A Formal Model of Extended Finite State Machine

Michael Foster Ramsay G. Taylor Achim D. Brucker
John Derrick

June 3, 2019

Department of Computer Science
The University of Sheffield
Sheffield, UK
{jmafoster1, a.brucker, r.g.taylor, j.derrick }@sheffield.ac.uk

In this article, we formalize

Keywords:

Contents

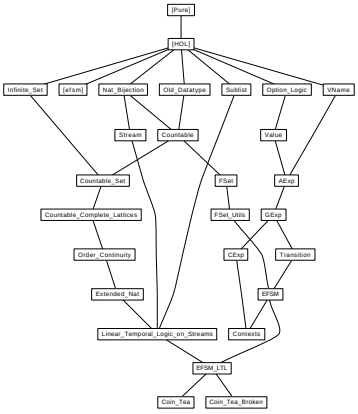

Figure 1: The Dependency Graph of the Isabelle Theories.

1 Introduction

[1]

2 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory *Old_Datatype*
imports *Main*
begin

2.1 The datatype universe

definition *"Node = {p. ∃f x k. p = (f :: nat => 'b + nat, x ::'a + nat) ∧ f k = Inr 0}"*

typedef *('a, 'b) node = "Node :: ((nat => 'b + nat) * ('a + nat)) set"*
  morphisms *Rep_Node Abs_Node*
  unfolding *Node_def* **by** *auto*

Datatypes will be represented by sets of type *node*

type_synonym *'a item = "('a, unit) node set"*
type_synonym *('a, 'b) dtree = "('a, 'b) node set"*

definition *Push :: "[('b + nat), nat => ('b + nat)] => (nat => ('b + nat))"*

  where *"Push == (%b h. case_nat b h)"*

definition *Push_Node :: "[('b + nat), ('a, 'b) node] => ('a, 'b) node"*
  where *"Push_Node == (%n x. Abs_Node (apfst (Push n) (Rep_Node x)))"*

definition *Atom :: "('a + nat) => ('a, 'b) dtree"*
  where *"Atom == (%x. {Abs_Node((%k. Inr 0, x))})"*
definition *Scons :: "[('a, 'b) dtree, ('a, 'b) dtree] => ('a, 'b) dtree"*
  where *"Scons M N == (Push_Node (Inr 1) ‘ M) Un (Push_Node (Inr (Suc 1)) ‘ N)"*

definition *Leaf :: "'a => ('a, 'b) dtree"*
  where *"Leaf == Atom ∘ Inl"*
definition *Numb :: "nat => ('a, 'b) dtree"*
  where *"Numb == Atom ∘ Inr"*

definition *In0 :: "('a, 'b) dtree => ('a, 'b) dtree"*
  where *"In0(M) == Scons (Numb 0) M"*
definition *In1 :: "('a, 'b) dtree => ('a, 'b) dtree"*
  where *"In1(M) == Scons (Numb 1) M"*

definition *Lim :: "('b => ('a, 'b) dtree) => ('a, 'b) dtree"*
  where *"Lim f == ⋃{z. ∃x. z = Push_Node (Inl x) ‘ (f x)}"*

definition *ndepth :: "('a, 'b) node => nat"*
  where *"ndepth(n) == (%(f,x). LEAST k. f k = Inr 0) (Rep_Node n)"*
definition *ntrunc :: "[nat, ('a, 'b) dtree] => ('a, 'b) dtree"*
  where *"ntrunc k N == {n. n∈N ∧ ndepth(n)<k}"*

definition *uprod :: "[('a, 'b) dtree set, ('a, 'b) dtree set]=> ('a, 'b) dtree set"*
  where *"uprod A B == UN x:A. UN y:B. { Scons x y }"*

definition *usum :: "[('a, 'b) dtree set, ('a, 'b) dtree set]=> ('a, 'b) dtree set"*
  where *"usum A B == In0`A Un In1`B"*

definition *Split :: "[[('a, 'b) dtree, ('a, 'b) dtree]=>'c, ('a, 'b) dtree] => 'c"*
  where *"Split c M == THE u. ∃x y. M = Scons x y ∧ u = c x y"*

definition *Case :: "[[('a, 'b) dtree]=>'c, [('a, 'b) dtree]=>'c, ('a, 'b) dtree] => 'c"*
  where *"Case c d M == THE u. (∃x . M = In0(x) ∧ u = c(x)) ∨ (∃y . M = In1(y) ∧ u = d(y))"*

definition *dprod :: "[(('a, 'b) dtree * ('a, 'b) dtree)set, (('a, 'b) dtree * ('a, 'b) dtree)set]*
*=> (('a, 'b) dtree * ('a, 'b) dtree)set"*
  where *"dprod r s == UN (x,x'):r. UN (y,y'):s. {(Scons x y, Scons x' y')}"*

definition *dsum :: "[(('a, 'b) dtree * ('a, 'b) dtree)set, (('a, 'b) dtree * ('a, 'b) dtree)set]*
*=> (('a, 'b) dtree * ('a, 'b) dtree)set"*
  where *"dsum r s == (UN (x,x'):r. {(In0(x),In0(x'))}) Un (UN (y,y'):s. {(In1(y),In1(y'))})"*

lemma *apfst_convE:*
"[| q = apfst f p; !!x y. [| p = (x,y); q = (f(x),y) |] ==> R
|] ==> R"
by *(force simp add: apfst_def)*

lemma *Push_inject1: "Push i f = Push j g ==> i=j"*
apply *(simp add: Push_def fun_eq_iff)*
apply *(drule_tac x=0* **in** *spec, simp)*
done

lemma *Push_inject2: "Push i f = Push j g ==> f=g"*
apply *(auto simp add: Push_def fun_eq_iff)*
apply *(drule_tac x="Suc x"* **in** *spec, simp)*
done

lemma *Push_inject:*
"[| Push i f =Push j g; [| i=j; f=g |] ==> P |] ==> P"
by *(blast dest: Push_inject1 Push_inject2)*

lemma *Push_neq_K0: "Push (Inr (Suc k)) f = (%z. Inr 0) ==> P"*
by *(auto simp add: Push_def fun_eq_iff split: nat.split_asm)*

lemmas *Abs_Node_inj = Abs_Node_inject [THEN [2] rev_iffD1]*

lemma *Node_K0_I: "(λk. Inr 0, a) ∈ Node"*
by *(simp add: Node_def)*

lemma *Node_Push_I: "p ∈ Node ⟹ apfst (Push i) p ∈ Node"*
apply *(simp add: Node_def Push_def)*
apply *(fast intro!: apfst_conv nat.case(2)[THEN trans])*
done

2.2 Freeness: Distinctness of Constructors

lemma *Scons_not_Atom [iff]: "Scons M N ≠ Atom(a)"*
unfolding *Atom_def Scons_def Push_Node_def One_nat_def*
by *(blast intro: Node_K0_I Rep_Node [THEN Node_Push_I]*
dest!: Abs_Node_inj
elim!: apfst_convE sym [THEN Push_neq_K0])

lemmas *Atom_not_Scons [iff] = Scons_not_Atom [THEN not_sym]*

lemma *inj_Atom: "inj(Atom)"*
apply *(simp add: Atom_def)*
apply *(blast intro!: inj_onI Node_K0_I dest!: Abs_Node_inj)*
done
lemmas *Atom_inject = inj_Atom [THEN injD]*

lemma *Atom_Atom_eq [iff]: "(Atom(a)=Atom(b)) = (a=b)"*
by *(blast dest!: Atom_inject)*

lemma *inj_Leaf: "inj(Leaf)"*
apply *(simp add: Leaf_def o_def)*
apply *(rule inj_onI)*
apply *(erule Atom_inject [THEN Inl_inject])*
done

lemmas *Leaf_inject [dest!] = inj_Leaf [THEN injD]*

lemma *inj_Numb: "inj(Numb)"*
apply *(simp add: Numb_def o_def)*
apply *(rule inj_onI)*
apply *(erule Atom_inject [THEN Inr_inject])*
done

lemmas *Numb_inject [dest!] = inj_Numb [THEN injD]*

lemma *Push_Node_inject:*
"[| Push_Node i m =Push_Node j n; [| i=j; m=n |] ==> P
|] ==> P"
apply *(simp add: Push_Node_def)*
apply *(erule Abs_Node_inj [THEN apfst_convE])*
apply *(rule Rep_Node [THEN Node_Push_I])+*
apply *(erule sym [THEN apfst_convE])*
apply *(blast intro: Rep_Node_inject [THEN iffD1] trans sym elim!: Push_inject)*
done

lemma *Scons_inject_lemma1: "Scons M N <= Scons M' N' ==> M<=M'"*
unfolding *Scons_def One_nat_def*
by *(blast dest!: Push_Node_inject)*

lemma *Scons_inject_lemma2: "Scons M N <= Scons M' N' ==> N<=N'"*

unfolding *Scons_def One_nat_def*
by *(blast dest!: Push_Node_inject)*

lemma *Scons_inject1: "Scons M N = Scons M' N' ==> M=M'"*
apply *(erule equalityE)*
apply *(iprover intro: equalityI Scons_inject_lemma1)*
done

lemma *Scons_inject2: "Scons M N = Scons M' N' ==> N=N'"*
apply *(erule equalityE)*
apply *(iprover intro: equalityI Scons_inject_lemma2)*
done

lemma *Scons_inject:*
"[| Scons M N = Scons M' N'; [| M=M'; N=N' |] ==> P |] ==> P"
by *(iprover dest: Scons_inject1 Scons_inject2)*

lemma *Scons_Scons_eq [iff]: "(Scons M N = Scons M' N') = (M=M' ∧ N=N')"*
by *(blast elim!: Scons_inject)*

lemma *Scons_not_Leaf [iff]: "Scons M N ≠ Leaf(a)"*
unfolding *Leaf_def o_def* **by** *(rule Scons_not_Atom)*

lemmas *Leaf_not_Scons [iff] = Scons_not_Leaf [THEN not_sym]*

lemma *Scons_not_Numb [iff]: "Scons M N ≠ Numb(k)"*
unfolding *Numb_def o_def* **by** *(rule Scons_not_Atom)*

lemmas *Numb_not_Scons [iff] = Scons_not_Numb [THEN not_sym]*

lemma *Leaf_not_Numb [iff]: "Leaf(a) ≠ Numb(k)"*
by *(simp add: Leaf_def Numb_def)*

lemmas *Numb_not_Leaf [iff] = Leaf_not_Numb [THEN not_sym]*

lemma *ndepth_K0: "ndepth (Abs_Node(%k. Inr 0, x)) = 0"*
by *(simp add: ndepth_def Node_K0_I [THEN Abs_Node_inverse] Least_equality)*

lemma *ndepth_Push_Node_aux:*
"case_nat (Inr (Suc i)) f k = Inr 0 ⟶ Suc(LEAST x. f x = Inr 0) ≤ k"
apply *(induct_tac "k", auto)*
apply *(erule Least_le)*
done

lemma *ndepth_Push_Node:*
"ndepth (Push_Node (Inr (Suc i)) n) = Suc(ndepth(n))"
apply *(insert Rep_Node [of n, unfolded Node_def])*
apply *(auto simp add: ndepth_def Push_Node_def*
Rep_Node [THEN Node_Push_I, THEN Abs_Node_inverse])

apply *(rule Least_equality)*
apply *(auto simp add: Push_def ndepth_Push_Node_aux)*
apply *(erule LeastI)*
done

lemma *ntrunc_0 [simp]: "ntrunc 0 M = {}"*
by *(simp add: ntrunc_def)*

lemma *ntrunc_Atom [simp]: "ntrunc (Suc k) (Atom a) = Atom(a)"*
by *(auto simp add: Atom_def ntrunc_def ndepth_K0)*

lemma *ntrunc_Leaf [simp]: "ntrunc (Suc k) (Leaf a) = Leaf(a)"*
unfolding *Leaf_def o_def* **by** *(rule ntrunc_Atom)*

lemma *ntrunc_Numb [simp]: "ntrunc (Suc k) (Numb i) = Numb(i)"*
unfolding *Numb_def o_def* **by** *(rule ntrunc_Atom)*

lemma *ntrunc_Scons [simp]:*
"ntrunc (Suc k) (Scons M N) = Scons (ntrunc k M) (ntrunc k N)"
unfolding *Scons_def ntrunc_def One_nat_def*
by *(auto simp add: ndepth_Push_Node)*

lemma *ntrunc_one_In0 [simp]: "ntrunc (Suc 0) (In0 M) = {}"*
apply *(simp add: In0_def)*
apply *(simp add: Scons_def)*
done

lemma *ntrunc_In0 [simp]: "ntrunc (Suc(Suc k)) (In0 M) = In0 (ntrunc (Suc k) M)"*
by *(simp add: In0_def)*

lemma *ntrunc_one_In1 [simp]: "ntrunc (Suc 0) (In1 M) = {}"*
apply *(simp add: In1_def)*
apply *(simp add: Scons_def)*
done

lemma *ntrunc_In1 [simp]: "ntrunc (Suc(Suc k)) (In1 M) = In1 (ntrunc (Suc k) M)"*
by *(simp add: In1_def)*

2.3 Set Constructions

lemma *uprodI [intro!]: "⟦M∈A; N∈B⟧ ⟹ Scons M N ∈ uprod A B"*
by *(simp add: uprod_def)*

lemma *uprodE [elim!]:*
"⟦c ∈ uprod A B;
⋀x y. ⟦x ∈ A; y ∈ B; c = Scons x y⟧ ⟹ P
⟧ ⟹ P"
by *(auto simp add: uprod_def)*

lemma *uprodE2: "⟦Scons M N ∈ uprod A B; ⟦M ∈ A; N ∈ B⟧ ⟹ P⟧ ⟹ P"*
by *(auto simp add: uprod_def)*

lemma *usum_In0I [intro]: "M ∈ A ⟹ In0(M) ∈ usum A B"*
by *(simp add: usum_def)*

lemma *usum_In1I [intro]: "N ∈ B ⟹ In1(N) ∈ usum A B"*
by *(simp add: usum_def)*

lemma *usumE [elim!]:*
"⟦u ∈ usum A B;
⋀x. ⟦x ∈ A; u=In0(x)⟧ ⟹ P;
⋀y. ⟦y ∈ B; u=In1(y)⟧ ⟹ P
⟧ ⟹ P"
by *(auto simp add: usum_def)*

lemma *In0_not_In1 [iff]: "In0(M) ≠ In1(N)"*
unfolding *In0_def In1_def One_nat_def* **by** *auto*

lemmas *In1_not_In0 [iff] = In0_not_In1 [THEN not_sym]*

lemma *In0_inject: "In0(M) = In0(N) ==> M=N"*
by *(simp add: In0_def)*

lemma *In1_inject: "In1(M) = In1(N) ==> M=N"*
by *(simp add: In1_def)*

lemma *In0_eq [iff]: "(In0 M = In0 N) = (M=N)"*
by *(blast dest!: In0_inject)*

lemma *In1_eq [iff]: "(In1 M = In1 N) = (M=N)"*
by *(blast dest!: In1_inject)*

lemma *inj_In0: "inj In0"*
by *(blast intro!: inj_onI)*

lemma *inj_In1: "inj In1"*
by *(blast intro!: inj_onI)*

lemma *Lim_inject: "Lim f = Lim g ==> f = g"*
apply *(simp add: Lim_def)*
apply *(rule ext)*
apply *(blast elim!: Push_Node_inject)*
done

lemma *ntrunc_subsetI: "ntrunc k M <= M"*
by *(auto simp add: ntrunc_def)*

lemma *ntrunc_subsetD: "(!!k. ntrunc k M <= N) ==> M<=N"*
by *(auto simp add: ntrunc_def)*

lemma *ntrunc_equality: "(!!k. ntrunc k M = ntrunc k N) ==> M=N"*
apply *(rule equalityI)*
apply *(rule_tac [!] ntrunc_subsetD)*
apply *(rule_tac [!] ntrunc_subsetI [THEN [2] subset_trans], auto)*
done

lemma *ntrunc_o_equality:*
"[| !!k. (ntrunc(k) ∘ h1) = (ntrunc(k) ∘ h2) |] ==> h1=h2"
apply *(rule ntrunc_equality [THEN ext])*
apply *(simp add: fun_eq_iff)*
done

lemma *uprod_mono: "[| A<=A'; B<=B' |] ==> uprod A B <= uprod A' B'"*
by *(simp add: uprod_def, blast)*

lemma *usum_mono: "[| A<=A'; B<=B' |] ==> usum A B <= usum A' B'"*
by *(simp add: usum_def, blast)*

lemma *Scons_mono: "[| M<=M'; N<=N' |] ==> Scons M N <= Scons M' N'"*
by *(simp add: Scons_def, blast)*

lemma *In0_mono: "M<=N ==> In0(M) <= In0(N)"*
by *(simp add: In0_def Scons_mono)*

lemma *In1_mono: "M<=N ==> In1(M) <= In1(N)"*
by *(simp add: In1_def Scons_mono)*

lemma *Split [simp]: "Split c (Scons M N) = c M N"*
by *(simp add: Split_def)*

lemma *Case_In0 [simp]: "Case c d (In0 M) = c(M)"*
by *(simp add: Case_def)*

lemma *Case_In1 [simp]: "Case c d (In1 N) = d(N)"*
by *(simp add: Case_def)*

lemma *ntrunc_UN1: "ntrunc k (UN x. f(x)) = (UN x. ntrunc k (f x))"*
by *(simp add: ntrunc_def, blast)*

lemma *Scons_UN1_x: "Scons (UN x. f x) M = (UN x. Scons (f x) M)"*
by *(simp add: Scons_def, blast)*

lemma *Scons_UN1_y: "Scons M (UN x. f x) = (UN x. Scons M (f x))"*
by *(simp add: Scons_def, blast)*

lemma *In0_UN1: "In0(UN x. f(x)) = (UN x. In0(f(x)))"*
by *(simp add: In0_def Scons_UN1_y)*

lemma *In1_UN1: "In1(UN x. f(x)) = (UN x. In1(f(x)))"*
by *(simp add: In1_def Scons_UN1_y)*

lemma *dprodI [intro!]:*
"⟦(M,M') ∈ r; (N,N') ∈ s⟧ ⟹ (Scons M N, Scons M' N') ∈ dprod r s"
by *(auto simp add: dprod_def)*

lemma *dprodE [elim!]:*
"⟦c ∈ dprod r s;
⋀x y x' y'. ⟦(x,x') ∈ r; (y,y') ∈ s;
c = (Scons x y, Scons x' y')⟧ ⟹ P
⟧ ⟹ P"
by *(auto simp add: dprod_def)*

lemma *dsum_In0I [intro]: "(M,M') ∈ r ⟹ (In0(M), In0(M')) ∈ dsum r s"*
by *(auto simp add: dsum_def)*

lemma *dsum_In1I [intro]: "(N,N') ∈ s ⟹ (In1(N), In1(N')) ∈ dsum r s"*
by *(auto simp add: dsum_def)*

lemma *dsumE [elim!]:*
"⟦w ∈ dsum r s;
⋀x x'. ⟦ (x,x') ∈ r; w = (In0(x), In0(x')) ⟧ ⟹ P;
⋀y y'. ⟦ (y,y') ∈ s; w = (In1(y), In1(y')) ⟧ ⟹ P
⟧ ⟹ P"
by *(auto simp add: dsum_def)*

lemma *dprod_mono: "[| r<=r'; s<=s' |] ==> dprod r s <= dprod r' s'"*
by *blast*

lemma *dsum_mono: "[| r<=r'; s<=s' |] ==> dsum r s <= dsum r' s'"*
by *blast*

lemma *dprod_Sigma: "(dprod (A × B) (C × D)) <= (uprod A C) × (uprod B D)"*
by *blast*

lemmas *dprod_subset_Sigma = subset_trans [OF dprod_mono dprod_Sigma]*

lemma *dprod_subset_Sigma2:*
"(dprod (Sigma A B) (Sigma C D)) <= Sigma (uprod A C) (Split (%x y. uprod (B x) (D y)))"
by *auto*

lemma *dsum_Sigma: "(dsum (A × B) (C × D)) <= (usum A C) × (usum B D)"*
by *blast*

lemmas *dsum_subset_Sigma = subset_trans [OF dsum_mono dsum_Sigma]*

lemma *Domain_dprod [simp]: "Domain (dprod r s) = uprod (Domain r) (Domain s)"*
  by *auto*

lemma *Domain_dsum [simp]: "Domain (dsum r s) = usum (Domain r) (Domain s)"*
  by *auto*

hides popular names

hide_type (**open**) *node item*
hide_const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

ML_file *"~~/src/HOL/Tools/Old_Datatype/old_datatype.ML"*

end

3 Bijections between natural numbers and other types

theory *Nat_Bijection*
  imports *Main*
begin

3.1 Type *nat* × *nat*

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition *triangle :: "nat ⇒ nat"*
  where *"triangle n = (n * Suc n) div 2"*

lemma *triangle_0 [simp]: "triangle 0 = 0"*
  by *(simp add: triangle_def)*

lemma *triangle_Suc [simp]: "triangle (Suc n) = triangle n + Suc n"*
  by *(simp add: triangle_def)*

definition *prod_encode :: "nat × nat ⇒ nat"*
  where *"prod_encode = (λ(m, n). triangle (m + n) + m)"*

In this auxiliary function, *triangle k + m* is an invariant.

fun *prod_decode_aux :: "nat ⇒ nat ⇒ nat × nat"*
  where *"prod_decode_aux k m =*
    (if m ≤ k then (m, k - m) else prod_decode_aux (Suc k) (m - Suc k))"

declare *prod_decode_aux.simps [simp del]*

definition *prod_decode :: "nat ⇒ nat × nat"*
  where *"prod_decode = prod_decode_aux 0"*

lemma *prod_encode_prod_decode_aux: "prod_encode (prod_decode_aux k m) = triangle k + m"*
  apply *(induct k m rule: prod_decode_aux.induct)*
  apply *(subst prod_decode_aux.simps)*
  apply *(simp add: prod_encode_def)*
  done

lemma *prod_decode_inverse [simp]: "prod_encode (prod_decode n) = n"*
  by *(simp add: prod_decode_def prod_encode_prod_decode_aux)*

lemma *prod_decode_triangle_add: "prod_decode (triangle k + m) = prod_decode_aux k m"*
  apply *(induct k arbitrary: m)*
   apply *(simp add: prod_decode_def)*
  apply *(simp only: triangle_Suc add.assoc)*

```
  apply (subst prod_decode_aux.simps)
  apply simp
  done

lemma prod_encode_inverse [simp]: "prod_decode (prod_encode x) = x"
  unfolding prod_encode_def
  apply (induct x)
  apply (simp add: prod_decode_triangle_add)
  apply (subst prod_decode_aux.simps)
  apply simp
  done

lemma inj_prod_encode: "inj_on prod_encode A"
  by (rule inj_on_inverseI) (rule prod_encode_inverse)

lemma inj_prod_decode: "inj_on prod_decode A"
  by (rule inj_on_inverseI) (rule prod_decode_inverse)

lemma surj_prod_encode: "surj prod_encode"
  by (rule surjI) (rule prod_decode_inverse)

lemma surj_prod_decode: "surj prod_decode"
  by (rule surjI) (rule prod_encode_inverse)

lemma bij_prod_encode: "bij prod_encode"
  by (rule bijI [OF inj_prod_encode surj_prod_encode])

lemma bij_prod_decode: "bij prod_decode"
  by (rule bijI [OF inj_prod_decode surj_prod_decode])

lemma prod_encode_eq: "prod_encode x = prod_encode y ⟷ x = y"
  by (rule inj_prod_encode [THEN inj_eq])

lemma prod_decode_eq: "prod_decode x = prod_decode y ⟷ x = y"
  by (rule inj_prod_decode [THEN inj_eq])
```

Ordering properties

```
lemma le_prod_encode_1: "a ≤ prod_encode (a, b)"
  by (simp add: prod_encode_def)

lemma le_prod_encode_2: "b ≤ prod_encode (a, b)"
  by (induct b) (simp_all add: prod_encode_def)
```

3.2 Type *nat + nat*

```
definition sum_encode :: "nat + nat ⇒ nat"
  where "sum_encode x = (case x of Inl a ⇒ 2 * a | Inr b ⇒ Suc (2 * b))"

definition sum_decode :: "nat ⇒ nat + nat"
  where "sum_decode n = (if even n then Inl (n div 2) else Inr (n div 2))"

lemma sum_encode_inverse [simp]: "sum_decode (sum_encode x) = x"
  by (induct x) (simp_all add: sum_decode_def sum_encode_def)

lemma sum_decode_inverse [simp]: "sum_encode (sum_decode n) = n"
  by (simp add: even_two_times_div_two sum_decode_def sum_encode_def)

lemma inj_sum_encode: "inj_on sum_encode A"
  by (rule inj_on_inverseI) (rule sum_encode_inverse)

lemma inj_sum_decode: "inj_on sum_decode A"
```

by *(rule inj_on_inverseI) (rule sum_decode_inverse)*

lemma *surj_sum_encode: "surj sum_encode"*
by *(rule surjI) (rule sum_decode_inverse)*

lemma *surj_sum_decode: "surj sum_decode"*
by *(rule surjI) (rule sum_encode_inverse)*

lemma *bij_sum_encode: "bij sum_encode"*
by *(rule bijI [OF inj_sum_encode surj_sum_encode])*

lemma *bij_sum_decode: "bij sum_decode"*
by *(rule bijI [OF inj_sum_decode surj_sum_decode])*

lemma *sum_encode_eq: "sum_encode x = sum_encode y ⟷ x = y"*
by *(rule inj_sum_encode [THEN inj_eq])*

lemma *sum_decode_eq: "sum_decode x = sum_decode y ⟷ x = y"*
by *(rule inj_sum_decode [THEN inj_eq])*

3.3 Type *int*

definition *int_encode :: "int ⇒ nat"*
where *"int_encode i = sum_encode (if 0 ≤ i then Inl (nat i) else Inr (nat (- i - 1)))"*

definition *int_decode :: "nat ⇒ int"*
where *"int_decode n = (case sum_decode n of Inl a ⇒ int a | Inr b ⇒ - int b - 1)"*

lemma *int_encode_inverse [simp]: "int_decode (int_encode x) = x"*
by *(simp add: int_decode_def int_encode_def)*

lemma *int_decode_inverse [simp]: "int_encode (int_decode n) = n"*
unfolding *int_decode_def int_encode_def*
using *sum_decode_inverse [of n]* **by** *(cases "sum_decode n") simp_all*

lemma *inj_int_encode: "inj_on int_encode A"*
by *(rule inj_on_inverseI) (rule int_encode_inverse)*

lemma *inj_int_decode: "inj_on int_decode A"*
by *(rule inj_on_inverseI) (rule int_decode_inverse)*

lemma *surj_int_encode: "surj int_encode"*
by *(rule surjI) (rule int_decode_inverse)*

lemma *surj_int_decode: "surj int_decode"*
by *(rule surjI) (rule int_encode_inverse)*

lemma *bij_int_encode: "bij int_encode"*
by *(rule bijI [OF inj_int_encode surj_int_encode])*

lemma *bij_int_decode: "bij int_decode"*
by *(rule bijI [OF inj_int_decode surj_int_decode])*

lemma *int_encode_eq: "int_encode x = int_encode y ⟷ x = y"*
by *(rule inj_int_encode [THEN inj_eq])*

lemma *int_decode_eq: "int_decode x = int_decode y ⟷ x = y"*
by *(rule inj_int_decode [THEN inj_eq])*

3.4 Type *nat list*

```
fun list_encode :: "nat list ⇒ nat"
  where
    "list_encode [] = 0"
  | "list_encode (x # xs) = Suc (prod_encode (x, list_encode xs))"

function list_decode :: "nat ⇒ nat list"
  where
    "list_decode 0 = []"
  | "list_decode (Suc n) = (case prod_decode n of (x, y) ⇒ x # list_decode y)"
  by pat_completeness auto

termination list_decode
  apply (relation "measure id")
   apply simp_all
  apply (drule arg_cong [where f="prod_encode"])
  apply (drule sym)
  apply (simp add: le_imp_less_Suc le_prod_encode_2)
  done

lemma list_encode_inverse [simp]: "list_decode (list_encode x) = x"
  by (induct x rule: list_encode.induct) simp_all

lemma list_decode_inverse [simp]: "list_encode (list_decode n) = n"
  apply (induct n rule: list_decode.induct)
   apply simp
  apply (simp split: prod.split)
  apply (simp add: prod_decode_eq [symmetric])
  done

lemma inj_list_encode: "inj_on list_encode A"
  by (rule inj_on_inverseI) (rule list_encode_inverse)

lemma inj_list_decode: "inj_on list_decode A"
  by (rule inj_on_inverseI) (rule list_decode_inverse)

lemma surj_list_encode: "surj list_encode"
  by (rule surjI) (rule list_decode_inverse)

lemma surj_list_decode: "surj list_decode"
  by (rule surjI) (rule list_encode_inverse)

lemma bij_list_encode: "bij list_encode"
  by (rule bijI [OF inj_list_encode surj_list_encode])

lemma bij_list_decode: "bij list_decode"
  by (rule bijI [OF inj_list_decode surj_list_decode])

lemma list_encode_eq: "list_encode x = list_encode y ⟷ x = y"
  by (rule inj_list_encode [THEN inj_eq])

lemma list_decode_eq: "list_decode x = list_decode y ⟷ x = y"
  by (rule inj_list_decode [THEN inj_eq])
```

3.5 Finite sets of naturals

3.5.1 Preliminaries

```
lemma finite_vimage_Suc_iff: "finite (Suc -` F) ⟷ finite F"
  apply (safe intro!: finite_vimageI inj_Suc)
```

```
  apply (rule finite_subset [where B="insert 0 (Suc ` Suc -` F)"])
   apply (rule subsetI)
   apply (case_tac x)
    apply simp
   apply simp
  apply (rule finite_insert [THEN iffD2])
  apply (erule finite_imageI)
  done

lemma vimage_Suc_insert_0: "Suc -` insert 0 A = Suc -` A"
  by auto

lemma vimage_Suc_insert_Suc: "Suc -` insert (Suc n) A = insert n (Suc -` A)"
  by auto

lemma div2_even_ext_nat:
  fixes x y :: nat
  assumes "x div 2 = y div 2"
    and "even x ⟷ even y"
  shows "x = y"
proof -
  from ⟨even x ⟷ even y⟩ have "x mod 2 = y mod 2"
    by (simp only: even_iff_mod_2_eq_zero) auto
  with assms have "x div 2 * 2 + x mod 2 = y div 2 * 2 + y mod 2"
    by simp
  then show ?thesis
    by simp
qed
```

3.5.2 From sets to naturals

```
definition set_encode :: "nat set ⇒ nat"
  where "set_encode = sum ((^) 2)"

lemma set_encode_empty [simp]: "set_encode {} = 0"
  by (simp add: set_encode_def)

lemma set_encode_inf: "¬ finite A ⟹ set_encode A = 0"
  by (simp add: set_encode_def)

lemma set_encode_insert [simp]: "finite A ⟹ n ∉ A ⟹ set_encode (insert n A) = 2^n + set_encode
A"
  by (simp add: set_encode_def)

lemma even_set_encode_iff: "finite A ⟹ even (set_encode A) ⟷ 0 ∉ A"
  by (induct set: finite) (auto simp: set_encode_def)

lemma set_encode_vimage_Suc: "set_encode (Suc -` A) = set_encode A div 2"
  apply (cases "finite A")
   apply (erule finite_induct)
    apply simp
   apply (case_tac x)
    apply (simp add: even_set_encode_iff vimage_Suc_insert_0)
   apply (simp add: finite_vimageI add.commute vimage_Suc_insert_Suc)
  apply (simp add: set_encode_def finite_vimage_Suc_iff)
  done

lemmas set_encode_div_2 = set_encode_vimage_Suc [symmetric]
```

3.5.3 From naturals to sets

```
definition set_decode :: "nat ⇒ nat set"
  where "set_decode x = {n. odd (x div 2 ^ n)}"

lemma set_decode_0 [simp]: "0 ∈ set_decode x ⟷ odd x"
  by (simp add: set_decode_def)

lemma set_decode_Suc [simp]: "Suc n ∈ set_decode x ⟷ n ∈ set_decode (x div 2)"
  by (simp add: set_decode_def div_mult2_eq)

lemma set_decode_zero [simp]: "set_decode 0 = {}"
  by (simp add: set_decode_def)

lemma set_decode_div_2: "set_decode (x div 2) = Suc -` set_decode x"
  by auto

lemma set_decode_plus_power_2:
  "n ∉ set_decode z ⟹ set_decode (2 ^ n + z) = insert n (set_decode z)"
proof (induct n arbitrary: z)
  case 0
  show ?case
  proof (rule set_eqI)
    show "q ∈ set_decode (2 ^ 0 + z) ⟷ q ∈ insert 0 (set_decode z)" for q
      by (induct q) (use 0 in simp_all)
  qed
next
  case (Suc n)
  show ?case
  proof (rule set_eqI)
    show "q ∈ set_decode (2 ^ Suc n + z) ⟷ q ∈ insert (Suc n) (set_decode z)" for q
      by (induct q) (use Suc in simp_all)
  qed
qed

lemma finite_set_decode [simp]: "finite (set_decode n)"
  apply (induct n rule: nat_less_induct)
  apply (case_tac "n = 0")
   apply simp
  apply (drule_tac x="n div 2" in spec)
  apply simp
  apply (simp add: set_decode_div_2)
  apply (simp add: finite_vimage_Suc_iff)
  done
```

