid1) empty)
(Const 0)) else None),
    SContexts = (λ-. None),
    SPrograms = (λa. if a = vid1 then
      Some (EvolveODE (OProd
                        (OSing vid1 (Function fid1 empty))
                        (OSing vid2 (Var vid1)))
                      (And TT (Geq (Function fid1 empty) (Const 0))))
                      else None),
    SODEs = (λ-. None)
  |)
```

definition *SystemVSubst2*::('sf,'sc,'sz) *subst*
where *SystemVSubst2* =
⦇ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ*p*. *if p* = *vid1 then Some* (*Geq* (*Function* (*Inl fid1*) *empty*) (*Const 0*)) *else None*),
SContexts = (λ-. *None*),
SPrograms = (λ*a*. *if a* = *vid1 then*
Some (*EvolveODE* (*OProd*
(*OSing vid1* (*Function fid1 empty*))
(*OSing vid2* (*Var vid1*)))
TT)
else None),
SODEs = (λ-. *None*)
⦈

definition *SystemDESubst*::('sf,'sc,'sz) *subst*
where *SystemDESubst* =
⦇ *SFunctions* = (λ*f*. *if f* = *fid1 then Some*(*Function* (*Inl fid1*) *empty*) *else None*),
SPredicates = (λ*p*. *if p* = *vid2 then Some*(*And TT* (*Geq* (*Function* (*Inl fid1*) *empty*) (*Const 0*))) *else None*),
SContexts = (λ*C*. *if C* = *pid1 then Some*(*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))) *else None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⦈

lemma *systemdesubst-correct*:∃ *ODE*.((([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]] ((*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*))))) ↔
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*))) (*And TT* (*Geq* (*f0 fid1*) (*Const 0*)))]]
[[*DiffAssign vid1* (*f0 fid1*)]](*Geq* (*Differential* (*Var vid1*)) (*Differential* (*Const 0*)))))
= *Fsubst* (((([[*EvolveODE* (*OProd* (*OSing vid1* (*f1 fid1 vid1*)) *ODE*) (*p1 vid2 vid1*)]] (*P pid1*)) ↔
([[*EvolveODE* ((*OProd* (*OSing vid1* (*f1 fid1 vid1*))) *ODE*) (*p1 vid2 vid1*)]]
[[*DiffAssign vid1* (*f1 fid1 vid1*)]]*P pid1*))) *SystemDESubst*
⟨*proof*⟩
definition *SystemKSubst*::('sf,'sc,'sz) *subst*
where *SystemKSubst* = ⦇ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*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*),
SPrograms = (λ*c*. *if c* = *vid1 then Some* (*EvolveODE* (*OProd* (*OSing vid1*

(*Function fid1 empty*)) (*OSing vid2* (*Var vid1*))) (*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*)))) *else None*),
SODEs = (λ-. *None*)
⦈

lemma *subst-imp-simp*:*Fsubst* (*Implies p q*) σ = (*Implies* (*Fsubst p* σ) (*Fsubst q* σ))
⟨*proof*⟩

lemma *subst-equiv-simp*:*Fsubst* (*Equiv p q*) σ = (*Equiv* (*Fsubst p* σ) (*Fsubst q* σ))
⟨*proof*⟩

lemma *subst-box-simp*:*Fsubst* (*Box p q*) σ = (*Box* (*Psubst p* σ) (*Fsubst q* σ))
⟨*proof*⟩

lemma *pfsubst-box-simp*:*PFsubst* (*Box p q*) σ = (*Box* (*PPsubst p* σ) (*PFsubst q* σ))
⟨*proof*⟩

lemma *pfsubst-imp-simp*:*PFsubst* (*Implies p q*) σ = (*Implies* (*PFsubst p* σ) (*PFsubst q* σ))
⟨*proof*⟩

definition *SystemDWSubst*::(′*sf*,′*sc*,′*sz*) *subst*
where *SystemDWSubst* = ⦇ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. *if C* = *pid1 then Some* (*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*))) *else None*),
SPrograms = (λ-. *None*),
SODEs = (λ*c*. *if c* = *vid1 then Some* (*OProd* (*OSing vid1* (*Function fid1 empty*)) (*OSing vid2* (*Var vid1*))) *else None*)
⦈