3.5.4 Proof of isomorphism

```
lemma set_decode_inverse [simp]: "set_encode (set_decode n) = n"
  apply (induct n rule: nat_less_induct)
  apply (case_tac "n = 0")
   apply simp
  apply (drule_tac x="n div 2" in spec)
  apply simp
  apply (simp add: set_decode_div_2 set_encode_vimage_Suc)
  apply (erule div2_even_ext_nat)
  apply (simp add: even_set_encode_iff)
  done

lemma set_encode_inverse [simp]: "finite A ⟹ set_decode (set_encode A) = A"
  apply (erule finite_induct)
   apply simp_all
```

```
  apply (simp add: set_decode_plus_power_2)
  done

lemma inj_on_set_encode: "inj_on set_encode (Collect finite)"
  by (rule inj_on_inverseI [where g = "set_decode"]) simp

lemma set_encode_eq: "finite A ⟹ finite B ⟹ set_encode A = set_encode B ⟷ A = B"
  by (rule iffI) (simp_all add: inj_onD [OF inj_on_set_encode])

lemma subset_decode_imp_le:
  assumes "set_decode m ⊆ set_decode n"
  shows "m ≤ n"
proof -
  have "n = m + set_encode (set_decode n - set_decode m)"
  proof -
    obtain A B where
      "m = set_encode A" "finite A"
      "n = set_encode B" "finite B"
      by (metis finite_set_decode set_decode_inverse)
  with assms show ?thesis
    by auto (simp add: set_encode_def add.commute sum.subset_diff)
  qed
  then show ?thesis
    by (metis le_add1)
qed

end
```

4 Encoding (almost) everything into natural numbers

```
theory Countable
imports Old_Datatype HOL.Rat Nat_Bijection
begin
```

4.1 The class of countable types

```
class countable =
  assumes ex_inj: "∃to_nat :: 'a ⇒ nat. inj to_nat"

lemma countable_classI:
  fixes f :: "'a ⇒ nat"
  assumes "⋀x y. f x = f y ⟹ x = y"
  shows "OFCLASS('a, countable_class)"
proof (intro_classes, rule exI)
  show "inj f"
    by (rule injI [OF assms]) assumption
qed
```

4.2 Conversion functions

```
definition to_nat :: "'a::countable ⇒ nat" where
  "to_nat = (SOME f. inj f)"

definition from_nat :: "nat ⇒ 'a::countable" where
  "from_nat = inv (to_nat :: 'a ⇒ nat)"

lemma inj_to_nat [simp]: "inj to_nat"
  by (rule exE_some [OF ex_inj]) (simp add: to_nat_def)

lemma inj_on_to_nat[simp, intro]: "inj_on to_nat S"
```

```
  using inj_to_nat by (auto simp: inj_on_def)

lemma surj_from_nat [simp]: "surj from_nat"
  unfolding from_nat_def by (simp add: inj_imp_surj_inv)

lemma to_nat_split [simp]: "to_nat x = to_nat y ⟷ x = y"
  using injD [OF inj_to_nat] by auto

lemma from_nat_to_nat [simp]:
  "from_nat (to_nat x) = x"
  by (simp add: from_nat_def)
```

4.3 Finite types are countable

```
subclass (in finite) countable
proof
  have "finite (UNIV::'a set)" by (rule finite_UNIV)
  with finite_conv_nat_seg_image [of "UNIV::'a set"]
  obtain n and f :: "nat ⇒ 'a"
    where "UNIV = f ` {i. i < n}" by auto
  then have "surj f" unfolding surj_def by auto
  then have "inj (inv f)" by (rule surj_imp_inj_inv)
  then show "∃to_nat :: 'a ⇒ nat. inj to_nat" by (rule exI[of inj])
qed
```

4.4 Automatically proving countability of old-style datatypes

```
context
begin

qualified inductive finite_item :: "'a Old_Datatype.item ⇒ bool" where
  undefined: "finite_item undefined"
| In0: "finite_item x ⟹ finite_item (Old_Datatype.In0 x)"
| In1: "finite_item x ⟹ finite_item (Old_Datatype.In1 x)"
| Leaf: "finite_item (Old_Datatype.Leaf a)"
| Scons: "⟦finite_item x; finite_item y⟧ ⟹ finite_item (Old_Datatype.Scons x y)"

qualified function nth_item :: "nat ⇒ ('a::countable) Old_Datatype.item"
where
  "nth_item 0 = undefined"
| "nth_item (Suc n) =
  (case sum_decode n of
    Inl i ⇒
    (case sum_decode i of
      Inl j ⇒ Old_Datatype.In0 (nth_item j)
    | Inr j ⇒ Old_Datatype.In1 (nth_item j))
  | Inr i ⇒
    (case sum_decode i of
      Inl j ⇒ Old_Datatype.Leaf (from_nat j)
    | Inr j ⇒
      (case prod_decode j of
        (a, b) ⇒ Old_Datatype.Scons (nth_item a) (nth_item b))))"
by pat_completeness auto

lemma le_sum_encode_Inl: "x ≤ y ⟹ x ≤ sum_encode (Inl y)"
unfolding sum_encode_def by simp

lemma le_sum_encode_Inr: "x ≤ y ⟹ x ≤ sum_encode (Inr y)"
unfolding sum_encode_def by simp

qualified termination
```

```
by (relation "measure id")
  (auto simp flip: sum_encode_eq prod_encode_eq
    simp: le_imp_less_Suc le_sum_encode_Inl le_sum_encode_Inr
    le_prod_encode_1 le_prod_encode_2)

lemma nth_item_covers: "finite_item x ⟹ ∃n. nth_item n = x"
proof (induct set: finite_item)
  case undefined
  have "nth_item 0 = undefined" by simp
  thus ?case ..
next
  case (In0 x)
  then obtain n where "nth_item n = x" by fast
  hence "nth_item (Suc (sum_encode (Inl (sum_encode (Inl n))))) = Old_Datatype.In0 x" by simp
  thus ?case ..
next
  case (In1 x)
  then obtain n where "nth_item n = x" by fast
  hence "nth_item (Suc (sum_encode (Inl (sum_encode (Inr n))))) = Old_Datatype.In1 x" by simp
  thus ?case ..
next
  case (Leaf a)
  have "nth_item (Suc (sum_encode (Inr (sum_encode (Inl (to_nat a)))))) = Old_Datatype.Leaf a"
    by simp
  thus ?case ..
next
  case (Scons x y)
  then obtain i j where "nth_item i = x" and "nth_item j = y" by fast
  hence "nth_item
    (Suc (sum_encode (Inr (sum_encode (Inr (prod_encode (i, j))))))) = Old_Datatype.Scons x y"
    by simp
  thus ?case ..
qed

theorem countable_datatype:
  fixes Rep :: "'b ⇒ ('a::countable) Old_Datatype.item"
  fixes Abs :: "('a::countable) Old_Datatype.item ⇒ 'b"
  fixes rep_set :: "('a::countable) Old_Datatype.item ⇒ bool"
  assumes type: "type_definition Rep Abs (Collect rep_set)"
  assumes finite_item: "⋀x. rep_set x ⟹ finite_item x"
  shows "OFCLASS('b, countable_class)"
proof
  define f where "f y = (LEAST n. nth_item n = Rep y)" for y
  {
    fix y :: 'b
    have "rep_set (Rep y)"
      using type_definition.Rep [OF type] by simp
    hence "finite_item (Rep y)"
      by (rule finite_item)
    hence "∃n. nth_item n = Rep y"
      by (rule nth_item_covers)
    hence "nth_item (f y) = Rep y"
      unfolding f_def by (rule LeastI_ex)
    hence "Abs (nth_item (f y)) = y"
      using type_definition.Rep_inverse [OF type] by simp
  }
  hence "inj f"
    by (rule inj_on_inverseI)
  thus "∃f::'b ⇒ nat. inj f"
    by - (rule exI)
qed
```

```
ML ‹
  fun old_countable_datatype_tac ctxt =
    SUBGOAL (fn (goal, _) =>
      let
        val ty_name =
          (case goal of
            (_ $ Const (@{const_name Pure.type}, Type (@{type_name itself}, [Type (n, _)]))) => n
          | _ => raise Match)
        val typedef_info = hd (Typedef.get_info ctxt ty_name)
        val typedef_thm = #type_definition (snd typedef_info)
        val pred_name =
          (case HOLogic.dest_Trueprop (Thm.concl_of typedef_thm) of
            (_ $ _ $ _ $ (_ $ Const (n, _))) => n
          | _ => raise Match)
        val induct_info = Inductive.the_inductive_global ctxt pred_name
        val pred_names = #names (fst induct_info)
        val induct_thms = #inducts (snd induct_info)
        val alist = pred_names ~~ induct_thms
        val induct_thm = the (AList.lookup (op =) alist pred_name)
        val vars = rev (Term.add_vars (Thm.prop_of induct_thm) [])
        val insts = vars |> map (fn (_, T) => try (Thm.cterm_of ctxt)
          (Const (@{const_name Countable.finite_item}, T)))
        val induct_thm' = Thm.instantiate' [] insts induct_thm
        val rules = @{thms finite_item.intros}
      in
        SOLVED' (fn i => EVERY
          [resolve_tac ctxt @{thms countable_datatype} i,
           resolve_tac ctxt [typedef_thm] i,
           eresolve_tac ctxt [induct_thm'] i,
           REPEAT (resolve_tac ctxt rules i ORELSE assume_tac ctxt i)]) 1
      end)
›

end
```

4.5 Automatically proving countability of datatypes

ML_file *"../Tools/BNF/bnf_lfp_countable.ML"*

```
ML ‹
fun countable_datatype_tac ctxt st =
  (case try (fn () => HEADGOAL (old_countable_datatype_tac ctxt) st) () of
    SOME res => res
  | NONE => BNF_LFP_Countable.countable_datatype_tac ctxt st);

(* compatibility *)
fun countable_tac ctxt =
  SELECT_GOAL (countable_datatype_tac ctxt);
›

method_setup countable_datatype = ‹
  Scan.succeed (SIMPLE_METHOD o countable_datatype_tac)
› "prove countable class instances for datatypes"
```

4.6 More Countable types

Naturals

```
instance nat :: countable
  by (rule countable_classI [of "id"]) simp
```

Pairs

```
instance prod :: (countable, countable) countable
  by (rule countable_classI [of "λ(x, y). prod_encode (to_nat x, to_nat y)"])
    (auto simp add: prod_encode_eq)
```

Sums

```
instance sum :: (countable, countable) countable
  by (rule countable_classI [of "(λx. case x of Inl a ⇒ to_nat (False, to_nat a)
                                    | Inr b ⇒ to_nat (True, to_nat b))"])
    (simp split: sum.split_asm)
```

Integers

```
instance int :: countable
  by (rule countable_classI [of int_encode]) (simp add: int_encode_eq)
```

Options

```
instance option :: (countable) countable
  by countable_datatype
```

Lists

```
instance list :: (countable) countable
  by countable_datatype
```

String literals

```
instance String.literal :: countable
  by (rule countable_classI [of "to_nat ∘ String.explode"]) (simp add: String.explode_inject)
```

Functions

```
instance "fun" :: (finite, countable) countable
proof
  obtain xs :: "'a list" where xs: "set xs = UNIV"
    using finite_list [OF finite_UNIV] ..
  show "∃to_nat::('a ⇒ 'b) ⇒ nat. inj to_nat"
  proof
    show "inj (λf. to_nat (map f xs))"
      by (rule injI, simp add: xs fun_eq_iff)
  qed
qed
```

Typereps

```
instance typerep :: countable
  by countable_datatype
```

4.7 The rationals are countably infinite

```
definition nat_to_rat_surj :: "nat ⇒ rat" where
  "nat_to_rat_surj n = (let (a, b) = prod_decode n in Fract (int_decode a) (int_decode b))"

lemma surj_nat_to_rat_surj: "surj nat_to_rat_surj"
unfolding surj_def
proof
  fix r::rat
  show "∃n. r = nat_to_rat_surj n"
  proof (cases r)
    fix i j assume [simp]: "r = Fract i j" and "j > 0"
    have "r = (let m = int_encode i; n = int_encode j in nat_to_rat_surj (prod_encode (m, n)))"
      by (simp add: Let_def nat_to_rat_surj_def)
    thus "∃n. r = nat_to_rat_surj n" by(auto simp: Let_def)
  qed
```

qed

lemma *Rats_eq_range_nat_to_rat_surj: "ℚ = range nat_to_rat_surj"*
  by *(simp add: Rats_def surj_nat_to_rat_surj)*

context *field_char_0*
begin

lemma *Rats_eq_range_of_rat_o_nat_to_rat_surj:*
  "ℚ = range (of_rat ∘ nat_to_rat_surj)"
  using *surj_nat_to_rat_surj*
  by *(auto simp: Rats_def image_def surj_def) (blast intro: arg_cong[***where** *f = of_rat])*

lemma *surj_of_rat_nat_to_rat_surj:*
  "r ∈ ℚ ⟹ ∃n. r = of_rat (nat_to_rat_surj n)"
  by *(simp add: Rats_eq_range_of_rat_o_nat_to_rat_surj image_def)*

end

instance *rat :: countable*
proof
  show *"∃to_nat::rat ⇒ nat. inj to_nat"*
  proof
    have *"surj nat_to_rat_surj"*
      by *(rule surj_nat_to_rat_surj)*
    then show *"inj (inv nat_to_rat_surj)"*
      by *(rule surj_imp_inj_inv)*
  qed
qed

theorem *rat_denum: "∃f :: nat ⇒ rat. surj f"*
 using *surj_nat_to_rat_surj* **by** *metis*

end

5 Type of finite sets defined as a subtype of sets

theory *FSet*
imports *Main Countable*
begin

5.1 Definition of the type

typedef *'a fset = "{A :: 'a set. finite A}"* **morphisms** *fset Abs_fset*
by *auto*

setup_lifting *type_definition_fset*

5.2 Basic operations and type class instantiations

instantiation *fset :: (finite) finite*
begin
instance by *(standard; transfer; simp)*
end

instantiation *fset :: (type) "{bounded_lattice_bot, distrib_lattice, minus}"*
begin

lift_definition *bot_fset :: "'a fset"* **is** *"{}"* **parametric** *empty_transfer* **by** *simp*

lift_definition *less_eq_fset :: "'a fset ⇒ 'a fset ⇒ bool"* **is** *subset_eq* **parametric** *subset_transfer*
.

definition *less_fset :: "'a fset ⇒ 'a fset ⇒ bool"* **where** *"xs < ys ≡ xs ≤ ys ∧ xs ≠ (ys::'a fset)"*

lemma *less_fset_transfer[transfer_rule]:*
includes *lifting_syntax*
assumes *[transfer_rule]: "bi_unique A"*
shows *"((pcr_fset A) ===> (pcr_fset A) ===> (=)) (⊂) (<)"*
unfolding *less_fset_def[abs_def] psubset_eq[abs_def]* **by** *transfer_prover*

lift_definition *sup_fset :: "'a fset ⇒ 'a fset ⇒ 'a fset"* **is** *union* **parametric** *union_transfer*
by *simp*

lift_definition *inf_fset :: "'a fset ⇒ 'a fset ⇒ 'a fset"* **is** *inter* **parametric** *inter_transfer*
by *simp*

lift_definition *minus_fset :: "'a fset ⇒ 'a fset ⇒ 'a fset"* **is** *minus* **parametric** *Diff_transfer*
by *simp*

instance
by *(standard; transfer; auto)+*

end

abbreviation *fempty :: "'a fset" ("{|||}")* **where** *"{|||} ≡ bot"*
abbreviation *fsubset_eq :: "'a fset ⇒ 'a fset ⇒ bool"* **(infix** *"|⊆|" 50)* **where** *"xs |⊆| ys ≡ xs ≤ ys"*
abbreviation *fsubset :: "'a fset ⇒ 'a fset ⇒ bool"* **(infix** *"|⊂|" 50)* **where** *"xs |⊂| ys ≡ xs < ys"*
abbreviation *funion :: "'a fset ⇒ 'a fset ⇒ 'a fset"* **(infixl** *"|∪|" 65)* **where** *"xs |∪| ys ≡ sup xs ys"*
abbreviation *finter :: "'a fset ⇒ 'a fset ⇒ 'a fset"* **(infixl** *"|∩|" 65)* **where** *"xs |∩| ys ≡ inf xs ys"*
abbreviation *fminus :: "'a fset ⇒ 'a fset ⇒ 'a fset"* **(infixl** *"|-|" 65)* **where** *"xs |-| ys ≡ minus xs ys"*

instantiation *fset :: (equal) equal*
begin
definition *"HOL.equal A B ⟷ A |⊆| B ∧ B |⊆| A"*
instance by *intro_classes (auto simp add: equal_fset_def)*
end

instantiation *fset :: (type) conditionally_complete_lattice*
begin

context includes *lifting_syntax*
begin

lemma *right_total_Inf_fset_transfer:*
assumes *[transfer_rule]: "bi_unique A"* **and** *[transfer_rule]: "right_total A"*
shows *"(rel_set (rel_set A) ===> rel_set A)*
(λS. if finite (⋂S ∩ Collect (Domainp A)) then ⋂S ∩ Collect (Domainp A) else {})
(λS. if finite (Inf S) then Inf S else {})"
by *transfer_prover*

lemma *Inf_fset_transfer:*
assumes *[transfer_rule]: "bi_unique A"* **and** *[transfer_rule]: "bi_total A"*
shows *"(rel_set (rel_set A) ===> rel_set A) (λA. if finite (Inf A) then Inf A else {})*
(λA. if finite (Inf A) then Inf A else {})"
by *transfer_prover*

```
lift_definition Inf_fset :: "'a fset set ⇒ 'a fset" is "λA. if finite (Inf A) then Inf A else {}"
parametric right_total_Inf_fset_transfer Inf_fset_transfer by simp

lemma Sup_fset_transfer:
  assumes [transfer_rule]: "bi_unique A"
  shows "(rel_set (rel_set A) ===> rel_set A) (λA. if finite (Sup A) then Sup A else {})
  (λA. if finite (Sup A) then Sup A else {})" by transfer_prover

lift_definition Sup_fset :: "'a fset set ⇒ 'a fset" is "λA. if finite (Sup A) then Sup A else {}"
parametric Sup_fset_transfer by simp

lemma finite_Sup: "∃z. finite z ∧ (∀a. a ∈ X ⟶ a ≤ z) ⟹ finite (Sup X)"
by (auto intro: finite_subset)

lemma transfer_bdd_below[transfer_rule]: "(rel_set (pcr_fset (=)) ===> (=)) bdd_below bdd_below"
  by auto

end

instance
proof
  fix x z :: "'a fset"
  fix X :: "'a fset set"
  {
    assume "x ∈ X" "bdd_below X"
    then show "Inf X |⊆| x" by transfer auto
  next
    assume "X ≠ {}" "(⋀x. x ∈ X ⟹ z |⊆| x)"
    then show "z |⊆| Inf X" by transfer (clarsimp, blast)
  next
    assume "x ∈ X" "bdd_above X"
    then obtain z where "x ∈ X" "(⋀x. x ∈ X ⟹ x |⊆| z)"
      by (auto simp: bdd_above_def)
    then show "x |⊆| Sup X"
      by transfer (auto intro!: finite_Sup)
  next
    assume "X ≠ {}" "(⋀x. x ∈ X ⟹ x |⊆| z)"
    then show "Sup X |⊆| z" by transfer (clarsimp, blast)
  }
qed
end

instantiation fset :: (finite) complete_lattice
begin

lift_definition top_fset :: "'a fset" is UNIV parametric right_total_UNIV_transfer UNIV_transfer
  by simp

instance
  by (standard; transfer; auto)

end

instantiation fset :: (finite) complete_boolean_algebra
begin

lift_definition uminus_fset :: "'a fset ⇒ 'a fset" is uminus
  parametric right_total_Compl_transfer Compl_transfer by simp

instance
```

```
  by (standard; transfer) (simp_all add: Inf_Sup Diff_eq)
end

abbreviation fUNIV :: "'a::finite fset" where "fUNIV ≡ top"
abbreviation fuminus :: "'a::finite fset ⇒ 'a fset" ("|-| _" [81] 80) where "|-| x ≡ uminus x"

declare top_fset.rep_eq[simp]
```

5.3 Other operations

```
lift_definition finsert :: "'a ⇒ 'a fset ⇒ 'a fset" is insert parametric Lifting_Set.insert_transfer
  by simp

syntax
  "_insert_fset"     :: "args => 'a fset"  ("{|(_)|}")

translations
  "{|x, xs|}" == "CONST finsert x {|xs|}"
  "{|x|}"     == "CONST finsert x {||}"

lift_definition fmember :: "'a ⇒ 'a fset ⇒ bool" (infix "|∈|" 50) is Set.member
  parametric member_transfer .

abbreviation notin_fset :: "'a ⇒ 'a fset ⇒ bool" (infix "|∉|" 50) where "x |∉| S ≡ ¬ (x |∈| S)"

context includes lifting_syntax
begin

lift_definition ffilter :: "('a ⇒ bool) ⇒ 'a fset ⇒ 'a fset" is Set.filter
  parametric Lifting_Set.filter_transfer unfolding Set.filter_def by simp

lift_definition fPow :: "'a fset ⇒ 'a fset fset" is Pow parametric Pow_transfer
by (simp add: finite_subset)

lift_definition fcard :: "'a fset ⇒ nat" is card parametric card_transfer .

lift_definition fimage :: "('a ⇒ 'b) ⇒ 'a fset ⇒ 'b fset" (infixr "|`|" 90) is image
  parametric image_transfer by simp

lift_definition fthe_elem :: "'a fset ⇒ 'a" is the_elem .

lift_definition fbind :: "'a fset ⇒ ('a ⇒ 'b fset) ⇒ 'b fset" is Set.bind parametric bind_transfer
by (simp add: Set.bind_def)

lift_definition ffUnion :: "'a fset fset ⇒ 'a fset" is Union parametric Union_transfer by simp

lift_definition fBall :: "'a fset ⇒ ('a ⇒ bool) ⇒ bool" is Ball parametric Ball_transfer .
lift_definition fBex :: "'a fset ⇒ ('a ⇒ bool) ⇒ bool" is Bex parametric Bex_transfer .

lift_definition ffold :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a fset ⇒ 'b" is Finite_Set.fold .

lift_definition fset_of_list :: "'a list ⇒ 'a fset" is set by (rule finite_set)

lift_definition sorted_list_of_fset :: "'a::linorder fset ⇒ 'a list" is sorted_list_of_set .
```

5.4 Transferred lemmas from Set.thy

```
lemmas fset_eqI = set_eqI[Transfer.transferred]
lemmas fset_eq_iff[no_atp] = set_eq_iff[Transfer.transferred]
lemmas fBallI[intro!] = ballI[Transfer.transferred]
lemmas fbspec[dest?] = bspec[Transfer.transferred]
```

lemmas *fBallE[elim] = ballE[Transfer.transferred]*
lemmas *fBexI[intro] = bexI[Transfer.transferred]*
lemmas *rev_fBexI[intro?] = rev_bexI[Transfer.transferred]*
lemmas *fBexCI = bexCI[Transfer.transferred]*
lemmas *fBexE[elim!] = bexE[Transfer.transferred]*
lemmas *fBall_triv[simp] = ball_triv[Transfer.transferred]*
lemmas *fBex_triv[simp] = bex_triv[Transfer.transferred]*
lemmas *fBex_triv_one_point1[simp] = bex_triv_one_point1[Transfer.transferred]*
lemmas *fBex_triv_one_point2[simp] = bex_triv_one_point2[Transfer.transferred]*
lemmas *fBex_one_point1[simp] = bex_one_point1[Transfer.transferred]*
lemmas *fBex_one_point2[simp] = bex_one_point2[Transfer.transferred]*
lemmas *fBall_one_point1[simp] = ball_one_point1[Transfer.transferred]*
lemmas *fBall_one_point2[simp] = ball_one_point2[Transfer.transferred]*
lemmas *fBall_conj_distrib = ball_conj_distrib[Transfer.transferred]*
lemmas *fBex_disj_distrib = bex_disj_distrib[Transfer.transferred]*
lemmas *fBall_cong[fundef_cong] = ball_cong[Transfer.transferred]*
lemmas *fBex_cong[fundef_cong] = bex_cong[Transfer.transferred]*
lemmas *fsubsetI[intro!] = subsetI[Transfer.transferred]*
lemmas *fsubsetD[elim, intro?] = subsetD[Transfer.transferred]*
lemmas *rev_fsubsetD[no_atp,intro?] = rev_subsetD[Transfer.transferred]*
lemmas *fsubsetCE[no_atp,elim] = subsetCE[Transfer.transferred]*
lemmas *fsubset_eq[no_atp] = subset_eq[Transfer.transferred]*
lemmas *contra_fsubsetD[no_atp] = contra_subsetD[Transfer.transferred]*
lemmas *fsubset_refl = subset_refl[Transfer.transferred]*
lemmas *fsubset_trans = subset_trans[Transfer.transferred]*
lemmas *fset_rev_mp = set_rev_mp[Transfer.transferred]*
lemmas *fset_mp = set_mp[Transfer.transferred]*
lemmas *fsubset_not_fsubset_eq[code] = subset_not_subset_eq[Transfer.transferred]*
lemmas *eq_fmem_trans = eq_mem_trans[Transfer.transferred]*
lemmas *fsubset_antisym[intro!] = subset_antisym[Transfer.transferred]*
lemmas *fequalityD1 = equalityD1[Transfer.transferred]*
lemmas *fequalityD2 = equalityD2[Transfer.transferred]*
lemmas *fequalityE = equalityE[Transfer.transferred]*
lemmas *fequalityCE[elim] = equalityCE[Transfer.transferred]*
lemmas *eqfset_imp_iff = eqset_imp_iff[Transfer.transferred]*
lemmas *eqfelem_imp_iff = eqelem_imp_iff[Transfer.transferred]*
lemmas *fempty_iff[simp] = empty_iff[Transfer.transferred]*
lemmas *fempty_fsubsetI[iff] = empty_subsetI[Transfer.transferred]*
lemmas *equalsffemptyI = equals0I[Transfer.transferred]*
lemmas *equalsffemptyD = equals0D[Transfer.transferred]*
lemmas *fBall_fempty[simp] = ball_empty[Transfer.transferred]*
lemmas *fBex_fempty[simp] = bex_empty[Transfer.transferred]*
lemmas *fPow_iff[iff] = Pow_iff[Transfer.transferred]*
lemmas *fPowI = PowI[Transfer.transferred]*
lemmas *fPowD = PowD[Transfer.transferred]*
lemmas *fPow_bottom = Pow_bottom[Transfer.transferred]*
lemmas *fPow_top = Pow_top[Transfer.transferred]*
lemmas *fPow_not_fempty = Pow_not_empty[Transfer.transferred]*
lemmas *finter_iff[simp] = Int_iff[Transfer.transferred]*
lemmas *finterI[intro!] = IntI[Transfer.transferred]*
lemmas *finterD1 = IntD1[Transfer.transferred]*
lemmas *finterD2 = IntD2[Transfer.transferred]*
lemmas *finterE[elim!] = IntE[Transfer.transferred]*
lemmas *funion_iff[simp] = Un_iff[Transfer.transferred]*
lemmas *funionI1[elim?] = UnI1[Transfer.transferred]*
lemmas *funionI2[elim?] = UnI2[Transfer.transferred]*
lemmas *funionCI[intro!] = UnCI[Transfer.transferred]*
lemmas *funionE[elim!] = UnE[Transfer.transferred]*
lemmas *fminus_iff[simp] = Diff_iff[Transfer.transferred]*
lemmas *fminusI[intro!] = DiffI[Transfer.transferred]*
lemmas *fminusD1 = DiffD1[Transfer.transferred]*

lemmas *fminusD2 = DiffD2[Transfer.transferred]*
lemmas *fminusE[elim!] = DiffE[Transfer.transferred]*
lemmas *finsert_iff[simp] = insert_iff[Transfer.transferred]*
lemmas *finsertI1 = insertI1[Transfer.transferred]*
lemmas *finsertI2 = insertI2[Transfer.transferred]*
lemmas *finsertE[elim!] = insertE[Transfer.transferred]*
lemmas *finsertCI[intro!] = insertCI[Transfer.transferred]*
lemmas *fsubset_finsert_iff = subset_insert_iff[Transfer.transferred]*
lemmas *finsert_ident = insert_ident[Transfer.transferred]*
lemmas *fsingletonI[intro!,no_atp] = singletonI[Transfer.transferred]*
lemmas *fsingletonD[dest!,no_atp] = singletonD[Transfer.transferred]*
lemmas *fsingleton_iff = singleton_iff[Transfer.transferred]*
lemmas *fsingleton_inject[dest!] = singleton_inject[Transfer.transferred]*
lemmas *fsingleton_finsert_inj_eq[iff,no_atp] = singleton_insert_inj_eq[Transfer.transferred]*
lemmas *fsingleton_finsert_inj_eq'[iff,no_atp] = singleton_insert_inj_eq'[Transfer.transferred]*
lemmas *fsubset_fsingletonD = subset_singletonD[Transfer.transferred]*
lemmas *fminus_single_finsert = Diff_single_insert[Transfer.transferred]*
lemmas *fdoubleton_eq_iff = doubleton_eq_iff[Transfer.transferred]*
lemmas *funion_fsingleton_iff = Un_singleton_iff[Transfer.transferred]*
lemmas *fsingleton_funion_iff = singleton_Un_iff[Transfer.transferred]*
lemmas *fimage_eqI[simp, intro] = image_eqI[Transfer.transferred]*
lemmas *fimageI = imageI[Transfer.transferred]*
lemmas *rev_fimage_eqI = rev_image_eqI[Transfer.transferred]*
lemmas *fimageE[elim!] = imageE[Transfer.transferred]*
lemmas *Compr_fimage_eq = Compr_image_eq[Transfer.transferred]*
lemmas *fimage_funion = image_Un[Transfer.transferred]*
lemmas *fimage_iff = image_iff[Transfer.transferred]*
lemmas *fimage_fsubset_iff[no_atp] = image_subset_iff[Transfer.transferred]*
lemmas *fimage_fsubsetI = image_subsetI[Transfer.transferred]*
lemmas *fimage_ident[simp] = image_ident[Transfer.transferred]*
lemmas *if_split_fmem1 = if_split_mem1[Transfer.transferred]*
lemmas *if_split_fmem2 = if_split_mem2[Transfer.transferred]*
lemmas *pfsubsetI[intro!,no_atp] = psubsetI[Transfer.transferred]*
lemmas *pfsubsetE[elim!,no_atp] = psubsetE[Transfer.transferred]*
lemmas *pfsubset_finsert_iff = psubset_insert_iff[Transfer.transferred]*
lemmas *pfsubset_eq = psubset_eq[Transfer.transferred]*
lemmas *pfsubset_imp_fsubset = psubset_imp_subset[Transfer.transferred]*
lemmas *pfsubset_trans = psubset_trans[Transfer.transferred]*
lemmas *pfsubsetD = psubsetD[Transfer.transferred]*
lemmas *pfsubset_fsubset_trans = psubset_subset_trans[Transfer.transferred]*
lemmas *fsubset_pfsubset_trans = subset_psubset_trans[Transfer.transferred]*
lemmas *pfsubset_imp_ex_fmem = psubset_imp_ex_mem[Transfer.transferred]*
lemmas *fimage_fPow_mono = image_Pow_mono[Transfer.transferred]*
lemmas *fimage_fPow_surj = image_Pow_surj[Transfer.transferred]*
lemmas *fsubset_finsertI = subset_insertI[Transfer.transferred]*
lemmas *fsubset_finsertI2 = subset_insertI2[Transfer.transferred]*
lemmas *fsubset_finsert = subset_insert[Transfer.transferred]*
lemmas *funion_upper1 = Un_upper1[Transfer.transferred]*
lemmas *funion_upper2 = Un_upper2[Transfer.transferred]*
lemmas *funion_least = Un_least[Transfer.transferred]*
lemmas *finter_lower1 = Int_lower1[Transfer.transferred]*
lemmas *finter_lower2 = Int_lower2[Transfer.transferred]*
lemmas *finter_greatest = Int_greatest[Transfer.transferred]*
lemmas *fminus_fsubset = Diff_subset[Transfer.transferred]*
lemmas *fminus_fsubset_conv = Diff_subset_conv[Transfer.transferred]*
lemmas *fsubset_fempty[simp] = subset_empty[Transfer.transferred]*
lemmas *not_pfsubset_fempty[iff] = not_psubset_empty[Transfer.transferred]*
lemmas *finsert_is_funion = insert_is_Un[Transfer.transferred]*
lemmas *finsert_not_fempty[simp] = insert_not_empty[Transfer.transferred]*
lemmas *fempty_not_finsert = empty_not_insert[Transfer.transferred]*
lemmas *finsert_absorb = insert_absorb[Transfer.transferred]*

lemmas *finsert_absorb2[simp] = insert_absorb2[Transfer.transferred]*
lemmas *finsert_commute = insert_commute[Transfer.transferred]*
lemmas *finsert_fsubset[simp] = insert_subset[Transfer.transferred]*
lemmas *finsert_inter_finsert[simp] = insert_inter_insert[Transfer.transferred]*
lemmas *finsert_disjoint[simp,no_atp] = insert_disjoint[Transfer.transferred]*
lemmas *disjoint_finsert[simp,no_atp] = disjoint_insert[Transfer.transferred]*
lemmas *fimage_fempty[simp] = image_empty[Transfer.transferred]*
lemmas *fimage_finsert[simp] = image_insert[Transfer.transferred]*
lemmas *fimage_constant = image_constant[Transfer.transferred]*
lemmas *fimage_constant_conv = image_constant_conv[Transfer.transferred]*
lemmas *fimage_fimage = image_image[Transfer.transferred]*
lemmas *finsert_fimage[simp] = insert_image[Transfer.transferred]*
lemmas *fimage_is_fempty[iff] = image_is_empty[Transfer.transferred]*
lemmas *fempty_is_fimage[iff] = empty_is_image[Transfer.transferred]*
lemmas *fimage_cong = image_cong[Transfer.transferred]*
lemmas *fimage_finter_fsubset = image_Int_subset[Transfer.transferred]*
lemmas *fimage_fminus_fsubset = image_diff_subset[Transfer.transferred]*
lemmas *finter_absorb = Int_absorb[Transfer.transferred]*
lemmas *finter_left_absorb = Int_left_absorb[Transfer.transferred]*
lemmas *finter_commute = Int_commute[Transfer.transferred]*
lemmas *finter_left_commute = Int_left_commute[Transfer.transferred]*
lemmas *finter_assoc = Int_assoc[Transfer.transferred]*
lemmas *finter_ac = Int_ac[Transfer.transferred]*
lemmas *finter_absorb1 = Int_absorb1[Transfer.transferred]*
lemmas *finter_absorb2 = Int_absorb2[Transfer.transferred]*
lemmas *finter_fempty_left = Int_empty_left[Transfer.transferred]*
lemmas *finter_fempty_right = Int_empty_right[Transfer.transferred]*
lemmas *disjoint_iff_fnot_equal = disjoint_iff_not_equal[Transfer.transferred]*
lemmas *finter_funion_distrib = Int_Un_distrib[Transfer.transferred]*
lemmas *finter_funion_distrib2 = Int_Un_distrib2[Transfer.transferred]*
lemmas *finter_fsubset_iff[no_atp, simp] = Int_subset_iff[Transfer.transferred]*
lemmas *funion_absorb = Un_absorb[Transfer.transferred]*
lemmas *funion_left_absorb = Un_left_absorb[Transfer.transferred]*
lemmas *funion_commute = Un_commute[Transfer.transferred]*
lemmas *funion_left_commute = Un_left_commute[Transfer.transferred]*
lemmas *funion_assoc = Un_assoc[Transfer.transferred]*
lemmas *funion_ac = Un_ac[Transfer.transferred]*
lemmas *funion_absorb1 = Un_absorb1[Transfer.transferred]*
lemmas *funion_absorb2 = Un_absorb2[Transfer.transferred]*
lemmas *funion_fempty_left = Un_empty_left[Transfer.transferred]*
lemmas *funion_fempty_right = Un_empty_right[Transfer.transferred]*
lemmas *funion_finsert_left[simp] = Un_insert_left[Transfer.transferred]*
lemmas *funion_finsert_right[simp] = Un_insert_right[Transfer.transferred]*
lemmas *finter_finsert_left = Int_insert_left[Transfer.transferred]*
lemmas *finter_finsert_left_ifffempty[simp] = Int_insert_left_if0[Transfer.transferred]*
lemmas *finter_finsert_left_if1[simp] = Int_insert_left_if1[Transfer.transferred]*
lemmas *finter_finsert_right = Int_insert_right[Transfer.transferred]*
lemmas *finter_finsert_right_ifffempty[simp] = Int_insert_right_if0[Transfer.transferred]*
lemmas *finter_finsert_right_if1[simp] = Int_insert_right_if1[Transfer.transferred]*
lemmas *funion_finter_distrib = Un_Int_distrib[Transfer.transferred]*
lemmas *funion_finter_distrib2 = Un_Int_distrib2[Transfer.transferred]*
lemmas *funion_finter_crazy = Un_Int_crazy[Transfer.transferred]*
lemmas *fsubset_funion_eq = subset_Un_eq[Transfer.transferred]*
lemmas *funion_fempty[iff] = Un_empty[Transfer.transferred]*
lemmas *funion_fsubset_iff[no_atp, simp] = Un_subset_iff[Transfer.transferred]*
lemmas *funion_fminus_finter = Un_Diff_Int[Transfer.transferred]*
lemmas *ffunion_empty[simp] = Union_empty[Transfer.transferred]*
lemmas *ffunion_mono = Union_mono[Transfer.transferred]*
lemmas *ffunion_insert[simp] = Union_insert[Transfer.transferred]*
lemmas *fminus_finter2 = Diff_Int2[Transfer.transferred]*
lemmas *funion_finter_assoc_eq = Un_Int_assoc_eq[Transfer.transferred]*

lemmas *fBall_funion = ball_Un[Transfer.transferred]*
lemmas *fBex_funion = bex_Un[Transfer.transferred]*
lemmas *fminus_eq_fempty_iff[simp,no_atp] = Diff_eq_empty_iff[Transfer.transferred]*
lemmas *fminus_cancel[simp] = Diff_cancel[Transfer.transferred]*
lemmas *fminus_idemp[simp] = Diff_idemp[Transfer.transferred]*
lemmas *fminus_triv = Diff_triv[Transfer.transferred]*
lemmas *fempty_fminus[simp] = empty_Diff[Transfer.transferred]*
lemmas *fminus_fempty[simp] = Diff_empty[Transfer.transferred]*
lemmas *fminus_finsertffempty[simp,no_atp] = Diff_insert0[Transfer.transferred]*
lemmas *fminus_finsert = Diff_insert[Transfer.transferred]*
lemmas *fminus_finsert2 = Diff_insert2[Transfer.transferred]*
lemmas *finsert_fminus_if = insert_Diff_if[Transfer.transferred]*
lemmas *finsert_fminus1[simp] = insert_Diff1[Transfer.transferred]*
lemmas *finsert_fminus_single[simp] = insert_Diff_single[Transfer.transferred]*
lemmas *finsert_fminus = insert_Diff[Transfer.transferred]*
lemmas *fminus_finsert_absorb = Diff_insert_absorb[Transfer.transferred]*
lemmas *fminus_disjoint[simp] = Diff_disjoint[Transfer.transferred]*
lemmas *fminus_partition = Diff_partition[Transfer.transferred]*
lemmas *double_fminus = double_diff[Transfer.transferred]*
lemmas *funion_fminus_cancel[simp] = Un_Diff_cancel[Transfer.transferred]*
lemmas *funion_fminus_cancel2[simp] = Un_Diff_cancel2[Transfer.transferred]*
lemmas *fminus_funion = Diff_Un[Transfer.transferred]*
lemmas *fminus_finter = Diff_Int[Transfer.transferred]*
lemmas *funion_fminus = Un_Diff[Transfer.transferred]*
lemmas *finter_fminus = Int_Diff[Transfer.transferred]*
lemmas *fminus_finter_distrib = Diff_Int_distrib[Transfer.transferred]*
lemmas *fminus_finter_distrib2 = Diff_Int_distrib2[Transfer.transferred]*
lemmas *fUNIV_bool[no_atp] = UNIV_bool[Transfer.transferred]*
lemmas *fPow_fempty[simp] = Pow_empty[Transfer.transferred]*
lemmas *fPow_finsert = Pow_insert[Transfer.transferred]*
lemmas *funion_fPow_fsubset = Un_Pow_subset[Transfer.transferred]*
lemmas *fPow_finter_eq[simp] = Pow_Int_eq[Transfer.transferred]*
lemmas *fset_eq_fsubset = set_eq_subset[Transfer.transferred]*
lemmas *fsubset_iff[no_atp] = subset_iff[Transfer.transferred]*
lemmas *fsubset_iff_pfsubset_eq = subset_iff_psubset_eq[Transfer.transferred]*
lemmas *all_not_fin_conv[simp] = all_not_in_conv[Transfer.transferred]*
lemmas *ex_fin_conv = ex_in_conv[Transfer.transferred]*
lemmas *fimage_mono = image_mono[Transfer.transferred]*
lemmas *fPow_mono = Pow_mono[Transfer.transferred]*
lemmas *finsert_mono = insert_mono[Transfer.transferred]*
lemmas *funion_mono = Un_mono[Transfer.transferred]*
lemmas *finter_mono = Int_mono[Transfer.transferred]*
lemmas *fminus_mono = Diff_mono[Transfer.transferred]*
lemmas *fin_mono = in_mono[Transfer.transferred]*
lemmas *fthe_felem_eq[simp] = the_elem_eq[Transfer.transferred]*
lemmas *fLeast_mono = Least_mono[Transfer.transferred]*
lemmas *fbind_fbind = bind_bind[Transfer.transferred]*
lemmas *fempty_fbind[simp] = empty_bind[Transfer.transferred]*
lemmas *nonfempty_fbind_const = nonempty_bind_const[Transfer.transferred]*
lemmas *fbind_const = bind_const[Transfer.transferred]*
lemmas *ffmember_filter[simp] = member_filter[Transfer.transferred]*
lemmas *fequalityI = equalityI[Transfer.transferred]*
lemmas *fset_of_list_simps[simp] = set_simps[Transfer.transferred]*
lemmas *fset_of_list_append[simp] = set_append[Transfer.transferred]*
lemmas *fset_of_list_rev[simp] = set_rev[Transfer.transferred]*
lemmas *fset_of_list_map[simp] = set_map[Transfer.transferred]*

5.5 Additional lemmas

5.5.1 *ffUnion*

lemmas *ffUnion_funion_distrib[simp] = Union_Un_distrib[Transfer.transferred]*

5.5.2 *fbind*

lemma *fbind_cong[fundef_cong]: "A = B ⟹ (⋀x. x |∈| B ⟹ f x = g x) ⟹ fbind A f = fbind B g"*
by *transfer force*

5.5.3 *fsingleton*

lemmas *fsingletonE = fsingletonD [elim_format]*

5.5.4 *femepty*

lemma *fempty_ffilter[simp]: "ffilter (λ_. False) A = {|||}"*
by *transfer auto*

lemma *femptyE [elim!]: "a |∈| {|||} ⟹ P"*
by *simp*

5.5.5 *fset*

lemmas *fset_simps[simp] = bot_fset.rep_eq finsert.rep_eq*

lemma *finite_fset [simp]:*
shows *"finite (fset S)"*
by *transfer simp*

lemmas *fset_cong = fset_inject*

lemma *filter_fset [simp]:*
shows *"fset (ffilter P xs) = Collect P ∩ fset xs"*
by *transfer auto*

lemma *notin_fset: "x |∉| S ⟷ x ∉ fset S"* **by** *(simp add: fmember.rep_eq)*

lemmas *inter_fset[simp] = inf_fset.rep_eq*

lemmas *union_fset[simp] = sup_fset.rep_eq*

lemmas *minus_fset[simp] = minus_fset.rep_eq*

5.5.6 *ffilter*

lemma *subset_ffilter:*
"ffilter P A |⊆| ffilter Q A = (∀ x. x |∈| A ⟶ P x ⟶ Q x)"
by *transfer auto*

lemma *eq_ffilter:*
"(ffilter P A = ffilter Q A) = (∀x. x |∈| A ⟶ P x = Q x)"
by *transfer auto*

lemma *pfsubset_ffilter:*
"(⋀x. x |∈| A ⟹ P x ⟹ Q x) ⟹ (x |∈| A ∧ ¬ P x ∧ Q x) ⟹
ffilter P A |⊂| ffilter Q A"
unfolding *less_fset_def* **by** *(auto simp add: subset_ffilter eq_ffilter)*

5.5.7 *fset_of_list*

lemma *fset_of_list_filter[simp]:*
"fset_of_list (filter P xs) = ffilter P (fset_of_list xs)"
by *transfer (auto simp: Set.filter_def)*

lemma *fset_of_list_subset[intro]:*
"set xs ⊆ set ys ⟹ fset_of_list xs |⊆| fset_of_list ys"
by *transfer simp*

lemma *fset_of_list_elem: "(x |∈| fset_of_list xs) ⟷ (x ∈ set xs)"*
by *transfer simp*

5.5.8 *finsert*

lemma *set_finsert:*
assumes *"x |∈| A"*
obtains *B* **where** *"A = finsert x B"* **and** *"x |∉| B"*
using *assms* **by** *transfer (metis Set.set_insert finite_insert)*

lemma *mk_disjoint_finsert: "a |∈| A ⟹ ∃B. A = finsert a B ∧ a |∉| B"*
by *(rule exI [***where** *x = "A |-| {|a|}"]) blast*

lemma *finsert_eq_iff:*
assumes *"a |∉| A"* **and** *"b |∉| B"*
shows *"(finsert a A = finsert b B) =*
(if a = b then A = B else ∃C. A = finsert b C ∧ b |∉| C ∧ B = finsert a C ∧ a |∉| C)"
using *assms* **by** *transfer (force simp: insert_eq_iff)*

5.5.9 *fimage*

lemma *subset_fimage_iff: "(B |⊆| f|‘|A) = (∃ AA. AA |⊆| A ∧ B = f|‘|AA)"*
by *transfer (metis mem_Collect_eq rev_finite_subset subset_image_iff)*

5.5.10 bounded quantification

lemma *bex_simps [simp, no_atp]:*
"⋀A P Q. fBex A (λx. P x ∧ Q) = (fBex A P ∧ Q)"
"⋀A P Q. fBex A (λx. P ∧ Q x) = (P ∧ fBex A Q)"
"⋀P. fBex {||} P = False"
"⋀a B P. fBex (finsert a B) P = (P a ∨ fBex B P)"
"⋀A P f. fBex (f |‘| A) P = fBex A (λx. P (f x))"
"⋀A P. (¬ fBex A P) = fBall A (λx. ¬ P x)"
by *auto*

lemma *ball_simps [simp, no_atp]:*
"⋀A P Q. fBall A (λx. P x ∨ Q) = (fBall A P ∨ Q)"
"⋀A P Q. fBall A (λx. P ∨ Q x) = (P ∨ fBall A Q)"
"⋀A P Q. fBall A (λx. P ⟶ Q x) = (P ⟶ fBall A Q)"
"⋀A P Q. fBall A (λx. P x ⟶ Q) = (fBex A P ⟶ Q)"
"⋀P. fBall {||} P = True"
"⋀a B P. fBall (finsert a B) P = (P a ∧ fBall B P)"
"⋀A P f. fBall (f |‘| A) P = fBall A (λx. P (f x))"
"⋀A P. (¬ fBall A P) = fBex A (λx. ¬ P x)"
by *auto*

lemma *atomize_fBall:*
"(⋀x. x |∈| A ==> P x) == Trueprop (fBall A (λx. P x))"
apply *(simp only: atomize_all atomize_imp)*
apply *(rule equal_intr_rule)*
by *(transfer, simp)+*

lemma *fBall_mono[mono]: "P ≤ Q ⟹ fBall S P ≤ fBall S Q"*
by *auto*

lemma *fBex_mono[mono]: "P ≤ Q ⟹ fBex S P ≤ fBex S Q"*
by *auto*

end

5.5.11 *fcard*

lemma *fcard_fempty:*
"fcard {|||} = 0"
by *transfer (rule card_empty)*

lemma *fcard_finsert_disjoint:*
"x |∉| A ⟹ fcard (finsert x A) = Suc (fcard A)"
by *transfer (rule card_insert_disjoint)*

lemma *fcard_finsert_if:*
"fcard (finsert x A) = (if x |∈| A then fcard A else Suc (fcard A))"
by *transfer (rule card_insert_if)*

lemma *fcard_0_eq [simp, no_atp]:*
"fcard A = 0 ⟷ A = {|||}"
by *transfer (rule card_0_eq)*

lemma *fcard_Suc_fminus1:*
"x |∈| A ⟹ Suc (fcard (A |-| {|x|})) = fcard A"
by *transfer (rule card_Suc_Diff1)*

lemma *fcard_fminus_fsingleton:*
"x |∈| A ⟹ fcard (A |-| {|x|}) = fcard A - 1"
by *transfer (rule card_Diff_singleton)*

lemma *fcard_fminus_fsingleton_if:*
"fcard (A |-| {|x|}) = (if x |∈| A then fcard A - 1 else fcard A)"
by *transfer (rule card_Diff_singleton_if)*

lemma *fcard_fminus_finsert[simp]:*
assumes *"a |∈| A"* **and** *"a |∉| B"*
shows *"fcard (A |-| finsert a B) = fcard (A |-| B) - 1"*
using *assms* **by** *transfer (rule card_Diff_insert)*

lemma *fcard_finsert: "fcard (finsert x A) = Suc (fcard (A |-| {|x|}))"*
by *transfer (rule card_insert)*

lemma *fcard_finsert_le: "fcard A ≤ fcard (finsert x A)"*
by *transfer (rule card_insert_le)*

lemma *fcard_mono:*
"A |⊆| B ⟹ fcard A ≤ fcard B"
by *transfer (rule card_mono)*

lemma *fcard_seteq: "A |⊆| B ⟹ fcard B ≤ fcard A ⟹ A = B"*
by *transfer (rule card_seteq)*

lemma *pfsubset_fcard_mono: "A |⊂| B ⟹ fcard A < fcard B"*
by *transfer (rule psubset_card_mono)*

lemma *fcard_funion_finter:*
"fcard A + fcard B = fcard (A |∪| B) + fcard (A |∩| B)"

by *transfer (rule card_Un_Int)*

lemma *fcard_funion_disjoint:*
"A |∩| B = {||} ⟹ fcard (A |∪| B) = fcard A + fcard B"
by *transfer (rule card_Un_disjoint)*

lemma *fcard_funion_fsubset:*
"B |⊆| A ⟹ fcard (A |-| B) = fcard A - fcard B"
by *transfer (rule card_Diff_subset)*

lemma *diff_fcard_le_fcard_fminus:*
"fcard A - fcard B ≤ fcard(A |-| B)"
by *transfer (rule diff_card_le_card_Diff)*

lemma *fcard_fminus1_less: "x |∈| A ⟹ fcard (A |-| {|x|}) < fcard A"*
by *transfer (rule card_Diff1_less)*

lemma *fcard_fminus2_less:*
"x |∈| A ⟹ y |∈| A ⟹ fcard (A |-| {|x|} |-| {|y|}) < fcard A"
by *transfer (rule card_Diff2_less)*

lemma *fcard_fminus1_le: "fcard (A |-| {|x|}) ≤ fcard A"*
by *transfer (rule card_Diff1_le)*

lemma *fcard_pfsubset: "A |⊆| B ⟹ fcard A < fcard B ⟹ A < B"*
by *transfer (rule card_psubset)*

5.5.12 *sorted_list_of_fset*

lemma *sorted_list_of_fset_simps[simp]:*
"set (sorted_list_of_fset S) = fset S"
"fset_of_list (sorted_list_of_fset S) = S"
by *(transfer, simp)+*

5.5.13 *ffold*

context *comp_fun_commute*
begin
lemmas *ffold_empty[simp] = fold_empty[Transfer.transferred]*

lemma *ffold_finsert [simp]:*
assumes *"x |∉| A"*
shows *"ffold f z (finsert x A) = f x (ffold f z A)"*
using *assms* **by** *(transfer fixing: f) (rule fold_insert)*

lemma *ffold_fun_left_comm:*
"f x (ffold f z A) = ffold f (f x z) A"
by *(transfer fixing: f) (rule fold_fun_left_comm)*

lemma *ffold_finsert2:*
"x |∉| A ⟹ ffold f z (finsert x A) = ffold f (f x z) A"
by *(transfer fixing: f) (rule fold_insert2)*

lemma *ffold_rec:*
assumes *"x |∈| A"*
shows *"ffold f z A = f x (ffold f z (A |-| {|x|}))"*
using *assms* **by** *(transfer fixing: f) (rule fold_rec)*

lemma *ffold_finsert_fremove:*
"ffold f z (finsert x A) = f x (ffold f z (A |-| {|x|}))"
by *(transfer fixing: f) (rule fold_insert_remove)*

end

lemma *ffold_fimage:*
  assumes *"inj_on g (fset A)"*
  shows *"ffold f z (g |‘| A) = ffold (f ∘ g) z A"*
using *assms* **by** *transfer' (rule fold_image)*

lemma *ffold_cong:*
  assumes *"comp_fun_commute f" "comp_fun_commute g"*
  "⋀x. x |∈| A ⟹ f x = g x"
    and *"s = t"* **and** *"A = B"*
  shows *"ffold f s A = ffold g t B"*
using *assms* **by** *transfer (metis Finite_Set.fold_cong)*

context *comp_fun_idem*
begin

  lemma *ffold_finsert_idem:*
    "ffold f z (finsert x A) = f x (ffold f z A)"
    by *(transfer fixing: f) (rule fold_insert_idem)*

  declare *ffold_finsert [simp del] ffold_finsert_idem [simp]*

  lemma *ffold_finsert_idem2:*
    "ffold f z (finsert x A) = ffold f (f x z) A"
    by *(transfer fixing: f) (rule fold_insert_idem2)*

end

5.5.14 Group operations

locale *comm_monoid_fset = comm_monoid*
begin

sublocale *set: comm_monoid_set* **..**

lift_definition *F :: "('b ⇒ 'a) ⇒ 'b fset ⇒ 'a"* **is** *set.F* **.**

lemmas *cong[fundef_cong] = set.cong[Transfer.transferred]*

lemma *strong_cong[cong]:*
  assumes *"A = B" "⋀x. x |∈| B =simp=> g x = h x"*
  shows *"F g A = F h B"*
using *assms* **unfolding** *simp_implies_def* **by** *(auto cong: cong)*

end

context *comm_monoid_add* **begin**

sublocale *fsum: comm_monoid_fset plus 0*
  rewrites *"comm_monoid_set.F plus 0 = sum"*
  defines *fsum = fsum.F*
proof -
  show *"comm_monoid_fset (+) 0"* **by** *standard*

  show *"comm_monoid_set.F (+) 0 = sum"* **unfolding** *sum_def* **..**
qed

end

5.5.15 Semilattice operations

```
locale semilattice_fset = semilattice
begin

sublocale set: semilattice_set ..

lift_definition F :: "'a fset ⇒ 'a" is set.F .

lemma eq_fold: "F (finsert x A) = ffold f x A"
  by transfer (rule set.eq_fold)

lemma singleton [simp]: "F {|x|} = x"
  by transfer (rule set.singleton)

lemma insert_not_elem: "x |∉| A ⟹ A ≠ {||} ⟹ F (finsert x A) = x * F A"
  by transfer (rule set.insert_not_elem)

lemma in_idem: "x |∈| A ⟹ x * F A = F A"
  by transfer (rule set.in_idem)

lemma insert [simp]: "A ≠ {||} ⟹ F (finsert x A) = x * F A"
  by transfer (rule set.insert)

end

locale semilattice_order_fset = binary?: semilattice_order + semilattice_fset
begin

end


context linorder begin

sublocale fMin: semilattice_order_fset min less_eq less
  rewrites "semilattice_set.F min = Min"
  defines fMin = fMin.F
proof -
  show "semilattice_order_fset min (≤) (<)" by standard

  show "semilattice_set.F min = Min" unfolding Min_def ..
qed

sublocale fMax: semilattice_order_fset max greater_eq greater
  rewrites "semilattice_set.F max = Max"
  defines fMax = fMax.F
proof -
  show "semilattice_order_fset max (≥) (>)"
    by standard

  show "semilattice_set.F max = Max"
    unfolding Max_def ..
qed

end

lemma mono_fMax_commute: "mono f ⟹ A ≠ {||} ⟹ f (fMax A) = fMax (f |`| A)"
  by transfer (rule mono_Max_commute)

lemma mono_fMin_commute: "mono f ⟹ A ≠ {||} ⟹ f (fMin A) = fMin (f |`| A)"
  by transfer (rule mono_Min_commute)
```

```
lemma fMax_in[simp]: "A ≠ {||} ⟹ fMax A |∈| A"
  by transfer (rule Max_in)

lemma fMin_in[simp]: "A ≠ {||} ⟹ fMin A |∈| A"
  by transfer (rule Min_in)

lemma fMax_ge[simp]: "x |∈| A ⟹ x ≤ fMax A"
  by transfer (rule Max_ge)

lemma fMin_le[simp]: "x |∈| A ⟹ fMin A ≤ x"
  by transfer (rule Min_le)

lemma fMax_eqI: "(⋀y. y |∈| A ⟹ y ≤ x) ⟹ x |∈| A ⟹ fMax A = x"
  by transfer (rule Max_eqI)

lemma fMin_eqI: "(⋀y. y |∈| A ⟹ x ≤ y) ⟹ x |∈| A ⟹ fMin A = x"
  by transfer (rule Min_eqI)

lemma fMax_finsert[simp]: "fMax (finsert x A) = (if A = {||} then x else max x (fMax A))"
  by transfer simp

lemma fMin_finsert[simp]: "fMin (finsert x A) = (if A = {||} then x else min x (fMin A))"
  by transfer simp

context linorder begin

lemma fset_linorder_max_induct[case_names fempty finsert]:
  assumes "P {||}"
  and     "⋀x S. ⟦∀y. y |∈| S ⟶ y < x; P S⟧ ⟹ P (finsert x S)"
  shows "P S"
proof -

  note Domainp_forall_transfer[transfer_rule]
  show ?thesis
  using assms by (transfer fixing: less) (auto intro: finite_linorder_max_induct)
qed

lemma fset_linorder_min_induct[case_names fempty finsert]:
  assumes "P {||}"
  and     "⋀x S. ⟦∀y. y |∈| S ⟶ y > x; P S⟧ ⟹ P (finsert x S)"
  shows "P S"
proof -

  note Domainp_forall_transfer[transfer_rule]
  show ?thesis
  using assms by (transfer fixing: less) (auto intro: finite_linorder_min_induct)
qed

end
```

5.6 Choice in fsets

```
lemma fset_choice:
  assumes "∀x. x |∈| A ⟶ (∃y. P x y)"
  shows "∃f. ∀x. x |∈| A ⟶ P x (f x)"
  using assms by transfer metis
```

5.7 Induction and Cases rules for fsets

```
lemma fset_exhaust [case_names empty insert, cases type: fset]:
```

```
  assumes fempty_case: "S = {||} ⟹ P"
  and     finsert_case: "⋀x S'. S = finsert x S' ⟹ P"
  shows "P"
  using assms by transfer blast

lemma fset_induct [case_names empty insert]:
  assumes fempty_case: "P {||}"
  and     finsert_case: "⋀x S. P S ⟹ P (finsert x S)"
  shows "P S"
proof -

  note Domainp_forall_transfer[transfer_rule]
  show ?thesis
  using assms by transfer (auto intro: finite_induct)
qed

lemma fset_induct_stronger [case_names empty insert, induct type: fset]:
  assumes empty_fset_case: "P {||}"
  and     insert_fset_case: "⋀x S. ⟦x |∉| S; P S⟧ ⟹ P (finsert x S)"
  shows "P S"
proof -

  note Domainp_forall_transfer[transfer_rule]
  show ?thesis
  using assms by transfer (auto intro: finite_induct)
qed

lemma fset_card_induct:
  assumes empty_fset_case: "P {||}"
  and     card_fset_Suc_case: "⋀S T. Suc (fcard S) = (fcard T) ⟹ P S ⟹ P T"
  shows "P S"
proof (induct S)
  case empty
  show "P {||}" by (rule empty_fset_case)
next
  case (insert x S)
  have h: "P S" by fact
  have "x |∉| S" by fact
  then have "Suc (fcard S) = fcard (finsert x S)"
    by transfer auto
  then show "P (finsert x S)"
    using h card_fset_Suc_case by simp
qed

lemma fset_strong_cases:
  obtains "xs = {||}"
    | ys x where "x |∉| ys" and "xs = finsert x ys"
by transfer blast

lemma fset_induct2:
  "P {||} {||} ⟹
  (⋀x xs. x |∉| xs ⟹ P (finsert x xs) {||}) ⟹
  (⋀y ys. y |∉| ys ⟹ P {||} (finsert y ys)) ⟹
  (⋀x xs y ys. ⟦P xs ys; x |∉| xs; y |∉| ys⟧ ⟹ P (finsert x xs) (finsert y ys)) ⟹
  P xsa ysa"
  apply (induct xsa arbitrary: ysa)
  apply (induct_tac x rule: fset_induct_stronger)
  apply simp_all
  apply (induct_tac xa rule: fset_induct_stronger)
  apply simp_all
  done
```

5.8 Setup for Lifting/Transfer

5.8.1 Relator and predicator properties

lift_definition *rel_fset :: "('a ⇒ 'b ⇒ bool) ⇒ 'a fset ⇒ 'b fset ⇒ bool"* **is** *rel_set*
parametric *rel_set_transfer* .

lemma *rel_fset_alt_def: "rel_fset R = (λA B. (∀x.∃y. x|∈|A ⟶ y|∈|B ∧ R x y)*
∧ (∀y. ∃x. y|∈|B ⟶ x|∈|A ∧ R x y))"
apply *(rule ext)+*
apply *transfer'*
apply *(subst rel_set_def[unfolded fun_eq_iff])*
by *blast*

lemma *finite_rel_set:*
assumes *fin: "finite X" "finite Z"*
assumes *R_S: "rel_set (R OO S) X Z"*
shows *"∃Y. finite Y ∧ rel_set R X Y ∧ rel_set S Y Z"*
proof -
obtain *f* **where** *f: "∀x∈X. R x (f x) ∧ (∃z∈Z. S (f x) z)"*
apply *atomize_elim*
apply *(subst bchoice_iff[symmetric])*
using *R_S[unfolded rel_set_def OO_def]* **by** *blast*

obtain *g* **where** *g: "∀z∈Z. S (g z) z ∧ (∃x∈X. R x (g z))"*
apply *atomize_elim*
apply *(subst bchoice_iff[symmetric])*
using *R_S[unfolded rel_set_def OO_def]* **by** *blast*

let *?Y = "f ‘ X ∪ g ‘ Z"*
have *"finite ?Y"* **by** *(simp add: fin)*
moreover have *"rel_set R X ?Y"*
unfolding *rel_set_def*
using *f g* **by** *clarsimp blast*
moreover have *"rel_set S ?Y Z"*
unfolding *rel_set_def*
using *f g* **by** *clarsimp blast*
ultimately show *?thesis* **by** *metis*
qed

5.8.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting_syntax*
begin

lemmas *fempty_transfer [transfer_rule] = empty_transfer[Transfer.transferred]*

lemma *finsert_transfer [transfer_rule]:*
"(A ===> rel_fset A ===> rel_fset A) finsert finsert"
unfolding *rel_fun_def rel_fset_alt_def* **by** *blast*

lemma *funion_transfer [transfer_rule]:*
"(rel_fset A ===> rel_fset A ===> rel_fset A) funion funion"
unfolding *rel_fun_def rel_fset_alt_def* **by** *blast*

lemma *ffUnion_transfer [transfer_rule]:*
"(rel_fset (rel_fset A) ===> rel_fset A) ffUnion ffUnion"
unfolding *rel_fun_def rel_fset_alt_def* **by** *transfer (simp, fast)*

lemma *fimage_transfer [transfer_rule]:*

```
  "((A ===> B) ===> rel_fset A ===> rel_fset B) fimage fimage"
  unfolding rel_fun_def rel_fset_alt_def by simp blast

lemma fBall_transfer [transfer_rule]:
  "(rel_fset A ===> (A ===> (=)) ===> (=)) fBall fBall"
  unfolding rel_fset_alt_def rel_fun_def by blast

lemma fBex_transfer [transfer_rule]:
  "(rel_fset A ===> (A ===> (=)) ===> (=)) fBex fBex"
  unfolding rel_fset_alt_def rel_fun_def by blast


lemma fPow_transfer [transfer_rule]:
  "(rel_fset A ===> rel_fset (rel_fset A)) fPow fPow"
  unfolding rel_fun_def
  using Pow_transfer[unfolded rel_fun_def, rule_format, Transfer.transferred]
  by blast

lemma rel_fset_transfer [transfer_rule]:
  "((A ===> B ===> (=)) ===> rel_fset A ===> rel_fset B ===> (=))
    rel_fset rel_fset"
  unfolding rel_fun_def
  using rel_set_transfer[unfolded rel_fun_def,rule_format, Transfer.transferred, where A = A and B
= B]
  by simp

lemma bind_transfer [transfer_rule]:
  "(rel_fset A ===> (A ===> rel_fset B) ===> rel_fset B) fbind fbind"
  unfolding rel_fun_def
  using bind_transfer[unfolded rel_fun_def, rule_format, Transfer.transferred] by blast
```