definition *SystemCESubst*::(′*sf*,′*sc*,′*sz*) *subst*
where *SystemCESubst* = ⦇ *SFunctions* = (λ*f*. *None*),
SPredicates = (λ-. *None*),
SContexts = (λ*C*. *if C* = *pid1 then Some*(*Implies*(*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*))) ([[*DiffAssign vid1* (*Function fid1 empty*)]](*Predicational* (*Inr* ()))))) *else None*),
SPrograms = (λ-. *None*),
SODEs = (λ-. *None*)
⦈

lemma *SystemCESubstOK*:
step-ok
([([],[*Equiv* (*Implies*(*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*))) ([[*DiffAssign vid1* (*Function fid1 empty*)]](*SystemCEFml1*)))
(*Implies*(*And* (*Geq* (*Const 0*) (*Const 0*)) (*Geq* (*Function fid1 empty*) (*Const 0*))) ([[*DiffAssign vid1* (*Function fid1 empty*)]]((*SystemCEFml2*))))

])],
([],[]))

0
(CE SystemCEFml1 SystemCEFml2 SystemCESubst)
⟨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)
~~,(0, CloseId 0 0)~~
,(Suc 0, CloseId 1 0)
~~,(0, Rrule AndR 0)~~
,(0, Rrule ImplyR 0)
,(0, Cut SystemDCCut)
,(0, Lrule ImplyL 0)
,(0, Rrule CohideRR 0)
,(0, Lrule EquivBackwardL 0)
,(0, Rrule CohideR 0)
,(0, AxSubst ADC SystemDCSubst) — 17
,(0, CloseId 0 0)
,(0, Rrule CohideRR 0)
,(0, Cut SystemVCut)
,(0, Lrule ImplyL 0)
,(0, Rrule CohideRR 0)
,(0, Cut SystemDECut)
,(0, Lrule EquivBackwardL 0)
,(0, Rrule CohideRR 0)
,(1, CloseId (Suc 1) 0) — Last step
,(Suc 1, CloseId 0 0)

,(*1*, *AxSubst AV SystemVSubst*) — 28
,(*0*, *Cut SystemVCut2*)

,(*0*, *Lrule ImplyL 0*)
,(*0*, *Rrule CohideRR 0*)
,(*Suc 1*, *CloseId 0 0*)
,(*Suc 1*, *CloseId* (*Suc 2*) *0*)

,(*Suc 1*, *AxSubst AV SystemVSubst2*) — 34
,(*0*, *Rrule CohideRR 0*)
,(*0*, *DEAxiomSchema* (*OSing vid2* (*trm.Var vid1*)) *SystemDESubst*) — 36
,(*0*, *Cut SystemKCut*)
,(*0*, *Lrule ImplyL 0*)
,(*0*, *Rrule CohideRR 0*)
,(*0*, *Lrule ImplyL 0*)
,(*0*, *Rrule CohideRR 0*)
,(*0*, *AxSubst AK SystemKSubst*) — 42
,(*0*, *CloseId 0 0*)
,(*0*, *Rrule CohideR 0*)
,(*1*, *AxSubst ADW SystemDWSubst*) — 45
,(*0*, *G*)
,(*0*, *Cut SystemEquivCut*)
,(*0*, *Lrule EquivBackwardL 0*)
,(*0*, *Rrule CohideR 0*)
,(*0*, *CloseId 0 0*)
,(*0*, *Rrule CohideR 0*)
,(*0*, *CE SystemCEFml1 SystemCEFml2 SystemCESubst*) — 52
,(*0*, *Rrule ImplyR 0*)
,(*0*, *Lrule AndL 0*)
,(*0*, *Cut SystemDiffAssignCut*)
,(*0*, *Lrule EquivBackwardL 0*)
,(*0*, *Rrule CohideRR 0*)
,(*0*, *CloseId 0 0*)
,(*0*, *CloseId 1 0*)
,(*0*, *AxSubst Adassign SystemDiffAssignSubst*) — 60
])

lemma *system-result-correct*:*proof-result SystemProof* =
([],
([],[*Implies* (*And* (*Geq* (*Var vid1*) (*Const 0*)) (*Geq* (*f0 fid1*) (*Const 0*)))
([[*EvolveODE* (*OProd* (*OSing vid1* (*f0 fid1*)) (*OSing vid2* (*Var vid1*)))
(*TT*)]]*Geq* (*Var vid1*) (*Const 0*))]]))
⟨*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] R. Bohrer, V. Rahli, I. Vukotic, M. Völp, and A. Platzer. Formally verified differential dynamic logic. In Y. Bertot and V. Vafeiadis, editors, *Certified Programs and Proofs - 6th ACM SIGPLAN Conference, CPP 2017, Paris, France, January 16-17, 2017*, pages 208–221. ACM, 2017.

[2] A. Platzer. A uniform substitution calculus for differential dynamic logic. In A. P. Felty and A. Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 467–481. Springer, 2015.

[3] A. Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, 2016.