Rules requiring bi-unique, bi-total or right-total relations

```
lemma fmember_transfer [transfer_rule]:
  assumes "bi_unique A"
  shows "(A ===> rel_fset A ===> (=)) (|∈|) (|∈|)"
  using assms unfolding rel_fun_def rel_fset_alt_def bi_unique_def by metis

lemma finter_transfer [transfer_rule]:
  assumes "bi_unique A"
  shows "(rel_fset A ===> rel_fset A ===> rel_fset A) finter finter"
  using assms unfolding rel_fun_def
  using inter_transfer[unfolded rel_fun_def, rule_format, Transfer.transferred] by blast

lemma fminus_transfer [transfer_rule]:
  assumes "bi_unique A"
  shows "(rel_fset A ===> rel_fset A ===> rel_fset A) (|-|) (|-|)"
  using assms unfolding rel_fun_def
  using Diff_transfer[unfolded rel_fun_def, rule_format, Transfer.transferred] by blast

lemma fsubset_transfer [transfer_rule]:
  assumes "bi_unique A"
  shows "(rel_fset A ===> rel_fset A ===> (=)) (|⊆|) (|⊆|)"
  using assms unfolding rel_fun_def
  using subset_transfer[unfolded rel_fun_def, rule_format, Transfer.transferred] by blast

lemma fSup_transfer [transfer_rule]:
  "bi_unique A ⟹ (rel_set (rel_fset A) ===> rel_fset A) Sup Sup"
  unfolding rel_fun_def
  apply clarify
  apply transfer'
  using Sup_fset_transfer[unfolded rel_fun_def] by blast
```

```
lemma fInf_transfer [transfer_rule]:
  assumes "bi_unique A" and "bi_total A"
  shows "(rel_set (rel_fset A) ===> rel_fset A) Inf Inf"
  using assms unfolding rel_fun_def
  apply clarify
  apply transfer'
  using Inf_fset_transfer[unfolded rel_fun_def] by blast

lemma ffilter_transfer [transfer_rule]:
  assumes "bi_unique A"
  shows "((A ===> (=)) ===> rel_fset A ===> rel_fset A) ffilter ffilter"
  using assms unfolding rel_fun_def
  using Lifting_Set.filter_transfer[unfolded rel_fun_def, rule_format, Transfer.transferred] by blast

lemma card_transfer [transfer_rule]:
  "bi_unique A ⟹ (rel_fset A ===> (=)) fcard fcard"
  unfolding rel_fun_def
  using card_transfer[unfolded rel_fun_def, rule_format, Transfer.transferred] by blast

end

lifting_update fset.lifting
lifting_forget fset.lifting
```

5.9 BNF setup

```
context
includes fset.lifting
begin

lemma rel_fset_alt:
  "rel_fset R a b ⟷ (∀t ∈ fset a. ∃u ∈ fset b. R t u) ∧ (∀t ∈ fset b. ∃u ∈ fset a. R u t)"
by transfer (simp add: rel_set_def)

lemma fset_to_fset: "finite A ⟹ fset (the_inv fset A) = A"
apply (rule f_the_inv_into_f[unfolded inj_on_def])
apply (simp add: fset_inject)
apply (rule range_eqI Abs_fset_inverse[symmetric] CollectI)+
.

lemma rel_fset_aux:
"(∀t ∈ fset a. ∃u ∈ fset b. R t u) ∧ (∀u ∈ fset b. ∃t ∈ fset a. R t u) ⟷
 ((BNF_Def.Grp {a. fset a ⊆ {(a, b). R a b}} (fimage fst))^-1-1 OO
  BNF_Def.Grp {a. fset a ⊆ {(a, b). R a b}} (fimage snd)) a b" (is "?L = ?R")
proof
  assume ?L
  define R' where "R' =
    the_inv fset (Collect (case_prod R) ∩ (fset a × fset b))" (is "_ = the_inv fset ?L'")
  have "finite ?L'" by (intro finite_Int[OF disjI2] finite_cartesian_product) (transfer, simp)+
  hence *: "fset R' = ?L'" unfolding R'_def by (intro fset_to_fset)
  show ?R unfolding Grp_def relcompp.simps conversep.simps
  proof (intro CollectI case_prodI exI[of _ a] exI[of _ b] exI[of _ R'] conjI refl)
    from * show "a = fimage fst R'" using conjunct1[OF ‹?L›]
      by (transfer, auto simp add: image_def Int_def split: prod.splits)
    from * show "b = fimage snd R'" using conjunct2[OF ‹?L›]
      by (transfer, auto simp add: image_def Int_def split: prod.splits)
  qed (auto simp add: *)
next
```

```
  assume ?R thus ?L unfolding Grp_def relcompp.simps conversep.simps
  apply (simp add: subset_eq Ball_def)
  apply (rule conjI)
  apply (transfer, clarsimp, metis snd_conv)
  by (transfer, clarsimp, metis fst_conv)
qed

bnf "'a fset"
  map: fimage
  sets: fset
  bd: natLeq
  wits: "{||}"
  rel: rel_fset
apply -
         apply transfer' apply simp
        apply transfer' apply force
       apply transfer apply force
      apply transfer' apply force
     apply (rule natLeq_card_order)
    apply (rule natLeq_cinfinite)
   apply transfer apply (metis ordLess_imp_ordLeq finite_iff_ordLess_natLeq)
  apply (fastforce simp: rel_fset_alt)
 apply (simp add: Grp_def relcompp.simps conversep.simps fun_eq_iff rel_fset_alt
   rel_fset_aux[unfolded OO_Grp_alt])
apply transfer apply simp
done

lemma rel_fset_fset: "rel_set χ (fset A1) (fset A2) = rel_fset χ A1 A2"
  by transfer (rule refl)

end

lemmas [simp] = fset.map_comp fset.map_id fset.set_map
```

5.10 Size setup

```
context includes fset.lifting begin
lift_definition size_fset :: "('a ⇒ nat) ⇒ 'a fset ⇒ nat" is "λf. sum (Suc ∘ f)" .
end

instantiation fset :: (type) size begin
definition size_fset where
  size_fset_overloaded_def: "size_fset = FSet.size_fset (λ_. 0)"
instance ..
end

lemmas size_fset_simps[simp] =
  size_fset_def[THEN meta_eq_to_obj_eq, THEN fun_cong, THEN fun_cong,
    unfolded map_fun_def comp_def id_apply]

lemmas size_fset_overloaded_simps[simp] =
  size_fset_simps[of "λ_. 0", unfolded add_0_left add_0_right,
    folded size_fset_overloaded_def]

lemma fset_size_o_map: "inj f ⟹ size_fset g ∘ fimage f = size_fset (g ∘ f)"
  apply (subst fun_eq_iff)
  including fset.lifting by transfer (auto intro: sum.reindex_cong subset_inj_on)

setup ‹
BNF_LFP_Size.register_size_global @{type_name fset} @{const_name size_fset}
  @{thm size_fset_overloaded_def} @{thms size_fset_simps size_fset_overloaded_simps}
```

```
  @{thms fset_size_o_map}
⟩
```

lifting_update *fset.lifting*
lifting_forget *fset.lifting*

5.11 Advanced relator customization

Set vs. sum relators:

```
lemma rel_set_rel_sum[simp]:
"rel_set (rel_sum χ φ) A1 A2 ⟷
 rel_set χ (Inl -` A1) (Inl -` A2) ∧ rel_set φ (Inr -` A1) (Inr -` A2)"
(is "?L ⟷ ?Rl ∧ ?Rr")
proof safe
  assume L: "?L"
  show ?Rl unfolding rel_set_def Bex_def vimage_eq proof safe
    fix l1 assume "Inl l1 ∈ A1"
    then obtain a2 where a2: "a2 ∈ A2" and "rel_sum χ φ (Inl l1) a2"
    using L unfolding rel_set_def by auto
    then obtain l2 where "a2 = Inl l2 ∧ χ l1 l2" by (cases a2, auto)
    thus "∃ l2. Inl l2 ∈ A2 ∧ χ l1 l2" using a2 by auto
  next
    fix l2 assume "Inl l2 ∈ A2"
    then obtain a1 where a1: "a1 ∈ A1" and "rel_sum χ φ a1 (Inl l2)"
    using L unfolding rel_set_def by auto
    then obtain l1 where "a1 = Inl l1 ∧ χ l1 l2" by (cases a1, auto)
    thus "∃ l1. Inl l1 ∈ A1 ∧ χ l1 l2" using a1 by auto
  qed
  show ?Rr unfolding rel_set_def Bex_def vimage_eq proof safe
    fix r1 assume "Inr r1 ∈ A1"
    then obtain a2 where a2: "a2 ∈ A2" and "rel_sum χ φ (Inr r1) a2"
    using L unfolding rel_set_def by auto
    then obtain r2 where "a2 = Inr r2 ∧ φ r1 r2" by (cases a2, auto)
    thus "∃ r2. Inr r2 ∈ A2 ∧ φ r1 r2" using a2 by auto
  next
    fix r2 assume "Inr r2 ∈ A2"
    then obtain a1 where a1: "a1 ∈ A1" and "rel_sum χ φ a1 (Inr r2)"
    using L unfolding rel_set_def by auto
    then obtain r1 where "a1 = Inr r1 ∧ φ r1 r2" by (cases a1, auto)
    thus "∃ r1. Inr r1 ∈ A1 ∧ φ r1 r2" using a1 by auto
  qed
next
  assume Rl: "?Rl" and Rr: "?Rr"
  show ?L unfolding rel_set_def Bex_def vimage_eq proof safe
    fix a1 assume a1: "a1 ∈ A1"
    show "∃ a2. a2 ∈ A2 ∧ rel_sum χ φ a1 a2"
    proof(cases a1)
      case (Inl l1) then obtain l2 where "Inl l2 ∈ A2 ∧ χ l1 l2"
      using Rl a1 unfolding rel_set_def by blast
      thus ?thesis unfolding Inl by auto
    next
      case (Inr r1) then obtain r2 where "Inr r2 ∈ A2 ∧ φ r1 r2"
      using Rr a1 unfolding rel_set_def by blast
      thus ?thesis unfolding Inr by auto
    qed
  next
    fix a2 assume a2: "a2 ∈ A2"
    show "∃ a1. a1 ∈ A1 ∧ rel_sum χ φ a1 a2"
    proof(cases a2)
      case (Inl l2) then obtain l1 where "Inl l1 ∈ A1 ∧ χ l1 l2"
```

```
      using Rl a2 unfolding rel_set_def by blast
      thus ?thesis unfolding Inl by auto
    next
      case (Inr r2) then obtain r1 where "Inr r1 ∈ A1 ∧ φ r1 r2"
      using Rr a2 unfolding rel_set_def by blast
      thus ?thesis unfolding Inr by auto
    qed
  qed
qed
```

5.11.1 Countability

```
lemma exists_fset_of_list: "∃xs. fset_of_list xs = S"
including fset.lifting
by transfer (rule finite_list)

lemma fset_of_list_surj[simp, intro]: "surj fset_of_list"
proof -
  have "x ∈ range fset_of_list" for x :: "'a fset"
    unfolding image_iff
    using exists_fset_of_list by fastforce
  thus ?thesis by auto
qed

instance fset :: (countable) countable
proof
  obtain to_nat :: "'a list ⇒ nat" where "inj to_nat"
    by (metis ex_inj)
  moreover have "inj (inv fset_of_list)"
    using fset_of_list_surj by (rule surj_imp_inj_inv)
  ultimately have "inj (to_nat ∘ inv fset_of_list)"
    by (rule inj_comp)
  thus "∃to_nat::'a fset ⇒ nat. inj to_nat"
    by auto
qed
```

5.12 Quickcheck setup

Setup adapted from sets.

```
notation Quickcheck_Exhaustive.orelse (infixr "orelse" 55)

definition (in term_syntax) [code_unfold]:
"valterm_femptyset = Code_Evaluation.valtermify ({|||} :: ('a :: typerep) fset)"

definition (in term_syntax) [code_unfold]:
"valtermify_finsert x s = Code_Evaluation.valtermify finsert {·} (x :: ('a :: typerep * _)) {·} s"

instantiation fset :: (exhaustive) exhaustive
begin

fun exhaustive_fset where
"exhaustive_fset f i = (if i = 0 then None else (f {|||} orelse exhaustive_fset (λA. f A orelse Quickcheck_Exhaust
(λx. if x |∈| A then None else f (finsert x A)) (i - 1)) (i - 1)))"

instance ..

end

instantiation fset :: (full_exhaustive) full_exhaustive
begin
```

```
fun full_exhaustive_fset where
"full_exhaustive_fset f i = (if i = 0 then None else (f valterm_femptyset orelse full_exhaustive_fset
(λA. f A orelse Quickcheck_Exhaustive.full_exhaustive (λx. if fst x |∈| fst A then None else f (valtermify_finse
x A)) (i - 1)) (i - 1)))"

instance ..

end

no_notation Quickcheck_Exhaustive.orelse (infixr "orelse" 55)

notation scomp (infixl "○→" 60)

instantiation fset :: (random) random
begin

fun random_aux_fset :: "natural ⇒ natural ⇒ natural × natural ⇒ ('a fset × (unit ⇒ term)) × natural
× natural" where
"random_aux_fset 0 j = Quickcheck_Random.collapse (Random.select_weight [(1, Pair valterm_femptyset)])"
|
"random_aux_fset (Code_Numeral.Suc i) j =
  Quickcheck_Random.collapse (Random.select_weight
    [(1, Pair valterm_femptyset),
     (Code_Numeral.Suc i,
      Quickcheck_Random.random j ○→ (λx. random_aux_fset i j ○→ (λs. Pair (valtermify_finsert x s))))])"

lemma [code]:
  "random_aux_fset i j =
    Quickcheck_Random.collapse (Random.select_weight [(1, Pair valterm_femptyset),
      (i, Quickcheck_Random.random j ○→ (λx. random_aux_fset (i - 1) j ○→ (λs. Pair (valtermify_finsert
x s))))])"
proof (induct i rule: natural.induct)
  case zero
  show ?case by (subst select_weight_drop_zero[symmetric]) (simp add: less_natural_def)
next
  case (Suc i)
  show ?case by (simp only: random_aux_fset.simps Suc_natural_minus_one)
qed

definition "random_fset i = random_aux_fset i i"

instance ..

end

no_notation scomp (infixl "○→" 60)

end
```

5.13 Option Logic

This theory defines a three-valued logic such that nonsensical guard expressions cannot ever evaluate to true. Such expressions evaluate instead to None which, when negated, is still None.

```
theory Option_Logic
imports Main
begin

datatype trilean = true | false | invalid
```

```
fun maybe_not :: "trilean ⇒ trilean" ("¬? _" [60] 60) where
  "¬? true = false" |
  "¬? false = true" |
  "¬? invalid = invalid"

fun maybe_and :: "trilean ⇒ trilean ⇒ trilean" (infixl "∧?" 60) where
  "_ ∧? invalid = invalid" |
  "invalid ∧? _ = invalid" |
  "true ∧? true = true" |
  "_ ∧? false = false" |
  "false ∧? _ = false"

fun maybe_or :: "trilean ⇒ trilean ⇒ trilean" (infixl "∨?" 60) where
  "invalid ∨? _ = invalid" |
  "_ ∨? invalid = invalid" |
  "true ∨? _ = true" |
  "_ ∨? true = true" |
  "false ∨? false = false"


lemma maybe_and_associative: "a ∧? b ∧? c = a ∧? (b ∧? c)"
proof(induct a b arbitrary: c rule: maybe_or.induct)
case (1 uu)
  then show ?case
  proof -
    have "invalid ∧? (uu ∧? c) ≠ invalid ⟶ invalid ∧? (uu ∧? c) = invalid"
      by (metis (no_types) maybe_and.simps(1) maybe_and.simps(2) maybe_and.simps(3) trilean.exhaust)
    then show ?thesis
      by (metis (full_types) maybe_and.simps(1) maybe_and.simps(2) maybe_and.simps(3) trilean.exhaust)
  qed
next
  case "2_1"
  then show ?case
    by (metis (full_types) maybe_and.simps(1) maybe_and.simps(4) maybe_and.simps(5) maybe_not.cases)
next
  case "2_2"
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(2) maybe_and.simps(3) maybe_not.cases)
next
  case "3_1"
  then show ?case
    by (metis maybe_and.simps(4) maybe_and.simps(5) trilean.exhaust)
next
  case "3_2"
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(4) maybe_and.simps(5) maybe_not.cases)
next
  case 4
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(4) maybe_and.simps(5) maybe_and.simps(7) maybe_not.cases)
next
  case 5
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(6) trilean.exhaust)
qed

lemma maybe_and_commutative: "a ∧? b = b ∧? a"
  by (metis (full_types) maybe_and.simps(1) maybe_and.simps(2) maybe_and.simps(3) maybe_and.simps(5)
maybe_and.simps(7) trilean.distinct(5) trilean.exhaust)

lemma maybe_or_associative: "a ∨? b ∨? c = a ∨? (b ∨? c)"
```

```
proof(induct a b  arbitrary: c rule: maybe_or.induct)
case (1 uu)
  then show ?case
    by simp
next
  case "2_1"
  then show ?case
    by simp
next
  case "2_2"
  then show ?case
    by simp
next
  case "3_1"
  then show ?case
    by (metis maybe_not.cases maybe_or.simps(2) maybe_or.simps(4))
next
  case "3_2"
  then show ?case
    by (metis maybe_not.cases maybe_or.simps(3) maybe_or.simps(5) maybe_or.simps(6) maybe_or.simps(7))
next
  case 4
  then show ?case
    by (metis maybe_not.cases maybe_or.simps(2) maybe_or.simps(3) maybe_or.simps(4) maybe_or.simps(5)
maybe_or.simps(6))
next
  case 5
  then show ?case
    by (metis maybe_not.cases maybe_or.simps(6) maybe_or.simps(7))
qed

lemma maybe_or_commutative: "a ∨? b = b ∨? a"
proof(induct a b rule: maybe_or.induct)
  case (1 uu)
  then show ?case
    by (metis maybe_or.simps(1) maybe_or.simps(2) maybe_or.simps(3) trilean.exhaust)
next
  case "2_1"
  then show ?case
    by simp
next
  case "2_2"
  then show ?case
    by simp
next
  case "3_1"
  then show ?case
    by simp
next
  case "3_2"
  then show ?case
    by simp
next
  case 4
  then show ?case
    by simp
next
  case 5
  then show ?case
    by simp
qed
```

```
lemma trilean_distributivity: "a ∨? b ∧? c = a ∧? c ∨? (b ∧? c)"
proof(induct a b  arbitrary: c rule: maybe_or.induct)
case (1 uu)
  then show ?case
    by (metis maybe_and.simps(1) maybe_and_commutative maybe_or.simps(1))
next
case "2_1"
  then show ?case
    by (metis maybe_and.simps(1) maybe_and_commutative maybe_or.simps(1) maybe_or_commutative)
next
  case "2_2"
  then show ?case
    by (metis maybe_and.simps(1) maybe_and_commutative maybe_or.simps(1) maybe_or_commutative)
next
  case "3_1"
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(4) maybe_and.simps(7) maybe_and_commutative maybe_not.simps(1)
maybe_not.simps(3) maybe_or.simps(1) maybe_or.simps(4) maybe_or.simps(7) trilean.exhaust trilean.simps(2)
trilean.simps(6))
next
  case "3_2"
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(4) maybe_and.simps(6) maybe_and.simps(7) maybe_and_commutative
maybe_or.simps(1) maybe_or.simps(5) maybe_or.simps(7) trilean.exhaust trilean.simps(6))
next
  case 4
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(4) maybe_and.simps(6) maybe_and.simps(7) maybe_and_commutative
maybe_or.simps(1) maybe_or.simps(6) maybe_or.simps(7) trilean.exhaust)
next
case 5
  then show ?case
    by (metis maybe_and.simps(1) maybe_and.simps(6) maybe_and.simps(7) maybe_and_commutative maybe_or.simps(1)
maybe_or.simps(7) trilean.exhaust trilean.simps(6))
qed

instantiation trilean :: semiring begin
definition [simp]: "times_trilean = maybe_and"

definition [simp]: "plus_trilean = maybe_or"

instance
  apply standard
      apply (simp add: maybe_or_associative)
     apply (simp add: maybe_or_commutative)
    apply (simp add: maybe_and_associative)
   apply (simp add: trilean_distributivity)
  using maybe_and_commutative trilean_distributivity by auto
end

lemma maybe_or_idempotent: "a ∨? a = a"
  apply (cases a)
  by auto

lemma maybe_and_idempotent: "a ∧? a = a"
  apply (cases a)
  by auto

instantiation trilean :: ord begin
definition less_eq_trilean :: "trilean ⇒ trilean ⇒ bool" where
```

```
  "less_eq_trilean a b = (a + b = b)"

definition less_trilean :: "trilean ⇒ trilean ⇒ bool" where
  "less_trilean a b = (a ≤ b ∧ a ≠ b)"

declare less_trilean_def less_eq_trilean_def [simp]

instance
  by standard
end

lemma maybe_and_one: "true ∧? x = x"
  apply (cases x)
  by auto

lemma maybe_or_zero: "false ∨? x = x"
  apply (cases x)
  by auto

lemma maybe_double_negation: "¬? ¬? x = x"
  apply (cases x)
  by auto

lemma maybe_negate_true: "(¬? x = true) = (x = false)"
  apply (cases x)
  by auto

lemma maybe_and_true: "(x ∧? y = true) = (x = true ∧ y = true)"
  using maybe_and.elims by blast

lemma maybe_and_not_true: "(x ∧? y ≠ true) = (x ≠ true ∨ y ≠ true)"
  by (simp add: maybe_and_true)

lemma maybe_and_valid: "x ∧? y ≠ invalid ⟹ x ≠ invalid ∧ y ≠ invalid"
  using maybe_and.elims by blast

lemma maybe_or_valid: "x ∨? y ≠ invalid ⟹ x ≠ invalid ∧ y ≠ invalid"
  using maybe_or.elims by blast
end
theory Value
imports Option_Logic
begin
datatype "value" = Num int | Str String.literal

fun MaybeBoolInt :: "(int ⇒ int ⇒ bool) ⇒ value option ⇒ value option ⇒ trilean" where
  "MaybeBoolInt f (Some (Num a)) (Some (Num b)) = (if f a b then true else false)" |
  "MaybeBoolInt _ _ _ = invalid"

definition ValueGt :: "value option ⇒ value option ⇒ trilean"  where
  "ValueGt a b ≡ MaybeBoolInt (λx::int.λy::int.(x>y)) a b"

definition ValueLt :: "value option ⇒ value option ⇒ trilean"  where
  "ValueLt a b ≡ MaybeBoolInt (λx::int.λy::int.(x<y)) a b"

definition ValueEq :: "value option ⇒ value option ⇒ trilean"  where
  "ValueEq a b ≡ (if a = b then true else false)"

instantiation "value" :: linorder begin
fun less_eq_value :: "value ⇒ value ⇒ bool" where
  "less_eq_value (Num n) (Str s) = True" |
  "less_eq_value (Str s) (Num n) = False" |
```

```
  "less_eq_value (Str n) (Str s) = less_eq n s" |
  "less_eq_value (Num n) (Num s) = less_eq n s"

fun less_value :: "value ⇒ value ⇒ bool" where
  "less_value (Num n) (Str s) = True" |
  "less_value (Str s) (Num n) = False" |
  "less_value (Str n) (Str s) = less n s" |
  "less_value (Num n) (Num s) = less n s"

instance proof
  fix x y::"value"
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)"
  proof (induct x)
    case (Num x)
    then show ?case
      apply (cases y)
      by auto
  next
    case (Str x)
    then show ?case
      apply (cases y)
      by auto
  qed
  fix x :: "value"
  show "x ≤ x"
    apply (cases x)
    by auto
  fix x y z::"value"
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
  proof (induct x)
    case (Num n)
    then show ?case
    proof (induct y)
      case (Num x)
      then show ?case
        apply (cases z)
        by auto
    next
      case (Str x)
      then show ?case
        apply (cases z)
        by auto
    qed
  next
    case (Str s)
    then show ?case
    proof (induct y)
      case (Num x)
      then show ?case
        apply (cases z)
        by auto
    next
      case (Str x)
      then show ?case
        apply (cases z)
        by auto
    qed
  qed
next
  fix x y :: "value"
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
```

```
  proof (induct x)
    case (Num x)
    then show ?case
      apply (cases y)
      by auto
  next
    case (Str x)
    then show ?case
      apply (cases y)
      by auto
  qed
next
  fix x y::"value"
  show "x ≤ y ∨ y ≤ x"
  proof (induct x)
    case (Num x)
    then show ?case
      apply (cases y)
      by auto
  next
    case (Str x)
    then show ?case
      apply (cases y)
      by auto
  qed
qed
end

end
theory VName
imports Main
begin
datatype vname = I nat | R nat

instantiation vname :: linorder begin
fun less_eq_vname :: "vname ⇒ vname ⇒ bool" where
  "less_eq_vname (I n1) (R n2) = True" |
  "less_eq_vname (R n1) (I n2) = False" |
  "less_eq_vname (I n1) (I n2) = less_eq n1 n2" |
  "less_eq_vname (R n1) (R n2) = less_eq n1 n2"

fun less_vname :: "vname ⇒ vname ⇒ bool" where
  "less_vname (I n1) (R n2) = True" |
  "less_vname (R n1) (I n2) = False" |
  "less_vname (I n1) (I n2) = less n1 n2" |
  "less_vname (R n1) (R n2) = less n1 n2"

instance proof
  fix x y :: vname
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)"
  proof (induct x)
    case (I n)
    then show ?case
    proof (induct y)
      case (I m)
      then show ?case
        by auto
    next
      case (R m)
      then show ?case
        by simp
```

```
    qed
  next
    case (R n)
    then show ?case
    proof (induct y)
      case (I x)
      then show ?case
        by simp
    next
      case (R x)
      then show ?case
        by auto
    qed
  qed
next
  fix x :: vname
  show "x ≤ x"
  proof (induct x)
    case (I x)
    then show ?case
      by auto
  next
    case (R x)
    then show ?case
      by auto
  qed
next
  fix x y z::vname
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
  proof (induct x)
    case (I x)
    then show ?case
    proof (induct y)
      case (I xa)
      then show ?case
        apply (cases z)
        by auto
    next
      case (R xa)
      then show ?case
        apply (cases z)
        by auto
    qed
  next
    case (R x)
    then show ?case
    proof (induct y)
      case (I xa)
      then show ?case
        apply (cases z)
        by auto
    next
      case (R xa)
      then show ?case
        apply (cases z)
        by auto
    qed
  qed
next
  fix x y:: vname
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
```

```
  proof (induct x)
    case (I x)
    then show ?case
      apply (cases y)
      by auto
  next
    case (R x)
    then show ?case
      apply (cases y)
      by auto
  qed
next
  fix x y:: vname
  show "x ≤ y ∨ y ≤ x"
  proof (induct x)
    case (I x)
    then show ?case
      apply (cases y)
      by auto
  next
    case (R x)
    then show ?case
      apply (cases y)
      by auto
  qed
qed
end

end
```

6 Extended Finite State Machines

This section presents the theories associated with EFSMs. First we define a language of arithmetic expressions for guards, outputs, and updates similar to that in IMP [**?**]. We then go on to define the guard logic such that nonsensical guards (such as testing to see if an integer is greater than a string) can never evaluate to true. Next, the guard language is defined in terms of arithmetic expressions and binary relations. In the interest of simplifying the conversion of guards to constraints, we use a Nor logic, although we define syntax hacks for the expression of guards using other logical operators. With the underlying types defined, we then define EFSMs and prove that they are prefix-closed, that is to say that if a string of inputs is accepted by the machine then all of its prefixes are also accepted.

6.1 Arithmetic Expressions

This theory defines a language of arithmetic expressions over literal values and variables. Here, values are limited to integers and strings. Variables may be either inputs or registers. We also limit ourselves to a simple arithmetic of plus and minus as a proof of concept.

```
theory AExp
  imports Value VName
begin

type_synonym datastate = "vname ⇒ value option"

datatype aexp = L "value" | V vname | Plus aexp aexp | Minus aexp aexp

syntax (xsymbols)
  Plus :: "aexp ⇒ aexp ⇒ aexp"
  Minus :: "aexp ⇒ aexp ⇒ aexp"
```

```
fun value_plus :: "value option ⇒ value option ⇒ value option"  where
  "value_plus (Some (Num x)) (Some (Num y)) = Some (Num (x+y))" |
  "value_plus _ _ = None"

lemma plus_no_string [simp]:"value_plus a b ≠ Some (Str x)"
  using value_plus.elims by blast

lemma value_plus_symmetry: "value_plus x y = value_plus y x"
  proof (cases x)
    case None
    then show ?thesis by simp
  next
    case (Some a)
    then show ?thesis
      apply (cases y)
       apply simp
      apply (case_tac a)
       apply (case_tac aa)
        apply simp
       apply simp
      apply (case_tac aa)
      by simp_all
  qed

fun value_minus :: "value option ⇒ value option ⇒ value option"  where
  "value_minus (Some (Num x)) (Some (Num y)) = Some (Num (x-y))" |
  "value_minus _ _ = None"

lemma minus_no_string [simp]:"value_minus a b ≠ Some (Str x)"
  using value_minus.elims by blast

fun aval :: "aexp ⇒ datastate ⇒ value option" where
  "aval (L x) s = Some x" |
  "aval (V x) s = s x" |
  "aval (Plus a1 a2) s = value_plus (aval a1 s)(aval a2 s)" |
  "aval (Minus a1 a2) s = value_minus (aval a1 s) (aval a2 s)"

lemma aval_plus_symmetry: "aval (Plus x y) s = aval (Plus y x) s"
  by (simp add: value_plus_symmetry)

definition null_state ("<>") where
  "null_state ≡ λx. None"
declare null_state_def [simp]

syntax
  "_maplet"  :: "['a, 'a] ⇒ maplet"             ("_ /:=/ _")
  "_maplets" :: "['a, 'a] ⇒ maplet"             ("_ /[:=]/ _")
  "_Map"     :: "maplets ⇒ 'a ⇀ 'b"             ("(1<_>)")

instantiation aexp :: plus begin
fun plus_aexp :: "aexp ⇒ aexp ⇒ aexp" where
  "plus_aexp (L (Num n1)) (L (Num n2)) = L (Num (n1+n2))" |
  "plus_aexp x y = Plus x y"

instance by standard
end

instantiation aexp :: minus begin
fun minus_aexp :: "aexp ⇒ aexp ⇒ aexp" where
  "minus_aexp (L (Num n1)) (L (Num n2)) = L (Num (n1-n2))" |
  "minus_aexp x y = Minus x y"
```

```
instance by standard
end

lemma aval_plus_aexp: "aval (a+b) s = aval (Plus a b) s"
  apply (case_tac a)
     apply (case_tac x1)
      apply (case_tac b)
         apply (case_tac x1b)
  by auto

lemma aval_minus_aexp: "aval (a-b) s = aval (Minus a b) s"
  apply (case_tac a)
     apply (case_tac x1)
      apply (case_tac b)
         apply (case_tac x1b)
  by auto

fun aexp_constrains :: "aexp ⇒ aexp ⇒ bool" where
  "aexp_constrains (L l) a = (L l = a)" |
  "aexp_constrains (V v) v' = (V v = v')" |
  "aexp_constrains (Plus a1 a2) v = ((Plus a1 a2) = v ∨ (Plus a1 a2) = v ∨ (aexp_constrains a1 v ∨
aexp_constrains a2 v))" |
  "aexp_constrains (Minus a1 a2) v = ((Minus a1 a2) = v ∨ (aexp_constrains a1 v ∨ aexp_constrains a2
v))"

lemma constrains_implies_not_equal: "¬ aexp_constrains x a ⟹ x ≠ a"
  using aexp_constrains.elims(3) by blast

fun aexp_same_structure :: "aexp ⇒ aexp ⇒ bool" where
  "aexp_same_structure (L v) (L v') = True" |
  "aexp_same_structure (V v) (V v') = True" |
  "aexp_same_structure (Plus a1 a2) (Plus a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure
a2 a2')" |
  "aexp_same_structure (Minus a1 a2) (Minus a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure
a2 a2')" |
  "aexp_same_structure _ _ = False"

end
```

6.2 Guard Expressions

This theory defines the guard language of EFSMs which can be translated directly to and from contexts. This is similar to boolean expressions from IMP [**?**]. Boolean values true and false respectively represent the guards which are always and never satisfied. Guards may test for (in)equivalence of two arithmetic expressions or be connected using nor logic into compound expressions. Additionally, a guard may also test to see if a particular variable is null. This is useful if an EFSM transition is intended only to initialise a register. We also define syntax hacks for the relations less than, less than or equal to, greater than or equal to, and not equal to as well as the expression of logical conjunction, disjunction, and negation in terms of nor logic.

```
theory GExp
imports AExp Option_Logic
begin

datatype gexp = Bc bool | Eq aexp aexp | Gt aexp aexp | Nor gexp gexp | Null aexp
```

syntax *(xsymbols)*
  Eq :: "aexp ⇒ aexp ⇒ gexp"
  Gt :: "aexp ⇒ aexp ⇒ gexp"

fun *gval :: "gexp ⇒ datastate ⇒ trilean"* **where**
  "gval (Bc True) _ = true" |
  "gval (Bc False) _ = false" |
  "gval (Gt a₁ a₂) s = ValueGt (aval a₁ s) (aval a₂ s)" |
  "gval (Eq a₁ a₂) s = ValueEq (aval a₁ s) (aval a₂ s)" |
  "gval (Nor a₁ a₂) s = ¬? ((gval a₁ s) ∨? (gval a₂ s))" |
  "gval (Null v) s = ValueEq (aval v s) None"

abbreviation *gNot :: "gexp ⇒ gexp"* **where**
  "gNot g ≡ Nor g g"

abbreviation *gOr :: "gexp ⇒ gexp ⇒ gexp"* **where**
  "gOr v va ≡ Nor (Nor v va) (Nor v va)"

abbreviation *gAnd :: "gexp ⇒ gexp ⇒ gexp"* **where**
  "gAnd v va ≡ Nor (Nor v v) (Nor va va)"

lemma *inj_gAnd: "inj gAnd"*
  apply *(simp add: inj_def)*
  apply *clarify*
  by *(metis gexp.inject(4))*

lemma *gAnd_determinism: "(gAnd x y = gAnd x' y') = (x = x' ∧ y = y')"*
proof
  show *"gAnd x y = gAnd x' y' ⟹ x = x' ∧ y = y'"*
    by *(simp)*
next
  show *"x = x' ∧ y = y' ⟹ gAnd x y = gAnd x' y' "*
    by *simp*
qed

abbreviation *Lt :: "aexp ⇒ aexp ⇒ gexp"* **where**
  "Lt a b ≡ Gt b a"

abbreviation *Le :: "aexp ⇒ aexp ⇒ gexp"* **where**
  "Le v va ≡ gNot (Gt v va)"

abbreviation *Ge :: "aexp ⇒ aexp ⇒ gexp"* **where**
  "Ge v va ≡ gNot (Lt v va)"

abbreviation *Ne :: "aexp ⇒ aexp ⇒ gexp"* **where**
  "Ne v va ≡ gNot (Eq v va)"

lemma *or_equiv: "gval (gOr x y) r = (gval x r) ∨? (gval y r)"*
  by *(simp add: maybe_double_negation maybe_or_idempotent)*

lemma *not_equiv: "gval (gNot x) s = ¬? (gval x s)"*
  by *(simp add: maybe_or_idempotent)*

lemma *nor_equiv: "gval (gNot (gOr a b)) s = gval (Nor a b) s"*
  by *(metis maybe_double_negation not_equiv)*

definition *satisfiable :: "gexp ⇒ bool"* **where**
  "satisfiable g ≡ (∃s. gval g s = true)"

definition *gexp_valid :: "gexp ⇒ bool"* **where**

```
  "gexp_valid g ≡ (∀s. gval g s = true)"

definition gexp_equiv :: "gexp ⇒ gexp ⇒ bool" where
  "gexp_equiv a b ≡ ∀s. gval a s = gval b s"

lemma gexp_equiv_reflexive: "gexp_equiv x x"
  by (simp add: gexp_equiv_def)

lemma gexp_equiv_symmetric: "gexp_equiv x y ⟹ gexp_equiv y x"
  by (simp add: gexp_equiv_def)

lemma gexp_equiv_transitive: "gexp_equiv x y ∧ gexp_equiv y z ⟹ gexp_equiv x z"
  by (simp add: gexp_equiv_def)

lemma gval_subst: "gexp_equiv x y ⟹ P (gval x s) ⟹ P (gval y s)"
  by (simp add: gexp_equiv_def)

lemma gexp_equiv_satisfiable: "gexp_equiv x y ⟹ satisfiable x = satisfiable y"
  by (simp add: gexp_equiv_def satisfiable_def)

lemma gAnd_reflexivity: "gexp_equiv (gAnd x x) x"
  by (simp add: gexp_equiv_def maybe_double_negation maybe_or_idempotent)

lemma gAnd_zero: "gexp_equiv (gAnd (Bc True) x) x"
  apply (simp add: gexp_equiv_def)
  apply (rule allI)
  apply (case_tac "gval x s")
  by simp_all

lemma gAnd_symmetry: "gexp_equiv (gAnd x y) (gAnd y x)"
  by (simp add: gexp_equiv_def maybe_or_commutative)

lemma satisfiable_gAnd_self: "satisfiable (gAnd x x) = satisfiable x"
  by (simp add: gAnd_reflexivity gexp_equiv_satisfiable)

definition mutually_exclusive :: "gexp ⇒ gexp ⇒ bool" where
  "mutually_exclusive x y = (∀i. (gval x i = true ⟶ gval y i ≠ true) ∧
                                 (gval y i = true ⟶ gval x i ≠ true))"

lemma mutually_exclusive_unsatisfiable_conj: "mutually_exclusive x y = (¬ satisfiable (gAnd x y))"
  apply (simp add: mutually_exclusive_def satisfiable_def)
  by (metis maybe_double_negation maybe_not.simps(2) maybe_or_associative maybe_or_commutative maybe_or_idempoten
maybe_or_zero)

lemma unsatisfiable_conj_mutually_exclusive: "¬ satisfiable (gAnd x y) = mutually_exclusive x y"
  by (simp add: mutually_exclusive_unsatisfiable_conj)

lemma mutually_exclusive_reflexive: "satisfiable x ⟹ ¬ mutually_exclusive x x"
  by (simp add: mutually_exclusive_def satisfiable_def)

lemma mutually_exclusive_symmetric: "mutually_exclusive x y ⟹ mutually_exclusive y x"
  by (simp add: mutually_exclusive_def)

lemma not_mutually_exclusive_true: "satisfiable x = (¬ mutually_exclusive x (Bc True))"
  by (simp add: mutually_exclusive_def satisfiable_def)

lemma gval_gAnd: "gval (gAnd g1 g2) s = (gval g1 s) ∧? (gval g2 s)"
proof(induct "gval g1 s" "gval g2 s" rule: maybe_and.induct)
case 1
  then show ?case
    by (metis gval.simps(5) maybe_and_valid maybe_not.simps(3) maybe_or_valid)
```

```
next
  case "2_1"
  then show ?case
    by simp
next
  case "2_2"
  then show ?case
    by simp
next
  case 3
  then show ?case
    by simp
next
  case "4_1"
  then show ?case
    by simp
next
  case "4_2"
  then show ?case
    by simp
next
  case 5
  then show ?case
    by simp
qed

lemma gval_gAnd_True: "(gval (gAnd g1 g2) s = true) = ((gval g1 s = true) ∧ gval g2 s = true)"
  using gval_gAnd maybe_and_not_true by fastforce

declare gval.simps [simp del]

fun gexp_constrains :: "gexp ⇒ aexp ⇒ bool" where
  "gexp_constrains (gexp.Bc _) _ = False" |
  "gexp_constrains (Null a) v = aexp_constrains a v" |
  "gexp_constrains (gexp.Eq a1 a2) v = (aexp_constrains a1 v ∨ aexp_constrains a2 v)" |
  "gexp_constrains (gexp.Gt a1 a2) v = (aexp_constrains a1 v ∨ aexp_constrains a2 v)" |
  "gexp_constrains (gexp.Nor g1 g2) v = (gexp_constrains g1 v ∨ gexp_constrains g2 v)"

fun contains_bool :: "gexp ⇒ bool" where
  "contains_bool (Bc _) = True" |
  "contains_bool (Nor g1 g2) = (contains_bool g1 ∨ contains_bool g2)" |
  "contains_bool _ = False"

fun gexp_same_structure :: "gexp ⇒ gexp ⇒ bool" where
  "gexp_same_structure (gexp.Bc b) (gexp.Bc b') = (b = b')" |
  "gexp_same_structure (gexp.Eq a1 a2) (gexp.Eq a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure
a2 a2')" |
  "gexp_same_structure (gexp.Gt a1 a2) (gexp.Gt a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure
a2 a2')" |
  "gexp_same_structure (gexp.Nor g1 g2) (gexp.Nor g1' g2') = (gexp_same_structure g1 g1' ∧ gexp_same_structure
g2 g2')" |
  "gexp_same_structure (gexp.Null a1) (gexp.Null a2) = aexp_same_structure a1 a2" |
  "gexp_same_structure _ _ = False"

end
theory Transition
imports GExp
begin

type_synonym label = String.literal
type_synonym arity = nat
```

```
type_synonym inputs = "value list"
type_synonym outputs = "value option list"
type_synonym guard = "gexp"
type_synonym output_function = "aexp"
type_synonym update_function = "(vname × aexp)"
type_synonym updates = "update_function list"

record transition =
  Label :: label
  Arity :: nat
  Guard :: "guard list"
  Outputs :: "output_function list"
  Updates :: "update_function list"

lemma transition_equality: "((x::transition) = y) = ((Label x) = (Label y) ∧
                                (Arity x) = (Arity y) ∧
                                (Guard x) = (Guard y) ∧
                                (Outputs x) = (Outputs y) ∧
                                (Updates x) = (Updates y))"
proof
  fix x y :: transition
  assume "x = y"
  show "Label x = Label y ∧ Arity x = Arity y ∧ Guard x = Guard y ∧ Outputs x = Outputs y ∧ Updates
x = Updates y"
    by (simp add: ‹x = y›)
next
  fix x y :: transition
  assume "Label x = Label y ∧ Arity x = Arity y ∧ Guard x = Guard y ∧ Outputs x = Outputs y ∧ Updates
x = Updates y"
  show "x = y"
    by (simp add: ‹Label x = Label y ∧ Arity x = Arity y ∧ Guard x = Guard y ∧ Outputs x = Outputs
y ∧ Updates x = Updates y›)
qed

lemma unequal_labels[simp]: "Label t1 ≠ Label t2 ⟹ t1 ≠ t2"
  by auto

lemma unequal_arities[simp]: "Arity t1 ≠ Arity t2 ⟹ t1 ≠ t2"
  by auto

definition same_structure :: "transition ⇒ transition ⇒ bool" where
  "same_structure t1 t2 = (Label t1 = Label t2 ∧
                           Arity t1 = Arity t2 ∧
                           list_all (λ(g1, g2). gexp_same_structure g1 g2) (zip (Guard t1) (Guard t2)))"

end
theory FSet_Utils
  imports "~~/src/HOL/Library/FSet"
begin

context includes fset.lifting begin
lift_definition fprod  :: "'a fset ⇒ 'b fset ⇒ ('a × 'b) fset " (infixr "|×|" 80) is "λa b. fset a ×
fset b"
  by simp

lift_definition fis_singleton :: "'a fset ⇒ bool" is "λA. is_singleton (fset A)".
end

lemma fprod_subset: "x |⊆| x' ∧ y |⊆| y' ⟹ x |×| y |⊆| x' |×| y'"
  apply (simp add: fprod_def less_eq_fset_def Abs_fset_inverse)
  by auto
```

lemma *fimage_fprod: "(a, b) |∈| A |×| B ⟹ f a b |∈| (λ(x, y). f x y) |‘| (A |×| B)"*
by *force*

lemma *fprod_singletons: "{|a|} |×| {|b|} = {|(a, b)|}"*
apply *(simp add: fprod_def)*
by *(metis fset_inverse fset_simps(1) fset_simps(2))*

lemma *fset_both_sides: "(Abs_fset s = f) = (fset (Abs_fset s) = fset f)"*
by *(simp add: fset_inject)*

lemma *Abs_ffilter: "(ffilter f s = s') = (Set.filter f (fset s) = (fset s'))"*
by *(simp add: ffilter_def fset_both_sides Abs_fset_inverse)*

lemma *ffilter_empty: "ffilter f {||} = {||}"*
apply *(simp add: ffilter_def fset_both_sides Abs_fset_inverse)*
by *auto*

lemma *ffilter_finsert: "ffilter f (finsert a s) = (if f a then finsert a (ffilter f s) else (ffilter f s))"*
apply *simp*
apply *standard*
apply *(simp add: ffilter_def fset_both_sides Abs_fset_inverse)*
apply *auto[1]*
apply *(simp add: ffilter_def fset_both_sides Abs_fset_inverse)*
by *auto*

lemma *singleton_singleton [simp]: "fis_singleton {|a|}"*
by *(simp add: fis_singleton_def)*

lemma *not_singleton_emty [simp]: "¬fis_singleton {||}"*
apply *(simp add: fis_singleton_def)*
by *(simp add: is_singleton_altdef)*

lemma *abs_fset_fiveton[simp]: "Abs_fset {a, b, c, d, e} = {|a, b, c, d, e|}"*
by *(metis bot_fset.rep_eq finsert.rep_eq fset_inverse)*

lemma *abs_fset_fourton[simp]: "Abs_fset {a, b, c, d} = {|a, b, c, d|}"*
by *(metis bot_fset.rep_eq finsert.rep_eq fset_inverse)*

lemma *abs_fset_tripleton[simp]: "Abs_fset {a, b, c} = {|a, b, c|}"*
by *(metis bot_fset.rep_eq finsert.rep_eq fset_inverse)*

lemma *abs_fset_doubleton[simp]: "Abs_fset {a, b} = {|a, b|}"*
by *(metis bot_fset.rep_eq finsert.rep_eq fset_inverse)*

lemma *abs_fset_singleton[simp]: "Abs_fset {a} = {|a|}"*
by *(metis bot_fset.rep_eq finsert.rep_eq fset_inverse)*

lemma *abs_fset_empty[simp]: "Abs_fset {} = {||}"*
by *(simp add: bot_fset_def)*

lemma *fprod_empty[simp]: "∀a. fprod {||} a = {||}"*
by *(simp add: fprod_def)*

lemma *fprod_empty_2[simp]: "∀a. fprod a {||} = {||}"*
by *(simp add: fprod_def ffUnion_def)*

lemma *set_equiv: "(f1 = f2) = (fset f1 = fset f2)"*
by *(simp add: fset_inject)*

```
lemma fprod_equiv: "(fset (f |×| f') = s) = (((fset f) × (fset f')) = s)"
  by (simp add: fprod_def Abs_fset_inverse)

lemma finsert_equiv: "(finsert e f = f') = (insert e (fset f) = (fset f'))"
  by (simp add: finsert_def fset_both_sides Abs_fset_inverse)

lemma filter_elements: "x |∈| Abs_fset (Set.filter f (fset s)) = (x ∈ (Set.filter f (fset s)))"
  by (metis ffilter.rep_eq fset_inverse notin_fset)

lemma singleton_equiv: "is_singleton s ⟹ (the_elem s = i) = (s = {i})"
  by (meson is_singleton_the_elem the_elem_eq)

lemma sorted_list_of_empty [simp]: "sorted_list_of_fset {|||} = []"
  by (simp add: sorted_list_of_fset_def)

lemma fmember_implies_member: "e |∈| f ⟹ e ∈ fset f"
  by (simp add: fmember_def)

lemma ffilter_to_filter: "(ffilter f s = s') = (Set.filter f (fset s) = fset s')"
  by (metis ffilter.rep_eq fset_inject)

lemma fold_union_ffUnion: "fold (|∪|) l {|||} = ffUnion (fset_of_list l)"
proof(induct l rule: rev_induct)
case Nil
  then show ?case by simp
next
  case (snoc a l)
  then show ?case
    by simp
qed

lemma filter_filter: "ffilter P (ffilter Q xs) = ffilter (λx. Q x ∧ P x) xs"
  by auto
end
```

6.3 Extended Finite State Machines

This theory defines extended finite state machines. Each EFSM takes a type variable which represents S. This is a slight devaition from the definition presented in [**?**] as this type variable may be of an infinite type such as integers, however the intended use is for custom finite types. See the examples for details.

```
theory EFSM
  imports "~~/src/HOL/Library/FSet" Transition FSet_Utils
begin

type_synonym event = "(label × inputs)"
type_synonym trace = "event list"
type_synonym observation = "outputs list"
type_synonym transition_matrix = "((nat × nat) × transition) fset"

abbreviation Str :: "string ⇒ value" where
  "Str s ≡ value.Str (String.implode s)"

primrec input2state :: "value list ⇒ nat ⇒ datastate" where
  "input2state [] _ = <>" |
  "input2state (h#t) i = (λx. if x = I i then Some h else (input2state t (i+1)) x)"

lemma hd_input2state: "length i ≥ 1 ⟹ input2state i 1 (I 1) = Some (hd i)"
  by (metis hd_Cons_tl input2state.simps(2) le_numeral_extra(2) length_0_conv)

definition join_ir :: "value list ⇒ datastate ⇒ datastate" where
```

```
  "join_ir i r ≡ (λx. case x of
    R n ⇒ r (R n) |
    I n ⇒ (input2state i 1) (I n)
  )"
declare join_ir_def [simp]

definition S :: "transition_matrix ⇒ nat fset" where
  "S m = (fimage (λ((s, s'), t). s) m) |∪| fimage (λ((s, s'), t). s') m"

primrec apply_outputs :: "output_function list ⇒ datastate ⇒ outputs" where
  "apply_outputs [] _ = []" |
  "apply_outputs (h#t) s = (aval h s)#(apply_outputs t s)"

lemma apply_outputs_alt: "apply_outputs p s = map (λp. aval p s) p"
proof(induct p)
  case Nil
  then show ?case by simp
next
  case (Cons a p)
  then show ?case
    by simp
qed

lemma apply_outputs_preserves_length: "length (apply_outputs p s) = length p"
  by (simp add: apply_outputs_alt)

primrec apply_guards :: "guard list ⇒ datastate ⇒ bool" where
  "apply_guards [] _ = True" |
  "apply_guards (h#t) s =  ((gval h s) = true ∧ (apply_guards t s))"

lemma apply_guards_alt: "apply_guards G s = (∀g ∈ set (map (λg. gval g s) G). g = true)"
proof(induct G)
case Nil
  then show ?case
    by simp
next
  case (Cons a G)
  then show ?case
    by simp
qed

primrec apply_updates :: "(vname × aexp) list ⇒ datastate ⇒ datastate ⇒ datastate" where
  "apply_updates [] _ new = new" |
  "apply_updates (h#t) old new = (λx. if x = (fst h) then (aval (snd h) old) else (apply_updates t old
new) x)"

definition possible_steps :: "transition_matrix ⇒ nat ⇒ datastate ⇒ label ⇒ inputs ⇒ (nat × transition)
fset" where
  "possible_steps e s r l i = fimage (λ((origin, dest), t). (dest, t)) (ffilter (λ((origin, dest::nat),
t::transition). origin = s ∧ (Label t) = l ∧ (length i) = (Arity t) ∧ apply_guards (Guard t) (join_ir
i r)) e)"

lemma possible_steps_alt_aux: "(λ((origin, dest), t). (dest, t)) |`|
    ffilter
     (λ((origin, dest), t).
         origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guard t) (λx. case x of I n
⇒ input2state i 1 (I n) | R n ⇒ r (R n)))
     e =
    {|(d, t)|} ⟹
    ffilter
     (λ((origin, dest), t).
```

```
        origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guard t) (λx. case x of I n
⇒ input2state i 1 (I n) | R n ⇒ r (R n)))
     e =
    {|((s, d), t)|}"
proof(induct e)
  case empty
  then show ?case
    apply (simp add: ffilter_empty)
    by auto
next
  case (insert x e)
  then show ?case
    apply (cases x)
    apply (case_tac a)
    apply clarify
    apply simp
    apply (simp add: ffilter_finsert)
    apply (case_tac "aa = s")
     apply simp
     apply (case_tac "Label ba = l")
      apply simp
      apply (case_tac "length i = Arity ba")
       apply simp
       apply (case_tac "apply_guards (Guard ba) (case_vname (λn. input2state i (Suc 0) (I n)) (λn. r
(R n)))")
    by auto
qed

lemma possible_steps_alt: "(possible_steps e s r l i = {|(d, t)|}) = (ffilter
     (λ((origin, dest), t).
        origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guard t) (λx. case x of I n
⇒ input2state i 1 (I n) | R n ⇒ r (R n)))
     e =
    {|((s, d), t)|})"
  apply standard
   apply (simp add: possible_steps_def possible_steps_alt_aux)
  by (simp add: possible_steps_def)

lemma possible_steps_singleton: "(ffilter
     (λ((origin, dest), t).
        origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guard t) (λx. case x of I n
⇒ input2state i 1 (I n) | R n ⇒ r (R n)))
     e =
    {|((s, d), t)|}) ⟹ (possible_steps e s r l i = {|(d, t)|})"
  by (simp add: possible_steps_alt)

lemma singleton_dest: "fis_singleton (possible_steps e s r aa b) ⟹
       fthe_elem (possible_steps e s r aa b) = (baa, aba) ⟹
       ((s, baa), aba) |∈| e"
  apply (simp add: fis_singleton_def fthe_elem_def singleton_equiv)
  apply (simp add: possible_steps_def fmember_def)
  by auto

definition step :: "transition_matrix ⇒ nat ⇒ datastate ⇒ label ⇒ inputs ⇒ (transition × nat ×
outputs × datastate) option" where
"step e s r l i = (let possibilities = possible_steps e s r l i in
                   if possibilities = {||} then None
                   else
                     let (s', t) = Eps (λx. x |∈| possibilities) in
                     Some (t, s', (apply_outputs (Outputs t) (join_ir i r)), (apply_updates (Updates
t) (join_ir i r) r))
```

```
    )"

lemma no_possible_steps: "possible_steps e s r l i = {||} ⟹ step e s r l i = None"
  by (simp add: step_def)

lemma one_possible_step: "possible_steps e s r l i = {|(s', t)|} ⟹
      apply_outputs (Outputs t) (join_ir i r) = p ⟹
      apply_updates (Updates t) (join_ir i r) r = u ⟹
      step e s r l i = Some (t, s', p, u)"
  apply (simp add: step_def)
  apply standard
  using One_nat_def apply presburger
  using One_nat_def by presburger

lemma step_empty[simp]:"step {||} s r l i = None"
proof-
  have ffilter_empty: "ffilter
      (λ((origin, dest), t).
          origin = s ∧
          Label t = l ∧ length i = Arity t ∧ apply_guards (Guard t) (case_vname (λn. input2state i
1 (I n)) (λn. r (R n))))
      {||} = {||}"
    by auto
  show ?thesis
    by (simp add: step_def possible_steps_def ffilter_empty)
qed

primrec observe_all :: "transition_matrix ⇒ nat ⇒ datastate ⇒ trace ⇒ (transition × nat × outputs
× datastate) list" where
  "observe_all _ _ _ [] = []" |
  "observe_all e s r (h#t) =
    (case (step e s r (fst h) (snd h)) of
      (Some (transition, s', outputs, updated)) ⇒ (((transition, s', outputs, updated)#(observe_all
e s' updated t))) |
      _ ⇒ []
    )"

definition state :: "(transition × nat × outputs × datastate) ⇒ nat" where
  "state x ≡ fst (snd x)"

definition observe_trace :: "transition_matrix ⇒ nat ⇒ datastate ⇒ trace ⇒ observation" where
  "observe_trace e s r t ≡ map (λ(t,x,y,z). y) (observe_all e s r t)"

lemma observe_trace_step: "lst ≠ [] ⟹
      step e s r (fst (hd lst)) (snd (hd lst)) = Some (t, s', p, r') ⟹
      observe_trace e s' r' (tl lst) = obs ⟹
      observe_trace e s r lst = p#obs"
proof(induct lst)
  case Nil
  then show ?case by simp
next
  case (Cons a lst)
  then show ?case
    by (simp add: observe_trace_def)
qed


lemma observe_empty: "t = [] ⟹ observe_trace e 0 <> t = []"
  by (simp add: observe_trace_def)

definition state_trace :: "transition_matrix ⇒ nat ⇒ datastate ⇒ trace ⇒ nat list" where
```

```
  "state_trace e s r t ≡ map (λ(t,x,y,z). x) (observe_all e s r t)"

definition transition_trace :: "transition_matrix ⇒ nat ⇒ datastate ⇒ trace ⇒ transition list" where
  "transition_trace e s r t ≡ map (λ(t,x,y,z). t) (observe_all e s r t)"

definition efsm_equiv :: "transition_matrix ⇒ transition_matrix ⇒ trace ⇒ bool" where
  "efsm_equiv e1 e2 t ≡ ((observe_trace e1 0 <> t) = (observe_trace e2 0 <> t))"

lemma efsm_equiv_reflexive: "efsm_equiv e1 e1 t"
  by (simp add: efsm_equiv_def)

lemma efsm_equiv_symmetric: "efsm_equiv e1 e2 t ≡ efsm_equiv e2 e1 t"
  apply (simp add: efsm_equiv_def)
  by argo

lemma efsm_equiv_transitive: "efsm_equiv e1 e2 t ∧ efsm_equiv e2 e3 t ⟶ efsm_equiv e1 e3 t"
  by (simp add: efsm_equiv_def)

lemmas observations = observe_trace_def step_def possible_steps_def

lemma different_observation_techniques: "length(observe_all e s r t) = length(observe_trace e s r t)"
  by (simp add: observe_trace_def)

lemma length_observe_all_restricted: "⋀s r. length (observe_all e s r t) ≤ length t"
proof (induction t)
  case Nil
  then show ?case by simp
next
  case (Cons a t)
  then show ?case
  proof cases
    assume "step e s r (fst a) (snd a) = None"
    then show ?thesis by simp
  next
    assume "step e s r (fst a) (snd a) ≠  None"
    with Cons show ?thesis by(auto)
  qed
qed

inductive accepts :: "transition_matrix ⇒ nat ⇒ datastate ⇒ trace ⇒ bool" where
  base: "accepts e s d []" |
  step: "step e s d (fst h) (snd h) = Some (tr, s', p', d') ⟹ accepts e s' d' t ⟹ accepts e s d
(h#t)"

definition accepts_trace :: "transition_matrix ⇒ trace ⇒ bool" where
  "accepts_trace e t = accepts e 0 <> t"

lemma no_step_none: "step e s r aa ba = None ⟹ ¬accepts e s r ((aa, ba) # p)"
  apply safe
  apply (rule accepts.cases)
    apply simp
   apply simp
  by auto

lemma inaccepts_conditions: "¬accepts e s d (h # t) ⟹ step e s d (fst h) (snd h) = None ∨ (∃tr
s' p' d'. step e s d (fst h) (snd h) =  Some (tr, s', p', d') ∧ ¬accepts e s' d' t)"
  apply (rule accepts.cases)
  using accepts.base
    apply auto[1]
   apply (metis option.exhaust prod_cases4 accepts.step)
  by simp
```

```
lemma step_none_inaccepts: "((step e s d (fst h) (snd h)) = None) ⟹ ¬ (accepts e s d (h#t))"
  apply(clarify)
  apply(cases rule:accepts.cases)
    apply(simp)
   apply simp
  by(auto)

lemma inaccepts_future_inaccepts: "(∃tr s' p' d'. step e s d (fst h) (snd h) =  Some (tr, s', p', d')
∧ ¬accepts e s' d' t) ⟹ ¬accepts e s d (h#t)"
  apply clarify
    apply(cases rule: accepts.cases)
    apply simp
   apply simp
  by auto

lemma conditions_inaccepts: "step e s d (fst h) (snd h) = None ∨ (∃tr s' p' d'. step e s d (fst h)
(snd h) =  Some (tr, s', p', d') ∧ ¬accepts e s' d' t) ⟹ ¬ accepts e s d (h # t)"
  apply clarify
    apply(cases rule:accepts.cases)
    apply simp
   apply simp
  by auto

lemma accepts_head: "accepts e s d (h#t) ⟹ accepts e s d [h]"
  by (meson base conditions_inaccepts inaccepts_conditions)

lemma inaccepts_single_event: "¬ accepts e s d [(a, b)] ⟹ step e s d (fst (a, b)) (snd (a, b)) =
None"
  by (metis (mono_tags, lifting) base inaccepts_conditions)

lemma step_inaccepts: "¬ accepts e s d ((a, b) # t) ⟹ step e s d (fst (a, b)) (snd (a, b)) = Some
(tr, s', p', d') ⟹ ¬ accepts e s' d' t"
  using inaccepts_conditions by force

lemma step_none_inaccepts_append: "step e s d (fst a) (snd a) = None ⟹ ¬accepts e s d (a # t) ∧
¬accepts e s d (a # t @ t')"
  by (simp add: step_none_inaccepts)

lemma step_some: "step e s d (fst a) (snd a) = Some (tr, aa, ab, b) ⟹ accepts e s d (a # t) = accepts
e aa b t"
  apply safe
  using conditions_inaccepts apply fastforce
  by (simp add: accepts.step)

lemma aux1: "∀ s d. accepts e s d (t@t') ⟶ accepts e s d t"
  proof (induction t)
    case Nil
    then show ?case by (simp add: base)
  next
    case (Cons a t)
    then show ?case
      apply safe
      apply simp
      apply (case_tac "step e s d (fst a) (snd a) = None")
       apply (simp add: step_none_inaccepts)
      apply safe
      by (simp add: step_some)
  qed

lemma prefix_closure: "accepts e s d (t@t') ⟹ accepts e s d t"
```

```
  proof (induction "t")
    case Nil
    then show ?case by (simp add: base)
  next
    case (Cons x xs)
    then show ?case
      apply simp
      apply (case_tac "step e s d (fst x) (snd x) = None")
       apply (simp add: step_none_inaccepts)
      apply safe
      apply (simp add: step_some)
      using aux1 by force
  qed

lemma inaccepts_prefix: "¬accepts e s d t ⟹ ¬accepts e s d (t@t')"
  apply (rule ccontr)
  by (simp add: prefix_closure)

lemma length_observe_empty_trace: "length (observe_all e aa b []) = 0"
  by simp

lemma step_length_suc: "step e 0 <> (fst a) (snd a) = Some (tr, aa, ab, b) ⟹ length (observe_all
e 0 <> (a # t)) = Suc (length (observe_all e aa b t))"
  by simp

lemma accepts_trace_obs_equal_length: "accepts e 0 <> t ⟹ (length t = length (observe_all e 0 <>
t))"
  proof (induction t rule: accepts.induct)
    case (base e s d)
    then show ?case
      by simp
  next
    case (step e s d h tr s' p' d' t)
    then show ?case
      by simp
  qed

lemma aux3: "∀s d. (length t = length (observe_all e s d t)) ⟶ accepts e s d t"
  proof (induction t)
    case Nil
    then show ?case by (simp add: accepts.base)
  next
    case (Cons a t)
    then show ?case
      apply safe
      apply simp
      apply (case_tac "step e s d (fst a) (snd a)")
       apply simp
      apply simp
      apply (case_tac aa)
      apply simp
      by (simp only: step_length_suc step_some)
  qed

inductive gets_us_to :: "nat ⇒ transition_matrix ⇒ nat ⇒ datastate ⇒ trace ⇒ bool" where
  base: "s = target ⟹ gets_us_to target _ s _ []" |
  step_some: "step e s r (fst h) (snd h) =  Some (_, s', _, r') ⟹ gets_us_to target e s' r' t ⟹
gets_us_to target e s r (h#t)" |
  step_none: "step e s r (fst h) (snd h) = None ⟹ s=target ⟹ gets_us_to target e s r (h#t)"

lemma no_further_steps: "s ≠ s' ⟹ ¬ gets_us_to s e s' r []"
```

```
  apply safe
  apply (rule gets_us_to.cases)
  by auto

definition incoming_transition_to :: "transition_matrix ⇒ nat ⇒ bool" where
  "incoming_transition_to t s = ((ffilter (λ((from, to), t). to = s) t) ≠ {||})"

lemma incoming_transition_alt_def: "incoming_transition_to e n = (∃t from. ((from, n), t) |∈| e)"
  apply (simp add: incoming_transition_to_def)
  apply (simp add: ffilter_def fset_both_sides Abs_fset_inverse)
  apply (simp add: fmember_def)
  apply (simp add: Set.filter_def)
  by auto

end
```

7 Subsumption and Generalisation

We now define a language of constraint expressions to express restrictions on the known values of registers which can be grouped into *contexts* which are used to extend the idea of transition subsumption [**?**] to transitions with update functions. This forms the underpinning of an EFSM inference technique based on transition merging.

7.1 Constraint Expressions

This theory defines a language to express constraints on register values. Base restrictions are undefined, unrestricted, inconsistent, equal to a value, less than a value, greater than a value. Expressions may be combined using either negation or conjunction to form compound expressions. We also define syntax hacks for the relations less than or equal to, greater than or equal to, and not equal to as well as the expression of logical "or" in terms of negation and conjunction.

```
theory CExp
  imports AExp Option_Logic GExp
begin

datatype cexp = Undef | Bc bool | Eq "value" | Lt "value" | Gt "value" | Not cexp | And cexp cexp

fun "and" :: "cexp ⇒ cexp ⇒ cexp" where
  "and (Bc True) x = x" |
  "and x (Bc True) = x" |
  "and c c' = (if c = c' then c else And c c')"

fun "not" :: "cexp ⇒ cexp" where
  "not (Bc x) = (Bc (¬ x))" |
  "not (Not x) = x" |
  "not x = Not x"

abbreviation Leq :: "value ⇒ cexp" where
  "Leq v ≡ Not (Gt v)"

abbreviation Geq :: "value ⇒ cexp" where
  "Geq v ≡ Not (Lt v)"

abbreviation Neq :: "value ⇒ cexp" where
  "Neq v ≡ Not (Eq v)"

abbreviation Or :: "cexp ⇒ cexp ⇒ cexp" where
  "Or v va ≡ not (and (not v) (not va))"
```

```
fun cexp2gexp :: "aexp ⇒ cexp ⇒  gexp" where
  "cexp2gexp _ (Bc b) = gexp.Bc b" |
  "cexp2gexp a Undef = Null a" |
  "cexp2gexp a (Lt v) = gexp.Gt (L v) a" |
  "cexp2gexp a (Gt v) = gexp.Gt a (L v)" |
  "cexp2gexp a (Eq v) = gexp.Eq a (L v)" |
  "cexp2gexp a (Not v) = gNot (cexp2gexp a v)" |
  "cexp2gexp a (And v va) = gAnd (cexp2gexp a v) (cexp2gexp a va)"

definition cval :: "cexp ⇒ aexp ⇒ (datastate ⇒ trilean)" where
  "cval c a = gval (cexp2gexp a c)"

lemma cval_true: "cval (Bc True) a i = true"
  by (simp add: cval_def gval.simps)

lemma cval_false: "cval (cexp.Bc False) a i = false"
  by (simp add: cval_def gval.simps)

lemma cval_And_zero: "cval (And c (cexp.Bc True)) = cval c"
  apply (rule ext)+
  using cval_def gAnd_symmetry gAnd_zero gexp_equiv_def by force

lemma cval_And: "cval (And x y) a s = maybe_and (cval x a s) (cval y a s)"
  apply (simp only: cval_def)
  using gval_gAnd by auto

lemma cval_And_one: "cval (And c c) = cval c"
  apply (rule ext)+
  using cval_def cval_And maybe_and_idempotent by auto

lemma cval_And_fun: "cval (And x y) = (λr s. maybe_and (cval x r s) (cval y r s))"
  apply (rule ext)+
  by (simp only: cval_And)

lemma and_is_And :  "cval (and x y) = cval (And x y)"
proof (induct x y rule: and.induct)
  case (1 x)
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd gval.simps)
    by (simp add: maybe_double_negation maybe_or_idempotent maybe_or_zero)
next
  case "2_1"
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd gval.simps(1))
    by (simp add: maybe_and_commutative maybe_and_one)
next
  case "2_2"
  then show ?case
    apply (rule ext)+
    by (simp add: cval_def gval_gAnd gval.simps(2) gval.simps(1))
next
  case ("2_3" v)
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd gval.simps(1))
    by (simp add: maybe_and_commutative maybe_and_one)
next
  case ("2_4" v)
  then show ?case
```

```
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd gval.simps(1))
    by (simp add: maybe_and_commutative maybe_and_one)
next
  case ("2_5" v)
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd gval.simps(1))
    by (simp add: maybe_and_commutative maybe_and_one)
next
  case ("2_6" v)
  then show ?case
    apply (rule ext)+
    by (simp add: cval_And_zero)
next
  case ("2_7" v va)
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd gval.simps(1))
    by (simp add: maybe_and_commutative maybe_and_one)
next
  case "3_1"
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd)
    by (simp add: maybe_and_idempotent)
next
  case "3_2"
  then show ?case by (simp add: cval_def)
next
  case ("3_3" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_4" v)
then show ?case by (simp add: cval_def)
next
  case ("3_5" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_6" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_7" v va)
  then show ?case by (simp add: cval_def)
next
  case "3_8"
  then show ?case by (simp add: cval_def)
next
  case "3_9"
  then show ?case
    apply (rule ext)+
    by (simp add: cval_def gval_gAnd gval.simps(2))
next
  case ("3_10" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_11" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_12" v)
  then show ?case by (simp add: cval_def)
```

```
next
  case ("3_13" v)
then show ?case by (simp add: cval_def)
next
  case ("3_14" v va)
then show ?case by (simp add: cval_def)
next
  case ("3_15" v)
then show ?case by (simp add: cval_def)
next
  case ("3_16" v)
then show ?case by (simp add: cval_def)
next
  case ("3_17" v va)
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd)
    by (simp add: maybe_and_idempotent)
next
  case ("3_18" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_19" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_20" v va)
  then show ?case by (simp add: cval_def)
next
case ("3_21" v va vb)
  then show ?case by (simp add: cval_def)
next
case ("3_22" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_23" v)
then show ?case by (simp add: cval_def)
next
  case ("3_24" v va)
  then show ?case by (simp add: cval_def)
next
case ("3_25" v va)
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def gval_gAnd)
    by (simp add: maybe_and_idempotent)
next
  case ("3_26" v va)
then show ?case by (simp add: cval_def)
next
  case ("3_27" v va)
then show ?case by (simp add: cval_def)
next
  case ("3_28" v va vb)
  then show ?case by (simp add: cval_def)
next
  case ("3_29" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_30" v)
  then show ?case by (simp add: cval_def)
next
```

```
case ("3_31" v va)
  then show ?case by (simp add: cval_def)
next
case ("3_32" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_33" v va)
  then show ?case
    apply (rule ext)+
    apply (simp add: cval_def)
    by (simp add: maybe_or_idempotent or_equiv)
next
  case ("3_34" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_35" v va vb)
  then show ?case by (simp add: cval_def)
next
  case ("3_36" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_37" v)
  then show ?case by (simp add: cval_def)
next
  case ("3_38" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_39" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_40" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_41" v va)
  then show ?case
    by (simp add: cval_And_one)
next
  case ("3_42" v va vb)
  then show ?case by (simp add: cval_def)
next
  case ("3_43" v va)
  then show ?case by (simp add: cval_def)
next
  case ("3_44" v va)
  then show ?case by (simp add: cval_def)
next
case ("3_45" v va vb)
  then show ?case by (simp add: cval_def)
next
  case ("3_46" v va vb)
  then show ?case by (simp add: cval_def)
next
case ("3_47" v va vb)
  then show ?case by (simp add: cval_def)
next
  case ("3_48" v va vb)
  then show ?case by (simp add: cval_def)
next
case ("3_49" v va vb vc)
  then show ?case
    by (simp add: cval_And_one)
```

qed

definition *valid :: "cexp ⇒ bool"* **where**
"valid c ≡ (∀ a s. cval c a s = true)"

definition *satisfiable :: "cexp ⇒ bool"* **where**
"satisfiable c ≡ (∃ a s. cval c a s = true)"

fun *compose_plus :: "cexp ⇒ cexp ⇒ cexp"* **where**
"compose_plus Undef b = Undef" |
"compose_plus b Undef = Undef" |
"compose_plus (Bc False) _ = Bc False" |
"compose_plus _ (Bc False) = Bc False" |
"compose_plus (Bc True) _ = Bc True" |
"compose_plus _ (Bc True) = Bc True" |
"compose_plus (Eq (Num x)) (Eq (Num y)) = Eq (Num (x+y))" |
"compose_plus (Eq (Str x)) _ = Undef" |
"compose_plus _ (Eq (Str x)) = Undef" |
"compose_plus (Eq (Num x)) (Lt (Num y)) = Lt (Num (x+y))" |
"compose_plus (Lt (Num y)) (Eq (Num x)) = Lt (Num (x+y))" |
"compose_plus (Lt (Num va)) (Lt (Num vb)) = Lt (Num (va + vb))" |
"compose_plus (Lt (Num vb)) (Gt (Num v)) = Bc True" |
"compose_plus (Gt (Num v)) (Lt (Num vb)) = Bc True" |
"compose_plus _ (Lt (Str y)) = Undef" |
"compose_plus (Lt (Str y)) _ = Undef" |
"compose_plus (Eq (Num x)) (Gt (Num y)) = Gt (Num (x+y))" |
"compose_plus (Gt (Num y)) (Eq (Num x)) = Gt (Num (x+y))" |
"compose_plus (Gt (Num va)) (Gt (Num vb)) = Gt (Num (va + vb))" |
"compose_plus _ (Gt (Str y)) = Undef" |
"compose_plus (Gt (Str y)) _ = Undef" |
"compose_plus a (Not va) = (if (compose_plus a va) = Undef then Undef else (compose_plus a va))" |
"compose_plus (Not va) a = (if (compose_plus va a) = Undef then Undef else (compose_plus va a))" |
"compose_plus a (And v va) = and (compose_plus a v) (compose_plus a va)" |
"compose_plus (And v va) a = and (compose_plus a v) (compose_plus a va)"

fun *compose_minus :: "cexp ⇒ cexp ⇒ cexp"* **where**
"compose_minus Undef b = Undef" |
"compose_minus b Undef = Undef" |
"compose_minus (Bc False) _ = Bc False" |
"compose_minus _ (Bc False) = Bc False" |
"compose_minus (Bc True) _ = Bc True" |
"compose_minus _ (Bc True) = Bc True" |
"compose_minus (Eq (Num x)) (Eq (Num y)) = Eq (Num (x-y))" |
"compose_minus (Eq (Str x)) _ = Undef" |
"compose_minus _ (Eq (Str x)) = Undef" |
"compose_minus (Eq (Num x)) (Lt (Num y)) = Gt (Num (x - y))" |
"compose_minus (Lt (Num y)) (Eq (Num x)) = Lt (Num (y - x))" |
"compose_minus (Lt (Num a)) (Lt (Num b)) = Bc True" |
"compose_minus (Lt (Num vb)) (Gt (Num v)) = Lt (Num (vb - v))" |
"compose_minus (Gt (Num v)) (Lt (Num vb)) = Gt (Num (v - vb))" |
"compose_minus _ (Lt (Str y)) = Undef" |
"compose_minus (Lt (Str y)) _ = Undef" |
"compose_minus (Eq (Num d)) (Gt (Num b)) = Lt (Num (d - b))" |
"compose_minus (Gt (Num b)) (Eq (Num d)) = Gt (Num (b - d))" |
"compose_minus (Gt (Num va)) (Gt (Num vb)) = Bc True" |
"compose_minus _ (Gt (Str y)) = Undef" |
"compose_minus (Gt (Str y)) _ = Undef" |
"compose_minus a (Not va) = (if (compose_minus a va) = Undef then Undef else (compose_minus a va))"
|
"compose_minus (Not va) a = (if (compose_minus va a) = Undef then Undef else (compose_minus va a))"
|

```
  "compose_minus a (And v va) = and (compose_minus a v) (compose_minus a va)" |
  "compose_minus (And v va) a = and (compose_minus a v) (compose_minus a va)"

lemma valid_implies_satisfiable: "valid c ⟹ satisfiable c"
  by (simp add: valid_def satisfiable_def)

definition cexp_equiv :: "cexp ⇒ cexp ⇒ bool" where
  "cexp_equiv c c' ≡ (∀a s. cval c a s = cval c' a s)"

lemma cexp_equiv_reflexive: "cexp_equiv x x"
  by (simp add: cexp_equiv_def gexp_equiv_reflexive)

lemma gNegate: "gexp_equiv (gNot g) (gexp.Bc True) = gexp_equiv g (gexp.Bc False)"
  by (simp add: gexp_equiv_def gval.simps(1) gval.simps(2) maybe_negate_true not_equiv)

lemma cexp_equiv_valid: "valid c ⟶ cexp_equiv c (Bc True)"
  by (simp add: valid_def cexp_equiv_def cval_def gval.simps)

lemma cval_and: "cval (and x y) a s = maybe_and (cval x a s) (cval y a s)"
  by (simp only: and_is_And cval_And)

lemma cexp_equiv_redundant_and: "cexp_equiv (and c (and c c')) (and c c')"
  apply (simp add: cexp_equiv_def cval_and)
  by (metis maybe_and_associative maybe_and_idempotent)

lemma cval_And_commutative: "cval (And x y) a s = cval (And y x) a s"
  by (simp only: cval_And maybe_and_commutative)

lemma and_symmetric: "cexp_equiv (and x y) (and y x)"
  apply (simp only: cexp_equiv_def and_is_And)
  by (simp add: cval_And_commutative)

lemma gval_and: "gval (cexp2gexp a (and c1 c2)) = gval (gAnd (cexp2gexp a c1) (cexp2gexp a c2))"
  apply (rule ext)
  apply (simp only: gval_gAnd)
  by (metis cval_and cval_def)

lemma cexp_equiv_symmetric: "cexp_equiv x y = cexp_equiv y x"
  apply (simp only: cexp_equiv_def cval_def)
  by auto

lemma cexp_equiv_transitive: "cexp_equiv x y ⟹ cexp_equiv y z ⟹ cexp_equiv x z"
  by (simp add: cexp_equiv_def gexp_equiv_def)

lemma cval_Not: "cval (Not x) a s = maybe_not (cval x a s)"
  by (simp add: cval_def gval.simps maybe_or_idempotent)

lemma cval_not: "cval (not x) a s = maybe_not (cval x a s)"
proof(induct x)
  case Undef
  then show ?case by (simp add: cval_Not)
next
case (Bc x)
  then show ?case
    apply (case_tac x)
     apply (simp add: cval_false cval_true)
    by (simp add: cval_false cval_true)
next
case (Eq x)
  then show ?case by (simp add: cval_Not)
next
```

```
  case (Lt x)
  then show ?case by (simp add: cval_Not)
next
  case (Gt x)
  then show ?case by (simp add: cval_Not)
next
  case (Not x)
  then show ?case
    by (simp add: cval_Not maybe_double_negation)
next
  case (And x1 x2)
  then show ?case by (simp add: cval_Not)
qed

lemma cval_double_negation: "cval (Not (Not x)) = cval x"
  apply (rule ext)+
  by (simp only: cval_Not maybe_double_negation)

lemma valid_double_negation: "valid (Not (Not x)) = valid x"
  by (simp add: valid_def cval_double_negation)

lemma not_is_Not: "cval (not x) = cval (Not x)"
  apply (rule ext)+
  by (simp add: cval_not cval_Not)

lemma true_not_false: "cval (Bc True) = cval (Not (Bc False))"
  apply (rule ext)+
  by (simp add: cval_Not cval_false cval_true)

lemma false_not_true: "cval (Bc False) = cval (Not (Bc True))"
  apply (rule ext)+
  by (simp add: cval_Not cval_false cval_true)

lemma satisfiable_undef: "satisfiable Undef"
  apply (simp add: satisfiable_def)
  apply (rule_tac x="V (R 1)" in exI)
  apply (rule_tac x="<>" in exI)
  by (simp add: cval_def gval.simps ValueEq_def)

lemma invalid_undef: "¬ valid Undef"
  apply (simp add: valid_def cval_def)
  apply (rule_tac x="V (R 1)" in exI)
  apply (rule_tac x="<R 1 := Num 5>" in exI)
  by (simp add: cval_def gval.simps ValueEq_def)

lemma satisfiable_true: "satisfiable (Bc True)"
  by (simp add: satisfiable_def cval_def gval.simps)

lemma valid_true: "valid (Bc True)"
  by (simp add: valid_def cval_def gval.simps)

lemma unsatisfiable_false: "¬ satisfiable (Bc False)"
  by (simp add: satisfiable_def cval_def gval.simps)

lemma invalid_false: "¬valid (cexp.Bc False)"
  by (simp add: valid_def cval_def gval.simps)

lemma satisfiable_eq: "satisfiable (Eq x)"
  apply (simp add: satisfiable_def cval_def gval.simps ValueEq_def)
  using aval.simps(1) by blast
```

```
lemma invalid_eq: "¬ valid (cexp.Eq x)"
  apply (simp add: valid_def cval_def)
  apply (rule_tac x="V (R 1)" in exI)
  apply (rule_tac x="<>" in exI)
  by (simp add: cval_def gval.simps ValueEq_def)

lemma satisfiable_lt: "satisfiable (Lt (Num x))"
  apply (simp add: satisfiable_def cval_def gval.simps ValueGt_def)
  by (metis (full_types) MaybeBoolInt.simps(1) aval.simps(1) lt_ex)

lemma unsatisfiable_lt: "¬ satisfiable (Lt (Str s))"
  by (simp add: satisfiable_def cval_def gval.simps ValueGt_def)

lemma invalid_lt: "¬ valid (Lt x)"
  apply (simp add: valid_def cval_def)
  apply (rule_tac x="V (R 1)" in exI)
  apply (rule_tac x="<>" in exI)
  by (simp add: cval_def gval.simps ValueGt_def)

lemma satisfiable_gt: "satisfiable (Gt (Num x4))"
  apply (simp add: satisfiable_def cval_def gval.simps ValueGt_def)
  by (metis (full_types) MaybeBoolInt.simps(1) aval.simps(1) zless_iff_Suc_zadd)

lemma unsatisfiable_gt: "¬ satisfiable (Gt (Str s))"
  by (simp add: satisfiable_def cval_def gval.simps ValueGt_def)

lemma invalid_gt: "¬ valid (cexp.Gt x5)"
  apply (simp add: valid_def cval_def)
  apply (rule_tac x="V (R 2)" in exI)
  apply (rule_tac x="<>" in exI)
  by (simp add: gval.simps ValueGt_def)

lemma satisfiable_not_undef: "satisfiable (Not (Undef))"
  apply (simp add: satisfiable_def cval_def gval.simps ValueEq_def)
  using aval.simps(1) by blast

lemma satisfiable_neq: "satisfiable (Neq x3)"
  apply (simp add: satisfiable_def cval_def gval.simps ValueEq_def)
  by (metis aval.simps(1) option.inject value.simps(4))

lemma satisfiable_leq: "satisfiable (Leq (Num x))"
  apply (simp add: satisfiable_def cval_Not maybe_negate_true)
  apply (simp add: cval_def gval.simps ValueGt_def)
  by (metis MaybeBoolInt.simps(1) aval.simps(1) minf(4))

lemma satisfiable_geq: "satisfiable (Geq (Num x))"
  apply (simp add: satisfiable_def cval_Not maybe_negate_true)
  apply (simp add: cval_def gval.simps ValueGt_def)
  by (metis MaybeBoolInt.simps(1) aval.simps(1) pinf(4))

lemma "satisfiable (Not x) ⟹ ¬valid x"
proof(induct x)
case Undef
  then show ?case
    by (simp add: invalid_undef satisfiable_not_undef)
next
  case (Bc x)
  then show ?case
    by (metis (full_types) CExp.satisfiable_def false_not_true invalid_false  unsatisfiable_false)
next
  case (Eq x)
```

```
  then show ?case
    by (simp add: invalid_eq)
next
  case (Lt x)
  then show ?case
    by (simp add: invalid_lt)
next
  case (Gt x)
  then show ?case
    by (simp add: invalid_gt)
next
  case (Not x)
  then show ?case
    by (metis CExp.satisfiable_def cval_Not cval_double_negation valid_def)
next
  case (And x1 x2)
  then show ?case
    using CExp.satisfiable_def cval_Not valid_def by force
qed

lemma and_x_y_undef: "and x y = Undef ⟹ and y x = Undef"
  apply (induct x y rule: and.induct)
                    apply simp_all
      apply (case_tac "v = va")
       apply simp+
      apply (case_tac "v = va")
      apply simp+
      apply (case_tac "v = va")
     apply simp+
      apply (case_tac "v = va")
    apply simp+
  apply (case_tac "v = vb ∧ va = vc")
  by auto

definition mutually_exclusive :: "cexp ⇒ cexp ⇒ bool" where
  "mutually_exclusive x y = (∀a i. (cval x i a= true ⟶ cval y i a ≠ true) ∧
                                (cval y i a = true ⟶ cval x i a ≠ true))"

lemma mutually_exclusive_unsatisfiable_conj: "mutually_exclusive x y = (¬ satisfiable (And x y))"
  apply (simp add: mutually_exclusive_def satisfiable_def)
  apply (simp add: cval_And)
  by (metis (no_types, lifting) maybe_and_associative maybe_and_commutative maybe_and_idempotent maybe_and_one)

lemma unsatisfiable_conj_mutually_exclusive: "¬ satisfiable (And x y) = mutually_exclusive x y"
  by (simp add: mutually_exclusive_unsatisfiable_conj)

lemma mutually_exclusive_reflexive: "satisfiable x ⟹ ¬ mutually_exclusive x x"
  apply (simp add: mutually_exclusive_def satisfiable_def)
  by auto

lemma mutually_exclusive_symmetric: "mutually_exclusive x y ⟹ mutually_exclusive y x"
  by (simp add: mutually_exclusive_def)

lemma not_mutually_exclusive_true: "satisfiable x = (¬ mutually_exclusive x (Bc True))"
  apply (simp add: mutually_exclusive_def satisfiable_def)
  using valid_def valid_true by blast

lemma cval_values: "(cval x i a ≠ false) = (cval x i a = true ∨ cval x i a = invalid)"
  by (metis maybe_not.cases trilean.distinct(1) trilean.distinct(5))

lemma x_neq_not_x: "x ≠ cexp.Not x"
```

```
  apply (induct_tac x)
  by auto

lemma gval_And: "gval (cexp2gexp a (And c1 c2)) = gval (gAnd (cexp2gexp a c1) (cexp2gexp a c2))"
  apply (rule ext)
  by simp

lemma gval_not: "gval (cexp2gexp a (Not c)) = gval (gNot (cexp2gexp a c))"
  apply (rule ext)
  by simp

lemma gval_True: "gval (cexp2gexp a (cexp.Bc True)) x = true"
  by (simp add: gval.simps)

lemma gval_and_cexp: "gval (cexp2gexp i c1) s ≠ true ⟹  gval (cexp2gexp i (and c2 c1)) s ≠ true"
  apply (simp add: gval_and gval_gAnd)
  using maybe_and.elims by blast

lemma gval_and_false: "gval (cexp2gexp r (and (cexp.Bc False) c)) s ≠ true"
  apply (simp add: gval_and gval_gAnd gval.simps(2))
  using maybe_and.elims by blast

lemma gval_and_false_2: "gval (cexp2gexp uu (and x (cexp.Bc False))) s ≠ true"
  by (metis and.simps(17) gval_and_cexp gval_and_false)

lemma and_true: "and c (cexp.Bc True) = c"
  apply (case_tac c)
        apply simp
       apply (case_tac x2)
  by auto

lemma and_self: "and x x = x"
  apply (case_tac x)
        apply simp
       apply (case_tac x2)
  by auto

lemma and_false_not_undef: "and (Bc False) c ≠ Undef"
  apply (induct_tac c)
        apply simp
       apply (case_tac x)
  by auto

lemma and_And_false: "x ≠ cexp.Bc True ∧ x ≠ Bc False ⟹ and (Bc False) x = And (Bc False) x"
  apply (case_tac x)
  by auto

lemma cval_And_false: "cval (And c (Bc False)) a s ≠ true"
  using CExp.satisfiable_def cval_And maybe_and_not_true unsatisfiable_false by auto
```

7.2 A Linear Ordering for Constraint Expressions

Contexts represent constraints as a finite set of constraint expressions, the ffold operation on fsets is a pain to use as nothing proves. It's much easier to convert to a list and use the list fold method. In order to convert from an fset to a list, we need a linear order. We define that ordering here.

```
instantiation cexp :: linorder begin
fun less_cexp :: "cexp ⇒ cexp ⇒ bool" where
  "(Undef < Undef) = False" |
  "(Undef < _) = True" |
```

```
  "(Bc v < Undef) = False" |
  "(Bc v < Bc va) = less v va" |
  "(Bc v < _) = True" |

  "(Eq v < Undef) = False" |
  "(Eq v < Bc va) = False" |
  "(Eq v < Eq va) = less v va" |
  "(Eq v < _) = True" |

  "(Lt v < Undef) = False" |
  "(Lt v < Bc va) = False" |
  "(Lt v < Eq va) = False" |
  "(Lt v < Lt va) = less v va" |
  "(Lt v < _) = True" |

  "(Gt v < Gt va) = less v va" |
  "(Gt v < Not va) = True" |
  "(Gt v < And va vb) = True" |
  "(Gt v < _) = False" |

  "(Not v < Not va) = less v va" |
  "(Not v < And va vb) = True" |
  "(Not v < _) = False" |

  "(And g1 g2) < (And g1' g2') = ((g1 < g1') ∨ ((g1 = g1') ∧ (g2 < g2')))" |
  "(And vv v < _) = False"

definition less_eq_cexp :: "cexp ⇒ cexp ⇒ bool" where
  "less_eq_cexp a b = (a < b ∨ a = b)"
declare less_eq_cexp_def [simp]

lemma undef_minimal: "Undef ≠ z ⟹ Undef < z"
  apply (cases z)
  by auto

lemma hard_less: "((x::cexp) < y) = (x ≤ y ∧ ¬ y ≤ x)"
  apply (induct x y rule: less_cexp.induct)
  by auto

lemma x_leq_x: "(x::cexp) ≤ x"
  apply (induct x)
  by auto

lemma x_leq_y_or_y_lex_x: "(x::cexp) ≤ y ∨ y ≤ x"
  apply (induct x y rule: less_cexp.induct)
  by auto

lemma antisymmetry: "(x::cexp) ≤ y ⟹ y ≤ x ⟹ x = y"
  apply (induct x y rule: less_cexp.induct)
                    apply simp_all
      apply auto[1]
     apply auto[1]
    apply auto[1]
   apply auto[1]
  by (meson hard_less)

lemma transitivity: "(x::cexp) ≤ y ⟹ y ≤ z ⟹ x ≤ z"
proof (induct x y arbitrary: z rule: less_cexp.induct)
case 1
  then show ?case
    by simp
```

```
next
  case ("2_1" v)
then show ?case
  using less_eq_cexp_def undef_minimal by blast
next
case ("2_2" v)
then show ?case
  using less_eq_cexp_def undef_minimal by blast
next
case ("2_3" v)
  then show ?case
  using less_eq_cexp_def undef_minimal by blast
next
  case ("2_4" v)
  then show ?case
    using less_eq_cexp_def undef_minimal by blast
next
case ("2_5" v)
  then show ?case
    using less_eq_cexp_def undef_minimal by blast
next
  case ("2_6" v va)
then show ?case
    using less_eq_cexp_def undef_minimal by blast
next
  case (3 v)
  then show ?case
    by simp
next
  case (4 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("5_1" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("5_2" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("5_3" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("5_4" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("5_5" v va vb)
  then show ?case
    apply (cases z)
    by auto
next
  case (6 v)
  then show ?case
```

```
    apply (cases z)
    by auto
next
  case (7 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (8 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("9_1" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("9_2" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("9_3" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("9_4" v va vb)
  then show ?case
    apply (cases z)
    by auto
next
  case (10 v)
  then show ?case
    apply (cases z)
    by auto
next
  case (11 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (12 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (13 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("14_1" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("14_2" v va)
  then show ?case
    apply (cases z)
```

```
    by auto
next
  case ("14_3" v va vb)
  then show ?case
    apply (cases z)
    by auto
next
  case (15 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (16 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (17 v va vb)
  then show ?case
    apply (cases z)
    by auto
next
  case ("18_1" v)
  then show ?case
    apply (cases z)
    by auto
next
  case ("18_2" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("18_3" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("18_4" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (19 v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (20 v va vb)
  then show ?case
    apply (cases z)
    by auto
next
  case ("21_1" v)
  then show ?case
    apply (cases z)
    by auto
next
  case ("21_2" v va)
  then show ?case
    apply (cases z)
    by auto
```

```
next
  case ("21_3" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("21_4" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("21_5" v va)
  then show ?case
    apply (cases z)
    by auto
next
  case (22 g1 g2 g1' g2')
  then show ?case
    apply simp
    apply (case_tac "And g1' g2' = z")
     apply auto[1]
    apply simp
    apply (case_tac "g1 < g1'")
     apply simp
     apply (metis cexp.exhaust hard_less less_cexp.simps(14) less_cexp.simps(28) less_cexp.simps(31)
less_cexp.simps(43) less_cexp.simps(46) less_cexp.simps(49) less_cexp.simps(7) less_eq_cexp_def)
    apply simp
    apply (case_tac "g1 = g1' ∧ g2 < g2'")
     apply simp
     apply (metis cexp.exhaust hard_less less_cexp.simps(14) less_cexp.simps(28) less_cexp.simps(31)
less_cexp.simps(43) less_cexp.simps(46) less_cexp.simps(49) less_cexp.simps(7) less_eq_cexp_def)
    by simp
next
  case ("23_1" vv v)
  then show ?case
    apply (cases z)
    by auto
next
  case ("23_2" vv v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("23_3" vv v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("23_4" vv v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("23_5" vv v va)
  then show ?case
    apply (cases z)
    by auto
next
  case ("23_6" vv v va)
  then show ?case
    apply (cases z)
```

```
    by auto
qed

instance
  apply standard
  using hard_less apply blast
  apply simp
  using transitivity apply blast
  using antisymmetry apply blast
  using x_leq_y_or_y_lex_x by auto
end

end
```

7.3 Contexts

This theory defines contexts as a way of relating possible constraints on register values to observable output. We then use contexts to extend the idea of transition subsumption to EFSM transitions with register update functions.

```
theory Contexts
  imports
    EFSM GExp CExp
begin

type_synonym "context" = "aexp ⇒ cexp fset"


abbreviation empty :: "context" ("[]") where
  "empty ≡ (λx. case x of
    (V v) ⇒ (case v of R n ⇒ {|Undef|} | I n ⇒ {|Bc True|}) |
    _ ⇒ {|Bc True|}
  )"
syntax
  "_updbind" :: "'a ⇒ 'a ⇒ updbind" ("(2_ ↦/ _)")
  "_Context" :: "updbinds ⇒ 'a"      ("[_]")
translations
  "_Update f (_updbinds b bs)" ⇌ "_Update (_Update f b) bs"
  "_Context ms" == "_Update [] ms"
  "_Context (_updbinds b bs)" ⇌ "_Update (_Context b) bs"

lemma empty_register: "[] (V (R r)) = {|Undef|}"
  by (simp)

lemma empty_input: "[] (V (I i)) = {|Bc True|}"
  by (simp)

lemma consistent_empty_fball: "fBall ([] r) (λc. cval c r Map.empty = true)"
  apply (cases r)
     apply (simp add: cval_true)
    apply (case_tac x2)
     apply (simp add: cval_true)
    apply (simp add: cval_def gval.simps ValueEq_def)
  using cval_true by auto

lemma empty_not_false[simp]: "{|Bc False|} ≠ [] i"
proof (induct i)
case (L x)
then show ?case by simp
next
  case (V x)
```

```
  then show ?case
    apply (case_tac x)
    by simp_all
next
  case (Plus i1 i2)
  then show ?case
    by simp
next
  case (Minus i1 i2)
  then show ?case
    by simp
qed

lemma empty_variable_constraints: "⟦⟧ (V (R ri)) = {|Undef|} ∧ ⟦⟧ (V (I i)) = {|Bc True|}"
  by simp

fun get :: "context ⇒ aexp ⇒ cexp fset" where
  "get c (L n) = {|Eq n|}" |
  "get c (V v) = c (V v)" |
  "get c (Plus v va) = (c (Plus v va)) |∪| (c (Plus va v))" |
  "get c (Minus v va) = (c (Minus v va))"

fun update :: "context ⇒ aexp ⇒ cexp fset ⇒ context" where
  "update c (L n) _ = c" |
  "update c k v = (λr. if r=k then v else c r)"

definition conjoin :: "cexp fset ⇒ cexp" where
  "conjoin f = foldr And (sorted_list_of_fset f) (Bc True)"

definition consistent :: "context ⇒ bool" where
  "consistent c ≡ ∃s. ∀r. fBall (c r) (λc. (cval c r s = true))"

lemma subset_consistency: "∀r. c' r |⊆| c r ⟹ consistent c ⟹ consistent c'"
  apply (simp add: consistent_def)
  apply clarify
  apply (rule_tac x=s in exI)
  by auto

lemma possible_false_not_consistent: "∃r. c r = {|Bc False|} ⟹ ¬ consistent c"
  apply (simp add: consistent_def conjoin_def)
  apply clarify
  apply (rule_tac x=r in exI)
  by (simp add: sorted_list_of_fset_def cval_And cval_false cval_true)

lemma inconsistent_false: "¬consistent (λi. {|Bc False|})"
  using possible_false_not_consistent
  by simp

lemma consistent_empty_1: "empty r = {|Undef|} ∨ empty r = {|Bc True|}"
  apply (cases r)
  prefer 2
    apply (case_tac x2)
  by simp_all

theorem consistent_empty_2: "(∀r. c r = {|Bc True|}) ⟶ consistent c"
  apply (simp add: consistent_def conjoin_def sorted_list_of_fset_def)
  by (simp add: cval_And cval_true)

lemma consistent_empty_4: "⟦⟧ r = {|Undef|} ∨ gval (cexp2gexp r (conjoin (⟦⟧ r))) c = true"
  apply (case_tac r)
     apply (simp add: consistent_def conjoin_def sorted_list_of_fset_def maybe_double_negation)
```

```
     apply (simp add: gval_gAnd gval.simps(1))
    apply (case_tac x2)
     apply (simp add: conjoin_def sorted_list_of_fset_def gval_gAnd gval.simps(1))
    apply (simp add: conjoin_def)
   apply (simp add: conjoin_def sorted_list_of_fset_def gval_gAnd gval.simps(1))
  by (simp add: conjoin_def sorted_list_of_fset_def gval_gAnd gval.simps(1))

lemma consistent_empty [simp]: "consistent empty"
  apply (simp add: consistent_def cval_def)
  apply (rule_tac x="<>" in exI)
  apply clarify
  apply (case_tac r)
     apply (simp add: conjoin_def sorted_list_of_fset_def gval_gAnd gval.simps(1))
    apply (case_tac x2)
  by (simp_all add: conjoin_def sorted_list_of_fset_def gval_gAnd gval.simps ValueEq_def)

lemma cexp2gexp_double_neg: "gexp_equiv (cexp2gexp r (Not (Not x))) (cexp2gexp r x)"
  apply (simp add: gexp_equiv_def gval_gAnd)
  by (simp add: maybe_and_idempotent)

lemma gval_cexp2gexp_double_neg: "gval (cexp2gexp r (Not (Not x))) s = gval (cexp2gexp r x) s"
  using cexp2gexp_double_neg gexp_equiv_def by blast

fun make_gt :: "cexp ⇒ cexp" where
  "make_gt (Bc b) = Bc b" |
  "make_gt Undef = Undef" |
  "make_gt (Eq v) = Gt v" |
  "make_gt (Lt v) = Bc True" |
  "make_gt (Gt s) = Gt s" |
  "make_gt (Not v) = Not (make_gt v)" |
  "make_gt (And v va) = And (make_gt v) (make_gt va)"

lemma make_gt_twice: "make_gt (make_gt x) = make_gt x"
  apply (induct x)
  by auto

lemma cval_make_gt_not: "cval (make_gt (not x)) r s = maybe_not (cval (make_gt x) r s)"
proof(induct x)
case Undef
  then show ?case
    by (simp add: cval_Not)
next
  case (Bc x)
  then show ?case
    apply simp
    by (metis cval_not not.simps(1))
next
  case (Eq x)
  then show ?case
    by (simp add: cval_Not)
next
  case (Lt x)
  then show ?case
    by (simp add: cval_Not cval_false cval_true)
next
  case (Gt x)
  then show ?case
    by (simp add: cval_Not)
next
  case (Not x)
  then show ?case
```

```
    by (simp add: cval_Not maybe_double_negation)
next
  case (And x1 x2)
  then show ?case
    apply simp
    by (simp only: cval_And cval_Not cval_and)
qed

fun make_lt :: "cexp ⇒ cexp" where
  "make_lt (Bc b) = Bc b" |
  "make_lt Undef = Undef" |
  "make_lt (Eq v) = Lt v" |
  "make_lt (Lt v) = Lt v" |
  "make_lt (Gt v) = Bc True" |
  "make_lt (Not v) = Not (make_lt v)" |
  "make_lt (And v va) = And (make_lt v) (make_lt va)"

lemma make_lt_twice: "make_lt (make_lt x) = make_lt x"
  apply(induct x)
  by auto

fun guard2pairs :: "context ⇒ guard ⇒ (aexp × cexp fset) list" where
  "guard2pairs a (gexp.Bc True) = []" |
  "guard2pairs a (gexp.Bc False) = [(L (Num 0), {|Bc False|})]" |

  "guard2pairs a (gexp.Null v) = [(v, {|Undef|})]" |

  "guard2pairs a (gexp.Eq v (L n)) = [(v, {|Eq n|})]" |
  "guard2pairs a (gexp.Eq (L n) v) = [(v, {|Eq n|})]" |
  "guard2pairs a (gexp.Eq (Plus a1 a2) (Plus a4 a3)) = [((Plus a1 a2), (get a (Plus a2 a1)) |∪| (get
a (Plus a3 a4))),
                                                        ((Plus a2 a1), (get a (Plus a1 a2)) |∪| (get
a (Plus a3 a4))),
                                                        ((Plus a3 a4), (get a (Plus a4 a3)) |∪| (get
a (Plus a1 a2))),
                                                        ((Plus a4 a3), (get a (Plus a3 a4)) |∪| (get
a (Plus a1 a2)))]" |
  "guard2pairs a (gexp.Eq (Plus a1 a2) v) = [((Plus a1 a2), (get a v) |∪| (get a (Plus a1 a2))),
                                             ((Plus a2 a1), (get a v) |∪| (get a (Plus a2 a1))),
                                             (v, get a (Plus a1 a2))]" |
  "guard2pairs a (gexp.Eq v (Plus a1 a2)) = [((Plus a1 a2), (get a v) |∪| (get a (Plus a1 a2))),
                                             ((Plus a2 a1), (get a v) |∪| (get a (Plus a2 a1))),
                                             (v, get a (Plus a1 a2))]" |
  "guard2pairs a (gexp.Eq v va) = [(v, get a va), (va, get a v)]" |

  "guard2pairs a (gexp.Gt (L v) va) = (if L v = va then [(L (Num 0), {|Bc False|})] else [(va, {|Lt v|})])"
|
  "guard2pairs a (gexp.Gt va (L v)) = (if L v = va then [(L (Num 0), {|Bc False|})] else [(va, {|Gt v|})])"
|
  "guard2pairs a (gexp.Gt v vb) = (if v = vb then
                                     [(L (Num 0), {|Bc False|})]
                                   else
                                     [(v, fimage make_gt (get a vb))]
                                  )" |

  "guard2pairs a (Nor v va) = (map (λ(x, y). (x, fimage not y)) ((guard2pairs a v) @ (guard2pairs a
va)))"

definition pairs2context :: "(aexp × cexp fset) list ⇒ context" where
  "pairs2context l = (λr. fold funion (map snd (filter (λ(a, _). a = r) l)) {|||})"
```

```
lemma pairs2context_empty: "pairs2context [] x = {||}"
  by (simp add: pairs2context_def)

lemma pairs2context_append: "pairs2context (x @ y) ra = pairs2context x ra |∪| pairs2context y ra"
  apply (simp only: pairs2context_def)
  by (metis ffUnion_funion_distrib filter_append fold_union_ffUnion fset_of_list_append map_append map_eq_map_tai

lemma pairs2context_cons: "pairs2context (x # y) ra = pairs2context [x] ra |∪| pairs2context y ra"
  by (metis append_Cons append_self_conv2 pairs2context_append)

definition medial :: "context ⇒ guard list ⇒ context" where
 "medial c G = (λr. (c r) |∪| pairs2context (List.maps (guard2pairs c) G) r)"

lemma medial_cons: "medial c (a # G) ra = medial c [a] ra |∪| medial c G ra"
  apply (simp only: medial_def)
  by (simp add: inf_sup_aci(5) inf_sup_aci(7) maps_simps(1) maps_simps(2) pairs2context_append)

lemma List_maps_append:  "List.maps f (a@g) = (List.maps f a)@(List.maps f g)"
  by (simp add: List.maps_def)

lemma medial_append: "medial c (a @ G) ra = medial c a ra |∪| medial c G ra"
  apply (simp only: medial_def List_maps_append pairs2context_append)
  by auto

lemma medial_self_append: "medial c (g @ g) = medial c g"
  apply (rule ext)
  by (simp add: medial_append)

lemma medial_cons_subset: "medial c G ra |⊆| medial c (a # G) ra"
  apply (simp add: medial_def)
  apply (simp only: maps_simps(1))
  apply (simp only: pairs2context_append)
  by auto

lemma medial_filter: "medial c (filter f G) ra |⊆| medial c G ra"
proof(induct G)
  case Nil
  then show ?case by simp
next
  have aux1: "∀a f G ra c. medial c (a # filter f G) ra = medial c [a] ra |∪| medial c (filter f G)
ra"
    using medial_cons by blast
  case (Cons a G)
  then show ?case
    apply simp
    apply (case_tac "f a")
     apply simp
     defer
    apply simp
    using medial_cons_subset apply blast
    apply (simp only: aux1)
  proof -
    assume "medial c (filter f G) ra |⊆| medial c G ra"
    then have "medial c (a # G) ra = medial c [a] ra |∪| (medial c G ra |∪| medial c (filter f G) ra
|∪| medial c (filter f G) ra)"
      using medial_cons by blast
    then show "medial c [a] ra |∪| medial c (filter f G) ra |⊆| medial c (a # G) ra"
      by blast
  qed
qed
```

```
lemma medial_empty: "medial c [] = c"
  by (simp add: medial_def pairs2context_def List.maps_def)

lemma anterior_subset_medial: "c r |⊆| (medial c G r)"
    by (simp add: medial_def pairs2context_def)

fun apply_update :: "context ⇒ context ⇒ update_function ⇒ context" where
  "apply_update l c (v, (L n)) = update c (V v) {|(Eq n)|}" |
  "apply_update l c (v, V vb) = update c (V v) (l (V vb))" |
  "apply_update l c (v, Plus vb vc) = update c (V v) (fimage (λ(a, b). compose_plus a b) ((get l vb)
|×| (get l vc)))" |
  "apply_update l c (v, Minus vb vc) = update c (V v) (fimage (λ(a, b). compose_minus a b) ((get l vb)
|×| (get l vc)))"

primrec apply_updates :: "context ⇒ context ⇒ update_function list ⇒ context" where
  "apply_updates _ c [] = c" |
  "apply_updates l c (h#t) = (apply_update l (apply_updates l c t) h)"

definition can_take :: "transition ⇒ context ⇒ bool" where
  "can_take t c ≡ consistent (medial c (Guard t))"

lemma can_take_no_guards: "∀ c. (Contexts.consistent c ∧ (Guard t) = []) ⟶ Contexts.can_take t c"
  by (simp add: consistent_def Contexts.can_take_def medial_def pairs2context_def List.maps_def)

fun constrains_an_input :: "aexp ⇒ bool" where
  "constrains_an_input (L v) = False" |
  "constrains_an_input (V (R x)) = False" |
  "constrains_an_input (V (I x)) = True" |
  "constrains_an_input (Plus v va) = (constrains_an_input v ∨ constrains_an_input va)" |
  "constrains_an_input (Minus v va) = (constrains_an_input v ∨ constrains_an_input va)"

definition remove_obsolete_constraints :: "context ⇒ vname fset ⇒ context" where
  "remove_obsolete_constraints c vs = (λa. if ∃n. aexp_constrains a (V (I n)) ∨ fBex vs (λx. aexp_constrains
(V x) a ∧ a ≠ (V x)) then ⟦⟧ a else c a)"

lemma consistent_c_consistent_remove_obsolete_constraints: "consistent c ⟹ consistent (remove_obsolete_constr
c Any)"
  apply (simp add: remove_obsolete_constraints_def consistent_def)
  apply clarify
  apply (rule_tac x=s in exI)
  apply clarify
  apply (case_tac r)
     apply simp
  apply (case_tac x2)
  using cval_true by auto

lemma empty_inputs_are_true: "constrains_an_input x ⟹ ⟦⟧ x = {|Bc True|}"
  apply (case_tac x)
     apply simp
    apply (case_tac x2)
  by auto

lemma cval_empty_inputs: "constrains_an_input r ⟶ cval (conjoin (⟦⟧ r)) r ia = true"
proof(induct r)
case (L x)
  then show ?case by simp
next
  case (V x)
  then show ?case
    apply (cases x)
     apply (simp add: conjoin_def sorted_list_of_fset_def)
```

```
     apply (simp only: cval_And maybe_and_true cval_true)
    by simp
next
  case (Plus r1 r2)
  then show ?case
    apply (simp add: conjoin_def sorted_list_of_fset_def)
    apply (simp only: cval_And maybe_and_true cval_true)
    by simp
next
  case (Minus r1 r2)
  then show ?case
    apply (simp add: conjoin_def sorted_list_of_fset_def)
    apply (simp only: cval_And maybe_and_true cval_true)
    by simp
qed

lemma remove_input_constraints_empty[simp]: "remove_obsolete_constraints ⟦⟧ s = ⟦⟧"
  by (simp add: remove_obsolete_constraints_def)

definition posterior_separate :: "context ⇒ guard list ⇒ update_function list ⇒ context" where
  "posterior_separate c g u = (let c' = (medial c g) in (if consistent c' then remove_obsolete_constraints
(apply_updates c' c u) (fset_of_list (map fst u)) else (λi. {|Bc False|})))"

lemma posterior_separate_append_self: "posterior_separate c (g @ g) = posterior_separate c g"
  apply (rule ext)
  by (simp add: posterior_separate_def Let_def medial_self_append)

definition posterior :: "context ⇒ transition ⇒ context" where
  "posterior c t = posterior_separate c (Guard t) (Updates t)"

lemma posterior_consistent_medial: "medial c (Guard t) = c' ⟹ consistent c' ⟹ remove_obsolete_constraints
(apply_updates c' c (Updates t)) (fst |`| fset_of_list (Updates t)) = p ⟹ posterior c t = p"
  by (simp add: posterior_def posterior_separate_def)

primrec posterior_n :: "nat ⇒ transition ⇒ context ⇒ context" where
  "posterior_n 0 _ c = c " |
  "posterior_n (Suc m) t c = posterior_n m t (posterior c t)"

primrec posterior_sequence :: "context ⇒ transition_matrix ⇒ nat ⇒ datastate ⇒ trace ⇒ context"
where
  "posterior_sequence c _ _ _ [] = c" |
  "posterior_sequence c e s r (h#t) =
    (case (step e s r (fst h) (snd h)) of
      (Some (transition, s', outputs, r')) ⇒ (posterior_sequence (posterior c transition) e s' r' t)
|
      _ ⇒ c
    )"

definition datastate2context :: "datastate ⇒ context" where
  "datastate2context d = (λx. case x of V r ⇒ (case d r of None ⇒ {|Undef|} | Some v ⇒ {|Eq v|})
| _ ⇒ ⟦⟧ x)"

definition satisfies_context :: "datastate ⇒ context ⇒ bool" where
  "satisfies_context d c = consistent (λx. (datastate2context d x) |∪| c x)"

lemma satisfactory_registers: "c (V (R r)) = {|cexp.Eq v|} ⟹
      satisfies_context ra c ⟹
      ra (R r) = Some v"
proof-
  assume premise1: "c (V (R r)) = {|cexp.Eq v|}"
  assume premise2: "satisfies_context ra c"
```

```
  have contra: "c (V (R r)) = {|cexp.Eq v|} ⟹
                ra (R r) ≠ Some v ⟹
                ¬satisfies_context ra c"
    apply (simp add: satisfies_context_def datastate2context_def consistent_def)
    apply clarify
    apply (rule_tac x="V (R r)" in exI)
    apply (simp add: cval_def)
    apply (case_tac "ra (R r)")
    using gval.simps ValueEq_def
    by auto
  show ?thesis
    using premise1 premise2 contra by auto
qed

lemma cval_undef_empty: "cval Undef (V x) <> = true"
  by (simp add: cval_def gval.simps ValueEq_def)

lemma satisfies_context_empty: "satisfies_context <> ⟦⟧ ∧ satisfies_context Map.empty ⟦⟧"
  apply (simp add: satisfies_context_def datastate2context_def consistent_def)
  apply (rule_tac x="<>" in exI)
  apply clarify
  apply (case_tac r)
     apply (simp add: cval_true)
    apply (case_tac x2)
     apply (case_tac "x=Bc True")
  by (simp_all add: cval_true cval_def gval.simps ValueEq_def)


definition subsumes :: "transition ⇒ context ⇒ transition ⇒ bool" ("_ _⊒_" 60) where
  "subsumes t2 c t1 ≡ Label t1 = Label t2 ∧ Arity t1 = Arity t2 ∧ length (Outputs t1) = length (Outputs
t2) ∧
                      (∀r i. fBall (medial c (Guard t1) r) (λc. cval c r i = true) ⟶ fBall (medial
c (Guard t2) r) (λc. cval c r i = true)) ∧
                      (∀ i r. satisfies_context r c ⟶ apply_guards (Guard t1) (join_ir i r) ⟶ apply_outputs
(Outputs t1) (join_ir i r) = apply_outputs (Outputs t2) (join_ir i r)) ∧
                      (∃ i r. apply_outputs (Outputs t1) (join_ir i r) = apply_outputs (Outputs t2)
(join_ir i r)) ∧
                      (∀r i. fBall (posterior_separate c (Guard t1@Guard t2) (Updates t2) r) (λc. cval
c r i = true) ⟶ fBall (posterior c t1 r) (λc. cval c r i = true) ∨ (posterior c t1 r) = {|Undef|})
∧
                      (consistent (posterior c t1) ⟶ consistent (posterior c t2))"

lemma output_subsumption_violation: "¬ (∀ i r. satisfies_context r c ⟶ apply_guards (Guard t1) (join_ir
i r) ⟶ apply_outputs (Outputs t1) (join_ir i r) = apply_outputs (Outputs t2) (join_ir i r)) ⟹
     ¬ subsumes t2 c t1"
  by (simp add: subsumes_def)

lemma medial_subsumption_violation: "¬ (∀r i. fBall (medial c (Guard t1) r) (λc. cval c r i = true)
⟶ fBall (medial c (Guard t2) r) (λc. cval c r i = true)) ⟹
¬ subsumes t2 c t1"
  by (simp add: subsumes_def)

lemma update_subsumption_violation: "¬ (∀r i. fBall (posterior_separate c (Guard t1@Guard t2) (Updates
t2) r) (λc. cval c r i = true) ⟶ fBall (posterior c t1 r) (λc. cval c r i = true) ∨ (posterior c
t1 r) = {|Undef|}) ⟹
     ¬ subsumes t2 c t1"
  by (simp add: subsumes_def)

lemma outputs_never_equal: "¬(∃ i r. apply_outputs (Outputs t1) (join_ir i r) = apply_outputs (Outputs
t2) (join_ir i r)) ⟹
¬ subsumes t2 c t1"
```

```
  by (simp add: subsumes_def)

lemma subsumption: "Label t1 = Label t2 ∧
                    Arity t1 = Arity t2 ∧
                    length (Outputs t1) = length (Outputs t2) ⟹
                    (∀r i. fBall (medial c (Guard t1) r) (λc. cval c r i = true) ⟶ fBall (medial
c (Guard t2) r) (λc. cval c r i = true)) ⟹
                    (∀ i r. satisfies_context r c ⟶ apply_guards (Guard t1) (join_ir i r) ⟶ apply_outputs
(Outputs t1) (join_ir i r) = apply_outputs (Outputs t2) (join_ir i r)) ⟹
                    (∃ i r. apply_outputs (Outputs t1) (join_ir i r) = apply_outputs (Outputs t2) (join_ir
i r)) ⟹
                    (∀r i. fBall (posterior_separate c (Guard t1@Guard t2) (Updates t2) r) (λc. cval
c r i = true) ⟶ fBall (posterior c t1 r) (λc. cval c r i = true) ∨ (posterior c t1 r) = {|Undef|})
⟹
                    (consistent (posterior c t1) ⟶ consistent (posterior c t2)) ⟹
                    subsumes t2 c t1"
  by (simp add: subsumes_def)

definition anterior_context :: "transition_matrix ⇒ trace ⇒ context" where
 "anterior_context e t = posterior_sequence [[]] e 0 <> t"

lemma gexp_equiv_cexp_not_true:  "gexp_equiv (cexp2gexp a (Not (Bc True))) (gexp.Bc False)"
  by (simp add: gexp_equiv_def gval.simps)

lemma gexp_equiv_cexp_not_false:  "gexp_equiv (cexp2gexp a (Not (Bc False))) (gexp.Bc True)"
  by (simp add: gexp_equiv_def gval.simps)

lemma geq_to_ge: "Geq x = c r ⟹ (cexp2gexp r (c r)) = Ge r (L x)"
  by (metis cexp2gexp.simps(3) cexp2gexp.simps(6))

lemma leq_to_le: "Leq x = c r ⟹ (cexp2gexp r (c r)) = Le r (L x)"
  by (metis cexp2gexp.simps(4) cexp2gexp.simps(6))

lemma lt_to_lt: "Lt x = c r ⟹ (cexp2gexp r (c r)) = gexp.Gt (L x) r"
  by (metis cexp2gexp.simps(3))

lemma gt_to_gt: "Gt x = c r ⟹ (cexp2gexp r (c r)) = gexp.Gt r (L x)"
  by (metis cexp2gexp.simps(4))

lemma satisfiable_double_neg: "satisfiable (cexp.Not (cexp.Not x)) = satisfiable x"
  by (simp add: satisfiable_def cval_double_negation)

lemma gval_empty_r_neq_none[simp]: "gval (cexp2gexp r (conjoin ([[]] r))) s ≠ invalid"
  apply (case_tac r)
     apply (simp add: conjoin_def sorted_list_of_fset_def maybe_double_negation gval.simps)
    apply (case_tac x2)
  by (simp_all add: conjoin_def sorted_list_of_fset_def gval_gAnd gval.simps ValueEq_def)

lemma constrains_an_input_true: "constrains_an_input r ⟹ cval (conjoin ([[]] r)) r ia = true"
proof(induct r)
  case (L x)
  then show ?case by simp
next
  case (V x)
  then show ?case
    apply (case_tac x)
    by (simp_all add: conjoin_def cval_And cval_true sorted_list_of_fset_def gval.simps)
next
  case (Plus r1 r2)
  then show ?case
    by (simp add: conjoin_def sorted_list_of_fset_def cval_And cval_true)
```

```
next
  case (Minus r1 r2)
  then show ?case
    by (simp add: conjoin_def sorted_list_of_fset_def cval_And cval_true)
qed

lemma consistent_posterior_gives_consistent_medial: "consistent (posterior c x) ⟹ consistent (medial
c (Guard x))"
  apply (simp add: posterior_def Let_def posterior_separate_def)
  apply (case_tac "consistent (medial c (Guard x))")
   apply simp
  by (simp add: inconsistent_false)

lemma consistent_medial_gives_consistent_anterior: "consistent (medial c G) ⟹ consistent c"
  apply (simp add: consistent_def)
  by (metis (full_types) fBall_funion medial_def)

lemma medial_equivalent: "medial c (Guard t @ Guard t) = medial c (Guard t)"
  apply (rule ext)
  by (simp add: medial_append)

lemma transition_subsumes_self: "t ⊑c t"
  apply (simp add: subsumes_def)
  apply (simp only: posterior_separate_def Let_def posterior_def medial_equivalent)
  apply (case_tac "consistent (medial c (Guard t))")
   apply simp
  by simp

lemma medial_preserves_existing_elements: "x |∈| c r ⟹ x |∈| medial c G r "
  using anterior_subset_medial by blast

lemma remove_obsolete_constraints_input: "remove_obsolete_constraints c s (V (I i)) = {|Bc True|}"
  by (simp add: remove_obsolete_constraints_def)

lemma filter_simp: "I i |∉| fst |`| fset_of_list as ⟹
(∃n. aexp_constrains a (V (I n))) ∨ V aa = a ∨ fBex (fset_of_list as) (λx. V (fst x) = a) =
(∃n. aexp_constrains a (V (I n))) ∨ fBex (fset_of_list as) (λx. V (fst x) = a)"
  by auto

end
```

8 Infinite Streams

```
theory Stream
  imports Nat_Bijection
begin

codatatype (sset: 'a) stream =
  SCons (shd: 'a) (stl: "'a stream") (infixr "##" 65)
for
  map: smap
  rel: stream_all2

context
begin

— for code generation only
qualified definition smember :: "'a ⇒ 'a stream ⇒ bool" where
  [code_abbrev]: "smember x s ⟷ x ∈ sset s"
```

lemma *smember_code[code, simp]: "smember x (y ## s) = (if x = y then True else smember x s)"*
unfolding *smember_def* **by** *auto*

end

lemmas *smap_simps[simp] = stream.map_sel*
lemmas *shd_sset = stream.set_sel(1)*
lemmas *stl_sset = stream.set_sel(2)*

theorem *sset_induct[consumes 1, case_names shd stl, induct set: sset]:*
assumes *"y ∈ sset s"* **and** *"⋀s. P (shd s) s"* **and** *"⋀s y. ⟦y ∈ sset (stl s); P y (stl s)⟧ ⟹ P y s"*
shows *"P y s"*
using *assms* **by** *induct (metis stream.sel(1), auto)*

lemma *smap_ctr: "smap f s = x ## s' ⟷ f (shd s) = x ∧ smap f (stl s) = s'"*
by *(cases s) simp*

8.1 prepend list to stream

primrec *shift :: "'a list ⇒ 'a stream ⇒ 'a stream"* **(infixr** *"@-" 65)* **where**
"shift [] s = s"
| "shift (x # xs) s = x ## shift xs s"

lemma *smap_shift[simp]: "smap f (xs @- s) = map f xs @- smap f s"*
by *(induct xs) auto*

lemma *shift_append[simp]: "(xs @ ys) @- s = xs @- ys @- s"*
by *(induct xs) auto*

lemma *shift_simps[simp]:*
"shd (xs @- s) = (if xs = [] then shd s else hd xs)"
"stl (xs @- s) = (if xs = [] then stl s else tl xs @- s)"
by *(induct xs) auto*

lemma *sset_shift[simp]: "sset (xs @- s) = set xs ∪ sset s"*
by *(induct xs) auto*

lemma *shift_left_inj[simp]: "xs @- s1 = xs @- s2 ⟷ s1 = s2"*
by *(induct xs) auto*

8.2 set of streams with elements in some fixed set

context
notes *[[inductive_internals]]*
begin

coinductive_set
streams :: "'a set ⇒ 'a stream set"
for *A :: "'a set"*
where
Stream[intro!, simp, no_atp]: "⟦a ∈ A; s ∈ streams A⟧ ⟹ a ## s ∈ streams A"

end

lemma *in_streams: "stl s ∈ streams S ⟹ shd s ∈ S ⟹ s ∈ streams S"*
by *(cases s) auto*

lemma *streamsE: "s ∈ streams A ⟹ (shd s ∈ A ⟹ stl s ∈ streams A ⟹ P) ⟹ P"*
by *(erule streams.cases) simp_all*

lemma *Stream_image: "x ## y ∈ ((##) x') ‘ Y ⟷ x = x' ∧ y ∈ Y"*
by *auto*

lemma *shift_streams: "⟦w ∈ lists A; s ∈ streams A⟧ ⟹ w @- s ∈ streams A"*
by *(induct w) auto*

lemma *streams_Stream: "x ## s ∈ streams A ⟷ x ∈ A ∧ s ∈ streams A"*
by *(auto elim: streams.cases)*

lemma *streams_stl: "s ∈ streams A ⟹ stl s ∈ streams A"*
by *(cases s) (auto simp: streams_Stream)*

lemma *streams_shd: "s ∈ streams A ⟹ shd s ∈ A"*
by *(cases s) (auto simp: streams_Stream)*

lemma *sset_streams:*
assumes *"sset s ⊆ A"*
shows *"s ∈ streams A"*
using *assms* **proof** *(coinduction arbitrary: s)*
case *streams* **then show** *?case* **by** *(cases s) simp*
qed

lemma *streams_sset:*
assumes *"s ∈ streams A"*
shows *"sset s ⊆ A"*
proof
fix *x* **assume** *"x ∈ sset s"* **from** *this ⟨s ∈ streams A⟩* **show** *"x ∈ A"*
by *(induct s) (auto intro: streams_shd streams_stl)*
qed

lemma *streams_iff_sset: "s ∈ streams A ⟷ sset s ⊆ A"*
by *(metis sset_streams streams_sset)*

lemma *streams_mono: "s ∈ streams A ⟹ A ⊆ B ⟹ s ∈ streams B"*
unfolding *streams_iff_sset* **by** *auto*

lemma *streams_mono2: "S ⊆ T ⟹ streams S ⊆ streams T"*
by *(auto intro: streams_mono)*

lemma *smap_streams: "s ∈ streams A ⟹ (⋀x. x ∈ A ⟹ f x ∈ B) ⟹ smap f s ∈ streams B"*
unfolding *streams_iff_sset stream.set_map* **by** *auto*

lemma *streams_empty: "streams {} = {}"*
by *(auto elim: streams.cases)*

lemma *streams_UNIV[simp]: "streams UNIV = UNIV"*
by *(auto simp: streams_iff_sset)*

8.3 nth, take, drop for streams

primrec *snth :: "'a stream ⇒ nat ⇒ 'a"* **(infixl** *"!!" 100)* **where**
"s !! 0 = shd s"
| "s !! Suc n = stl s !! n"

lemma *snth_Stream: "(x ## s) !! Suc i = s !! i"*
by *simp*

lemma *snth_smap[simp]: "smap f s !! n = f (s !! n)"*
by *(induct n arbitrary: s) auto*

lemma *shift_snth_less[simp]: "p < length xs ⟹ (xs @- s) !! p = xs ! p"*

```
  by (induct p arbitrary: xs) (auto simp: hd_conv_nth nth_tl)

lemma shift_snth_ge[simp]: "p ≥ length xs ⟹ (xs @- s) !! p = s !! (p - length xs)"
  by (induct p arbitrary: xs) (auto simp: Suc_diff_eq_diff_pred)

lemma shift_snth: "(xs @- s) !! n = (if n < length xs then xs ! n else s !! (n - length xs))"
  by auto

lemma snth_sset[simp]: "s !! n ∈ sset s"
  by (induct n arbitrary: s) (auto intro: shd_sset stl_sset)

lemma sset_range: "sset s = range (snth s)"
proof (intro equalityI subsetI)
  fix x assume "x ∈ sset s"
  thus "x ∈ range (snth s)"
  proof (induct s)
    case (stl s x)
    then obtain n where "x = stl s !! n" by auto
    thus ?case by (auto intro: range_eqI[of _ _ "Suc n"])
  qed (auto intro: range_eqI[of _ _ 0])
qed auto

lemma streams_iff_snth: "s ∈ streams X ⟷ (∀n. s !! n ∈ X)"
  by (force simp: streams_iff_sset sset_range)

lemma snth_in: "s ∈ streams X ⟹ s !! n ∈ X"
  by (simp add: streams_iff_snth)

primrec stake :: "nat ⇒ 'a stream ⇒ 'a list" where
  "stake 0 s = []"
| "stake (Suc n) s = shd s # stake n (stl s)"

lemma length_stake[simp]: "length (stake n s) = n"
  by (induct n arbitrary: s) auto

lemma stake_smap[simp]: "stake n (smap f s) = map f (stake n s)"
  by (induct n arbitrary: s) auto

lemma take_stake: "take n (stake m s) = stake (min n m) s"
proof (induct m arbitrary: s n)
  case (Suc m) thus ?case by (cases n) auto
qed simp

primrec sdrop :: "nat ⇒ 'a stream ⇒ 'a stream" where
  "sdrop 0 s = s"
| "sdrop (Suc n) s = sdrop n (stl s)"

lemma sdrop_simps[simp]:
  "shd (sdrop n s) = s !! n" "stl (sdrop n s) = sdrop (Suc n) s"
  by (induct n arbitrary: s)  auto

lemma sdrop_smap[simp]: "sdrop n (smap f s) = smap f (sdrop n s)"
  by (induct n arbitrary: s) auto

lemma sdrop_stl: "sdrop n (stl s) = stl (sdrop n s)"
  by (induct n) auto

lemma drop_stake: "drop n (stake m s) = stake (m - n) (sdrop n s)"
proof (induct m arbitrary: s n)
  case (Suc m) thus ?case by (cases n) auto
qed simp
```

```
lemma stake_sdrop: "stake n s @- sdrop n s = s"
  by (induct n arbitrary: s) auto

lemma id_stake_snth_sdrop:
  "s = stake i s @- s !! i ## sdrop (Suc i) s"
  by (subst stake_sdrop[symmetric, of _ i]) (metis sdrop_simps stream.collapse)

lemma smap_alt: "smap f s = s' ⟷ (∀n. f (s !! n) = s' !! n)" (is "?L = ?R")
proof
  assume ?R
  then have "⋀n. smap f (sdrop n s) = sdrop n s'"
    by coinduction (auto intro: exI[of _ 0] simp del: sdrop.simps(2))
  then show ?L using sdrop.simps(1) by metis
qed auto

lemma stake_invert_Nil[iff]: "stake n s = [] ⟷ n = 0"
  by (induct n) auto

lemma sdrop_shift: "sdrop i (w @- s) = drop i w @- sdrop (i - length w) s"
  by (induct i arbitrary: w s) (auto simp: drop_tl drop_Suc neq_Nil_conv)

lemma stake_shift: "stake i (w @- s) = take i w @ stake (i - length w) s"
  by (induct i arbitrary: w s) (auto simp: neq_Nil_conv)

lemma stake_add[simp]: "stake m s @ stake n (sdrop m s) = stake (m + n) s"
  by (induct m arbitrary: s) auto

lemma sdrop_add[simp]: "sdrop n (sdrop m s) = sdrop (m + n) s"
  by (induct m arbitrary: s) auto

lemma sdrop_snth: "sdrop n s !! m = s !! (n + m)"
  by (induct n arbitrary: m s) auto

partial_function (tailrec) sdrop_while :: "('a ⇒ bool) ⇒ 'a stream ⇒ 'a stream" where
  "sdrop_while P s = (if P (shd s) then sdrop_while P (stl s) else s)"

lemma sdrop_while_SCons[code]:
  "sdrop_while P (a ## s) = (if P a then sdrop_while P s else a ## s)"
  by (subst sdrop_while.simps) simp

lemma sdrop_while_sdrop_LEAST:
  assumes "∃n. P (s !! n)"
  shows "sdrop_while (Not ∘ P) s = sdrop (LEAST n. P (s !! n)) s"
proof -
  from assms obtain m where "P (s !! m)" "⋀n. P (s !! n) ⟹ m ≤ n"
    and *: "(LEAST n. P (s !! n)) = m" by atomize_elim (auto intro: LeastI Least_le)
  thus ?thesis unfolding *
  proof (induct m arbitrary: s)
    case (Suc m)
    hence "sdrop_while (Not ∘ P) (stl s) = sdrop m (stl s)"
      by (metis (full_types) not_less_eq_eq snth.simps(2))
    moreover from Suc(3) have "¬ (P (s !! 0))" by blast
    ultimately show ?case by (subst sdrop_while.simps) simp
  qed (metis comp_apply sdrop.simps(1) sdrop_while.simps snth.simps(1))
qed

primcorec sfilter where
  "shd (sfilter P s) = shd (sdrop_while (Not ∘ P) s)"
| "stl (sfilter P s) = sfilter P (stl (sdrop_while (Not ∘ P) s))"
```

```
lemma sfilter_Stream: "sfilter P (x ## s) = (if P x then x ## sfilter P s else sfilter P s)"
proof (cases "P x")
  case True thus ?thesis by (subst sfilter.ctr) (simp add: sdrop_while_SCons)
next
  case False thus ?thesis by (subst (1 2) sfilter.ctr) (simp add: sdrop_while_SCons)
qed
```

8.4 unary predicates lifted to streams

```
definition "stream_all P s = (∀p. P (s !! p))"

lemma stream_all_iff[iff]: "stream_all P s ⟷ Ball (sset s) P"
  unfolding stream_all_def sset_range by auto

lemma stream_all_shift[simp]: "stream_all P (xs @- s) = (list_all P xs ∧ stream_all P s)"
  unfolding stream_all_iff list_all_iff by auto

lemma stream_all_Stream: "stream_all P (x ## X) ⟷ P x ∧ stream_all P X"
  by simp
```

8.5 recurring stream out of a list

```
primcorec cycle :: "'a list ⇒ 'a stream" where
  "shd (cycle xs) = hd xs"
| "stl (cycle xs) = cycle (tl xs @ [hd xs])"

lemma cycle_decomp: "u ≠ [] ⟹ cycle u = u @- cycle u"
proof (coinduction arbitrary: u)
  case Eq_stream then show ?case using stream.collapse[of "cycle u"]
    by (auto intro!: exI[of _ "tl u @ [hd u]"])
qed

lemma cycle_Cons[code]: "cycle (x # xs) = x ## cycle (xs @ [x])"
  by (subst cycle.ctr) simp

lemma cycle_rotated: "⟦v ≠ []; cycle u = v @- s⟧ ⟹ cycle (tl u @ [hd u]) = tl v @- s"
  by (auto dest: arg_cong[of _ _ stl])

lemma stake_append: "stake n (u @- s) = take (min (length u) n) u @ stake (n - length u) s"
proof (induct n arbitrary: u)
  case (Suc n) thus ?case by (cases u) auto
qed auto

lemma stake_cycle_le[simp]:
  assumes "u ≠ []" "n < length u"
  shows "stake n (cycle u) = take n u"
using min_absorb2[OF less_imp_le_nat[OF assms(2)]]
  by (subst cycle_decomp[OF assms(1)], subst stake_append) auto

lemma stake_cycle_eq[simp]: "u ≠ [] ⟹ stake (length u) (cycle u) = u"
  by (subst cycle_decomp) (auto simp: stake_shift)

lemma sdrop_cycle_eq[simp]: "u ≠ [] ⟹ sdrop (length u) (cycle u) = cycle u"
  by (subst cycle_decomp) (auto simp: sdrop_shift)

lemma stake_cycle_eq_mod_0[simp]: "⟦u ≠ []; n mod length u = 0⟧ ⟹
   stake n (cycle u) = concat (replicate (n div length u) u)"
  by (induct "n div length u" arbitrary: n u)
    (auto simp: stake_add [symmetric] mod_eq_0_iff_dvd elim!: dvdE)

lemma sdrop_cycle_eq_mod_0[simp]: "⟦u ≠ []; n mod length u = 0⟧ ⟹
```

```
   sdrop n (cycle u) = cycle u"
  by (induct "n div length u" arbitrary: n u)
    (auto simp: sdrop_add [symmetric] mod_eq_0_iff_dvd elim!: dvdE)

lemma stake_cycle: "u ≠ [] ⟹
   stake n (cycle u) = concat (replicate (n div length u) u) @ take (n mod length u) u"
  by (subst div_mult_mod_eq[of n "length u", symmetric], unfold stake_add[symmetric]) auto

lemma sdrop_cycle: "u ≠ [] ⟹ sdrop n (cycle u) = cycle (rotate (n mod length u) u)"
  by (induct n arbitrary: u) (auto simp: rotate1_rotate_swap rotate1_hd_tl rotate_conv_mod[symmetric])

lemma sset_cycle[simp]:
  assumes "xs ≠ []"
  shows "sset (cycle xs) = set xs"
proof (intro set_eqI iffI)
  fix x
  assume "x ∈ sset (cycle xs)"
  then show "x ∈ set xs" using assms
    by (induction "cycle xs" arbitrary: xs rule: sset_induct) (fastforce simp: neq_Nil_conv)+
qed (metis assms UnI1 cycle_decomp sset_shift)
```

8.6 iterated application of a function

```
primcorec siterate where
  "shd (siterate f x) = x"
| "stl (siterate f x) = siterate f (f x)"

lemma stake_Suc: "stake (Suc n) s = stake n s @ [s !! n]"
  by (induct n arbitrary: s) auto

lemma snth_siterate[simp]: "siterate f x !! n = (f^^n) x"
  by (induct n arbitrary: x) (auto simp: funpow_swap1)

lemma sdrop_siterate[simp]: "sdrop n (siterate f x) = siterate f ((f^^n) x)"
  by (induct n arbitrary: x) (auto simp: funpow_swap1)

lemma stake_siterate[simp]: "stake n (siterate f x) = map (λn. (f^^n) x) [0 ..< n]"
  by (induct n arbitrary: x) (auto simp del: stake.simps(2) simp: stake_Suc)

lemma sset_siterate: "sset (siterate f x) = {(f^^n) x | n. True}"
  by (auto simp: sset_range)

lemma smap_siterate: "smap f (siterate f x) = siterate f (f x)"
  by (coinduction arbitrary: x) auto
```

8.7 stream repeating a single element

```
abbreviation "sconst ≡ siterate id"

lemma shift_replicate_sconst[simp]: "replicate n x @- sconst x = sconst x"
  by (subst (3) stake_sdrop[symmetric]) (simp add: map_replicate_trivial)

lemma sset_sconst[simp]: "sset (sconst x) = {x}"
  by (simp add: sset_siterate)

lemma sconst_alt: "s = sconst x ⟷ sset s = {x}"
proof
  assume "sset s = {x}"
  then show "s = sconst x"
  proof (coinduction arbitrary: s)
    case Eq_stream
```

```
    then have "shd s = x" "sset (stl s) ⊆ {x}" by (cases s; auto)+
    then have "sset (stl s) = {x}" by (cases "stl s") auto
    with ⟨shd s = x⟩ show ?case by auto
  qed
qed simp

lemma sconst_cycle: "sconst x = cycle [x]"
  by coinduction auto

lemma smap_sconst: "smap f (sconst x) = sconst (f x)"
  by coinduction auto

lemma sconst_streams: "x ∈ A ⟹ sconst x ∈ streams A"
  by (simp add: streams_iff_sset)

lemma streams_empty_iff: "streams S = {} ⟷ S = {}"
proof safe
  fix x assume "x ∈ S" "streams S = {}"
  then have "sconst x ∈ streams S"
    by (intro sconst_streams)
  then show "x ∈ {}"
    unfolding ⟨streams S = {}⟩ by simp
qed (auto simp: streams_empty)
```

8.8 stream of natural numbers

```
abbreviation "fromN ≡ siterate Suc"

abbreviation "nats ≡ fromN 0"

lemma sset_fromN[simp]: "sset (fromN n) = {n ..}"
  by (auto simp add: sset_siterate le_iff_add)

lemma stream_smap_fromN: "s = smap (λj. let i = j - n in s !! i) (fromN n)"
  by (coinduction arbitrary: s n)
    (force simp: neq_Nil_conv Let_def Suc_diff_Suc simp flip: snth.simps(2)
      intro: stream.map_cong split: if_splits)

lemma stream_smap_nats: "s = smap (snth s) nats"
  using stream_smap_fromN[where n = 0] by simp
```

8.9 flatten a stream of lists

```
primcorec flat where
  "shd (flat ws) = hd (shd ws)"
| "stl (flat ws) = flat (if tl (shd ws) = [] then stl ws else tl (shd ws) ## stl ws)"

lemma flat_Cons[simp, code]: "flat ((x # xs) ## ws) = x ## flat (if xs = [] then ws else xs ## ws)"
  by (subst flat.ctr) simp

lemma flat_Stream[simp]: "xs ≠ [] ⟹ flat (xs ## ws) = xs @- flat ws"
  by (induct xs) auto

lemma flat_unfold: "shd ws ≠ [] ⟹ flat ws = shd ws @- flat (stl ws)"
  by (cases ws) auto

lemma flat_snth: "∀xs ∈ sset s. xs ≠ [] ⟹ flat s !! n = (if n < length (shd s) then
  shd s ! n else flat (stl s) !! (n - length (shd s)))"
  by (metis flat_unfold not_less shd_sset shift_snth_ge shift_snth_less)

lemma sset_flat[simp]: "∀xs ∈ sset s. xs ≠ [] ⟹
```

```
  sset (flat s) = (⋃xs ∈ sset s. set xs)" (is "?P ⟹ ?L = ?R")
proof safe
  fix x assume ?P "x ∈ ?L"
  then obtain m where "x = flat s !! m" by (metis image_iff sset_range)
  with ‹?P› obtain n m' where "x = s !! n ! m'" "m' < length (s !! n)"
  proof (atomize_elim, induct m arbitrary: s rule: less_induct)
    case (less y)
    thus ?case
    proof (cases "y < length (shd s)")
      case True thus ?thesis by (metis flat_snth less(2,3) snth.simps(1))
    next
      case False
      hence "x = flat (stl s) !! (y - length (shd s))" by (metis less(2,3) flat_snth)
      moreover
      { from less(2) have *: "length (shd s) > 0" by (cases s) simp_all
        with False have "y > 0" by (cases y) simp_all
        with * have "y - length (shd s) < y" by simp
      }
      moreover have "∀xs ∈ sset (stl s). xs ≠ []" using less(2) by (cases s) auto
      ultimately have "∃n m'. x = stl s !! n ! m' ∧ m' < length (stl s !! n)" by (intro less(1)) auto
      thus ?thesis by (metis snth.simps(2))
    qed
  qed
  thus "x ∈ ?R" by (auto simp: sset_range dest!: nth_mem)
next
  fix x xs assume "xs ∈ sset s" ?P "x ∈ set xs" thus "x ∈ ?L"
    by (induct rule: sset_induct)
      (metis UnI1 flat_unfold shift.simps(1) sset_shift,
       metis UnI2 flat_unfold shd_sset stl_sset sset_shift)
qed
```

8.10 merge a stream of streams

```
definition smerge :: "'a stream stream ⇒ 'a stream" where
  "smerge ss = flat (smap (λn. map (λs. s !! n) (stake (Suc n) ss) @ stake n (ss !! n)) nats)"

lemma stake_nth[simp]: "m < n ⟹ stake n s ! m = s !! m"
  by (induct n arbitrary: s m) (auto simp: nth_Cons', metis Suc_pred snth.simps(2))

lemma snth_sset_smerge: "ss !! n !! m ∈ sset (smerge ss)"
proof (cases "n ≤ m")
  case False thus ?thesis unfolding smerge_def
    by (subst sset_flat)
      (auto simp: stream.set_map in_set_conv_nth simp del: stake.simps
        intro!: exI[of _ n, OF disjI2] exI[of _ m, OF mp])
next
  case True thus ?thesis unfolding smerge_def
    by (subst sset_flat)
      (auto simp: stream.set_map in_set_conv_nth image_iff simp del: stake.simps snth.simps
        intro!: exI[of _ m, OF disjI1] bexI[of _ "ss !! n"] exI[of _ n, OF mp])
qed

lemma sset_smerge: "sset (smerge ss) = UNION (sset ss) sset"
proof safe
  fix x assume "x ∈ sset (smerge ss)"
  thus "x ∈ UNION (sset ss) sset"
    unfolding smerge_def by (subst (asm) sset_flat)
      (auto simp: stream.set_map in_set_conv_nth sset_range simp del: stake.simps, fast+)
next
  fix s x assume "s ∈ sset ss" "x ∈ sset s"
  thus "x ∈ sset (smerge ss)" using snth_sset_smerge by (auto simp: sset_range)
```

qed

8.11 product of two streams

definition *sproduct :: "'a stream ⇒ 'b stream ⇒ ('a × 'b) stream"* **where**
"sproduct s1 s2 = smerge (smap (λx. smap (Pair x) s2) s1)"

lemma *sset_sproduct: "sset (sproduct s1 s2) = sset s1 × sset s2"*
unfolding *sproduct_def sset_smerge* **by** *(auto simp: stream.set_map)*

8.12 interleave two streams

primcorec *sinterleave* **where**
"shd (sinterleave s1 s2) = shd s1"
| "stl (sinterleave s1 s2) = sinterleave s2 (stl s1)"

lemma *sinterleave_code[code]:*
"sinterleave (x ## s1) s2 = x ## sinterleave s2 s1"
by *(subst sinterleave.ctr) simp*

lemma *sinterleave_snth[simp]:*
"even n ⟹ sinterleave s1 s2 !! n = s1 !! (n div 2)"
"odd n ⟹ sinterleave s1 s2 !! n = s2 !! (n div 2)"
by *(induct n arbitrary: s1 s2) simp_all*

lemma *sset_sinterleave: "sset (sinterleave s1 s2) = sset s1 ∪ sset s2"*
proof *(intro equalityI subsetI)*
fix *x* **assume** *"x ∈ sset (sinterleave s1 s2)"*
then obtain *n* **where** *"x = sinterleave s1 s2 !! n"* **unfolding** *sset_range* **by** *blast*
thus *"x ∈ sset s1 ∪ sset s2"* **by** *(cases "even n") auto*
next
fix *x* **assume** *"x ∈ sset s1 ∪ sset s2"*
thus *"x ∈ sset (sinterleave s1 s2)"*
proof
assume *"x ∈ sset s1"*
then obtain *n* **where** *"x = s1 !! n"* **unfolding** *sset_range* **by** *blast*
hence *"sinterleave s1 s2 !! (2 * n) = x"* **by** *simp*
thus *?thesis* **unfolding** *sset_range* **by** *blast*
next
assume *"x ∈ sset s2"*
then obtain *n* **where** *"x = s2 !! n"* **unfolding** *sset_range* **by** *blast*
hence *"sinterleave s1 s2 !! (2 * n + 1) = x"* **by** *simp*
thus *?thesis* **unfolding** *sset_range* **by** *blast*
qed
qed

8.13 zip

primcorec *szip* **where**
"shd (szip s1 s2) = (shd s1, shd s2)"
| "stl (szip s1 s2) = szip (stl s1) (stl s2)"

lemma *szip_unfold[code]: "szip (a ## s1) (b ## s2) = (a, b) ## (szip s1 s2)"*
by *(subst szip.ctr) simp*

lemma *snth_szip[simp]: "szip s1 s2 !! n = (s1 !! n, s2 !! n)"*
by *(induct n arbitrary: s1 s2) auto*

lemma *stake_szip[simp]:*
"stake n (szip s1 s2) = zip (stake n s1) (stake n s2)"
by *(induct n arbitrary: s1 s2) auto*

lemma *sdrop_szip[simp]: "sdrop n (szip s1 s2) = szip (sdrop n s1) (sdrop n s2)"*
by *(induct n arbitrary: s1 s2) auto*

lemma *smap_szip_fst:*
"smap (λx. f (fst x)) (szip s1 s2) = smap f s1"
by *(coinduction arbitrary: s1 s2) auto*

lemma *smap_szip_snd:*
"smap (λx. g (snd x)) (szip s1 s2) = smap g s2"
by *(coinduction arbitrary: s1 s2) auto*

8.14 zip via function

primcorec *smap2* **where**
"shd (smap2 f s1 s2) = f (shd s1) (shd s2)"
| "stl (smap2 f s1 s2) = smap2 f (stl s1) (stl s2)"

lemma *smap2_unfold[code]:*
"smap2 f (a ## s1) (b ## s2) = f a b ## (smap2 f s1 s2)"
by *(subst smap2.ctr) simp*

lemma *smap2_szip:*
"smap2 f s1 s2 = smap (case_prod f) (szip s1 s2)"
by *(coinduction arbitrary: s1 s2) auto*

lemma *smap_smap2[simp]:*
"smap f (smap2 g s1 s2) = smap2 (λx y. f (g x y)) s1 s2"
unfolding *smap2_szip stream.map_comp o_def split_def* **..**

lemma *smap2_alt:*
"(smap2 f s1 s2 = s) = (∀n. f (s1 !! n) (s2 !! n) = s !! n)"
unfolding *smap2_szip smap_alt* **by** *auto*

lemma *snth_smap2[simp]:*
"smap2 f s1 s2 !! n = f (s1 !! n) (s2 !! n)"
by *(induct n arbitrary: s1 s2) auto*

lemma *stake_smap2[simp]:*
"stake n (smap2 f s1 s2) = map (case_prod f) (zip (stake n s1) (stake n s2))"
by *(induct n arbitrary: s1 s2) auto*

lemma *sdrop_smap2[simp]:*
"sdrop n (smap2 f s1 s2) = smap2 f (sdrop n s1) (sdrop n s2)"
by *(induct n arbitrary: s1 s2) auto*

end

9 List prefixes, suffixes, and homeomorphic embedding

theory *Sublist*
imports *Main*
begin

9.1 Prefix order on lists

definition *prefix :: "'a list ⇒ 'a list ⇒ bool"*
where *"prefix xs ys ⟷ (∃zs. ys = xs @ zs)"*

definition *strict_prefix :: "'a list ⇒ 'a list ⇒ bool"*

where *"strict_prefix xs ys ⟷ prefix xs ys ∧ xs ≠ ys"*

interpretation *prefix_order: order prefix strict_prefix*
by *standard (auto simp: prefix_def strict_prefix_def)*

interpretation *prefix_bot: order_bot Nil prefix strict_prefix*
by *standard (simp add: prefix_def)*

lemma *prefixI [intro?]: "ys = xs @ zs ⟹ prefix xs ys"*
unfolding *prefix_def* **by** *blast*

lemma *prefixE [elim?]:*
assumes *"prefix xs ys"*
obtains *zs* **where** *"ys = xs @ zs"*
using *assms* **unfolding** *prefix_def* **by** *blast*

lemma *strict_prefixI' [intro?]: "ys = xs @ z # zs ⟹ strict_prefix xs ys"*
unfolding *strict_prefix_def prefix_def* **by** *blast*

lemma *strict_prefixE' [elim?]:*
assumes *"strict_prefix xs ys"*
obtains *z zs* **where** *"ys = xs @ z # zs"*
proof -
from ‹*strict_prefix xs ys*› **obtain** *us* **where** *"ys = xs @ us"* **and** *"xs ≠ ys"*
unfolding *strict_prefix_def prefix_def* **by** *blast*
with *that* **show** *?thesis* **by** *(auto simp add: neq_Nil_conv)*
qed

lemma *strict_prefixI [intro?]: "prefix xs ys ⟹ xs ≠ ys ⟹ strict_prefix xs ys"*
by(*fact prefix_order.le_neq_trans*)

lemma *strict_prefixE [elim?]:*
fixes *xs ys :: "'a list"*
assumes *"strict_prefix xs ys"*
obtains *"prefix xs ys"* **and** *"xs ≠ ys"*
using *assms* **unfolding** *strict_prefix_def* **by** *blast*

9.2 Basic properties of prefixes

theorem *Nil_prefix [simp]: "prefix [] xs"*
by *(fact prefix_bot.bot_least)*

theorem *prefix_Nil [simp]: "(prefix xs []) = (xs = [])"*
by *(fact prefix_bot.bot_unique)*

lemma *prefix_snoc [simp]: "prefix xs (ys @ [y]) ⟷ xs = ys @ [y] ∨ prefix xs ys"*
proof
assume *"prefix xs (ys @ [y])"*
then obtain *zs* **where** *zs: "ys @ [y] = xs @ zs"* **..**
show *"xs = ys @ [y] ∨ prefix xs ys"*
by *(metis append_Nil2 butlast_append butlast_snoc prefixI zs)*
next
assume *"xs = ys @ [y] ∨ prefix xs ys"*
then show *"prefix xs (ys @ [y])"*
by *(metis prefix_order.eq_iff prefix_order.order_trans prefixI)*
qed

lemma *Cons_prefix_Cons [simp]: "prefix (x # xs) (y # ys) = (x = y ∧ prefix xs ys)"*
by *(auto simp add: prefix_def)*

```
lemma prefix_code [code]:
  "prefix [] xs ⟷ True"
  "prefix (x # xs) [] ⟷ False"
  "prefix (x # xs) (y # ys) ⟷ x = y ∧ prefix xs ys"
  by simp_all

lemma same_prefix_prefix [simp]: "prefix (xs @ ys) (xs @ zs) = prefix ys zs"
  by (induct xs) simp_all

lemma same_prefix_nil [simp]: "prefix (xs @ ys) xs = (ys = [])"
  by (metis append_Nil2 append_self_conv prefix_order.eq_iff prefixI)

lemma prefix_prefix [simp]: "prefix xs ys ⟹ prefix xs (ys @ zs)"
  unfolding prefix_def by fastforce

lemma append_prefixD: "prefix (xs @ ys) zs ⟹ prefix xs zs"
  by (auto simp add: prefix_def)

theorem prefix_Cons: "prefix xs (y # ys) = (xs = [] ∨ (∃zs. xs = y # zs ∧ prefix zs ys))"
  by (cases xs) (auto simp add: prefix_def)

theorem prefix_append:
  "prefix xs (ys @ zs) = (prefix xs ys ∨ (∃us. xs = ys @ us ∧ prefix us zs))"
  apply (induct zs rule: rev_induct)
   apply force
  apply (simp flip: append_assoc)
  apply (metis append_eq_appendI)
  done

lemma append_one_prefix:
  "prefix xs ys ⟹ length xs < length ys ⟹ prefix (xs @ [ys ! length xs]) ys"
  proof (unfold prefix_def)
    assume a1: "∃zs. ys = xs @ zs"
    then obtain sk :: "'a list" where sk: "ys = xs @ sk" by fastforce
    assume a2: "length xs < length ys"
    have f1: "⋀v. ([]::'a list) @ v = v" using append_Nil2 by simp
    have "[] ≠ sk" using a1 a2 sk less_not_refl by force
    hence "∃v. xs @ hd sk # v = ys" using sk by (metis hd_Cons_tl)
    thus "∃zs. ys = (xs @ [ys ! length xs]) @ zs" using f1 by fastforce
  qed

theorem prefix_length_le: "prefix xs ys ⟹ length xs ≤ length ys"
  by (auto simp add: prefix_def)

lemma prefix_same_cases:
  "prefix (xs1::'a list) ys ⟹ prefix xs2 ys ⟹ prefix xs1 xs2 ∨ prefix xs2 xs1"
  unfolding prefix_def by (force simp: append_eq_append_conv2)

lemma prefix_length_prefix:
  "prefix ps xs ⟹ prefix qs xs ⟹ length ps ≤ length qs ⟹ prefix ps qs"
by (auto simp: prefix_def) (metis append_Nil2 append_eq_append_conv_if)

lemma set_mono_prefix: "prefix xs ys ⟹ set xs ⊆ set ys"
  by (auto simp add: prefix_def)

lemma take_is_prefix: "prefix (take n xs) xs"
  unfolding prefix_def by (metis append_take_drop_id)

lemma prefixeq_butlast: "prefix (butlast xs) xs"
by (simp add: butlast_conv_take take_is_prefix)
```

```
lemma map_mono_prefix: "prefix xs ys ⟹ prefix (map f xs) (map f ys)"
by (auto simp: prefix_def)

lemma filter_mono_prefix: "prefix xs ys ⟹ prefix (filter P xs) (filter P ys)"
by (auto simp: prefix_def)

lemma sorted_antimono_prefix: "prefix xs ys ⟹ sorted ys ⟹ sorted xs"
by (metis sorted_append prefix_def)

lemma prefix_length_less: "strict_prefix xs ys ⟹ length xs < length ys"
  by (auto simp: strict_prefix_def prefix_def)

lemma prefix_snocD: "prefix (xs@[x]) ys ⟹ strict_prefix xs ys"
  by (simp add: strict_prefixI' prefix_order.dual_order.strict_trans1)

lemma strict_prefix_simps [simp, code]:
  "strict_prefix xs [] ⟷ False"
  "strict_prefix [] (x # xs) ⟷ True"
  "strict_prefix (x # xs) (y # ys) ⟷ x = y ∧ strict_prefix xs ys"
  by (simp_all add: strict_prefix_def cong: conj_cong)

lemma take_strict_prefix: "strict_prefix xs ys ⟹ strict_prefix (take n xs) ys"
proof (induct n arbitrary: xs ys)
  case 0
  then show ?case by (cases ys) simp_all
next
  case (Suc n)
  then show ?case by (metis prefix_order.less_trans strict_prefixI take_is_prefix)
qed

lemma not_prefix_cases:
  assumes pfx: "¬ prefix ps ls"
  obtains
    (c1) "ps ≠ []" and "ls = []"
  | (c2) a as x xs where "ps = a#as" and "ls = x#xs" and "x = a" and "¬ prefix as xs"
  | (c3) a as x xs where "ps = a#as" and "ls = x#xs" and "x ≠ a"
proof (cases ps)
  case Nil
  then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ⟨ps = a#as⟩
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append_Nil2 pfx c1 same_prefix_nil)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases "x = a")
      case True
      have "¬ prefix as xs" using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
qed

lemma not_prefix_induct [consumes 1, case_names Nil Neq Eq]:
```

```
  assumes np: "¬ prefix ps ls"
    and base: "⋀x xs. P (x#xs) []"
    and r1: "⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)"
    and r2: "⋀x xs y ys. ⟦ x = y; ¬ prefix xs ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)"
  shows "P ps ls" using np
proof (induct ls arbitrary: ps)
  case Nil
  then show ?case
    by (auto simp: neq_Nil_conv elim!: not_prefix_cases intro!: base)
next
  case (Cons y ys)
  then have npfx: "¬ prefix ps (y # ys)" by simp
  then obtain x xs where pv: "ps = x # xs"
    by (rule not_prefix_cases) auto
  show ?case by (metis Cons.hyps Cons_prefix_Cons npfx pv r1 r2)
qed
```

9.3 Prefixes

```
primrec prefixes where
"prefixes [] = [[]]" |
"prefixes (x#xs) = [] # map ((#) x) (prefixes xs)"

lemma in_set_prefixes[simp]: "xs ∈ set (prefixes ys) ⟷ prefix xs ys"
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by (cases ys) auto
next
  case (Cons a xs)
  then show ?case by (cases ys) auto
qed

lemma length_prefixes[simp]: "length (prefixes xs) = length xs+1"
  by (induction xs) auto

lemma distinct_prefixes [intro]: "distinct (prefixes xs)"
  by (induction xs) (auto simp: distinct_map)

lemma prefixes_snoc [simp]: "prefixes (xs@[x]) = prefixes xs @ [xs@[x]]"
  by (induction xs) auto

lemma prefixes_not_Nil [simp]: "prefixes xs ≠ []"
  by (cases xs) auto

lemma hd_prefixes [simp]: "hd (prefixes xs) = []"
  by (cases xs) simp_all

lemma last_prefixes [simp]: "last (prefixes xs) = xs"
  by (induction xs) (simp_all add: last_map)

lemma prefixes_append:
  "prefixes (xs @ ys) = prefixes xs @ map (λys'. xs @ ys') (tl (prefixes ys))"
proof (induction xs)
  case Nil
  thus ?case by (cases ys) auto
qed simp_all

lemma prefixes_eq_snoc:
  "prefixes ys = xs @ [x] ⟷
  (ys = [] ∧ xs = [] ∨ (∃z zs. ys = zs@[z] ∧ xs = prefixes zs)) ∧ x = ys"
  by (cases ys rule: rev_cases) auto
```

```
lemma prefixes_tailrec [code]:
  "prefixes xs = rev (snd (foldl (λ(acc1, acc2) x. (x#acc1, rev (x#acc1)#acc2)) ([],[[]]) xs))"
proof -
  have "foldl (λ(acc1, acc2) x. (x#acc1, rev (x#acc1)#acc2)) (ys, rev ys # zs) xs =
        (rev xs @ ys, rev (map (λas. rev ys @ as) (prefixes xs)) @ zs)" for ys zs
  proof (induction xs arbitrary: ys zs)
    case (Cons x xs ys zs)
    from Cons.IH[of "x # ys" "rev ys # zs"]
      show ?case by (simp add: o_def)
  qed simp_all
  from this [of "[]" "[]"] show ?thesis by simp
qed

lemma set_prefixes_eq: "set (prefixes xs) = {ys. prefix ys xs}"
  by auto

lemma card_set_prefixes [simp]: "card (set (prefixes xs)) = Suc (length xs)"
  by (subst distinct_card) auto

lemma set_prefixes_append:
  "set (prefixes (xs @ ys)) = set (prefixes xs) ∪ {xs @ ys' |ys'. ys' ∈ set (prefixes ys)}"
  by (subst prefixes_append, cases ys) auto
```

9.4 Longest Common Prefix

```
definition Longest_common_prefix :: "'a list set ⇒ 'a list" where
"Longest_common_prefix L = (ARG_MAX length ps. ∀xs ∈ L. prefix ps xs)"

lemma Longest_common_prefix_ex: "L ≠ {} ⟹
  ∃ps. (∀xs ∈ L. prefix ps xs) ∧ (∀qs. (∀xs ∈ L. prefix qs xs) ⟶ size qs ≤ size ps)"
  (is "_ ⟹ ∃ps. ?P L ps")
proof(induction "LEAST n. ∃xs ∈L. n = length xs" arbitrary: L)
  case 0
  have "[] ∈ L" using "0.hyps" LeastI[of "λn. ∃xs∈L. n = length xs"] ‹L ≠ {}›
    by auto
  hence "?P L []" by(auto)
  thus ?case ..
next
  case (Suc n)
  let ?EX = "λn. ∃xs∈L. n = length xs"
  obtain x xs where xxs: "x#xs ∈ L" "size xs = n" using Suc.prems Suc.hyps(2)
    by(metis LeastI_ex[of ?EX] Suc_length_conv ex_in_conv)
  hence "[] ∉ L" using Suc.hyps(2) by auto
  show ?case
  proof (cases "∀xs ∈ L. ∃ys. xs = x#ys")
    case True
    let ?L = "{ys. x#ys ∈ L}"
    have 1: "(LEAST n. ∃xs ∈ ?L. n = length xs) = n"
      using xxs Suc.prems Suc.hyps(2) Least_le[of "?EX"]
      by - (rule Least_equality, fastforce+)
    have 2: "?L ≠ {}" using ‹x # xs ∈ L› by auto
    from Suc.hyps(1)[OF 1[symmetric] 2] obtain ps where IH: "?P ?L ps" ..
    { fix qs
      assume "∀qs. (∀xa. x # xa ∈ L ⟶ prefix qs xa) ⟶ length qs ≤ length ps"
      and "∀xs∈L. prefix qs xs"
      hence "length (tl qs) ≤ length ps"
        by (metis Cons_prefix_Cons hd_Cons_tl list.sel(2) Nil_prefix)
      hence "length qs ≤ Suc (length ps)" by auto
    }
    hence "?P L (x#ps)" using True IH by auto
```

```
    thus ?thesis ..
  next
    case False
    then obtain y ys where yys: "x≠y" "y#ys ∈ L" using ‹[] ∉ L›
      by (auto) (metis list.exhaust)
    have "∀qs. (∀xs∈L. prefix qs xs) ⟶ qs = []" using yys ‹x#xs ∈ L›
      by auto (metis Cons_prefix_Cons prefix_Cons)
    hence "?P L []" by auto
    thus ?thesis ..
  qed
qed

lemma Longest_common_prefix_unique: "L ≠ {} ⟹
  ∃! ps. (∀xs ∈ L. prefix ps xs) ∧ (∀qs. (∀xs ∈ L. prefix qs xs) ⟶ size qs ≤ size ps)"
by(rule ex_ex1I[OF Longest_common_prefix_ex];
   meson equals0I prefix_length_prefix prefix_order.antisym)

lemma Longest_common_prefix_eq:
 "⟦ L ≠ {};  ∀xs ∈ L. prefix ps xs;
    ∀qs. (∀xs ∈ L. prefix qs xs) ⟶ size qs ≤ size ps ⟧
  ⟹ Longest_common_prefix L = ps"
unfolding Longest_common_prefix_def arg_max_def is_arg_max_linorder
by(rule some1_equality[OF Longest_common_prefix_unique]) auto

lemma Longest_common_prefix_prefix:
  "xs ∈ L ⟹ prefix (Longest_common_prefix L) xs"
unfolding Longest_common_prefix_def arg_max_def is_arg_max_linorder
by(rule someI2_ex[OF Longest_common_prefix_ex]) auto

lemma Longest_common_prefix_longest:
  "L ≠ {} ⟹ ∀xs∈L. prefix ps xs ⟹ length ps ≤ length(Longest_common_prefix L)"
unfolding Longest_common_prefix_def arg_max_def is_arg_max_linorder
by(rule someI2_ex[OF Longest_common_prefix_ex]) auto

lemma Longest_common_prefix_max_prefix:
  "L ≠ {} ⟹ ∀xs∈L. prefix ps xs ⟹ prefix ps (Longest_common_prefix L)"
by(metis Longest_common_prefix_prefix Longest_common_prefix_longest
     prefix_length_prefix ex_in_conv)

lemma Longest_common_prefix_Nil: "[] ∈ L ⟹ Longest_common_prefix L = []"
using Longest_common_prefix_prefix prefix_Nil by blast

lemma Longest_common_prefix_image_Cons: "L ≠ {} ⟹
  Longest_common_prefix ((#) x ‘ L) = x # Longest_common_prefix L"
apply(rule Longest_common_prefix_eq)
  apply(simp)
 apply (simp add: Longest_common_prefix_prefix)
apply simp
by(metis Longest_common_prefix_longest[of L] Cons_prefix_Cons Nitpick.size_list_simp(2)
     Suc_le_mono hd_Cons_tl order.strict_implies_order zero_less_Suc)

lemma Longest_common_prefix_eq_Cons: assumes "L ≠ {}" "[] ∉ L"  "∀xs∈L. hd xs = x"
shows "Longest_common_prefix L = x # Longest_common_prefix {ys. x#ys ∈ L}"
proof -
  have "L = (#) x ‘ {ys. x#ys ∈ L}" using assms(2,3)
    by (auto simp: image_def)(metis hd_Cons_tl)
  thus ?thesis
    by (metis Longest_common_prefix_image_Cons image_is_empty assms(1))
qed

lemma Longest_common_prefix_eq_Nil:
```

```
  "⟦x#ys ∈ L; y#zs ∈ L; x ≠ y ⟧ ⟹ Longest_common_prefix L = []"
by (metis Longest_common_prefix_prefix list.inject prefix_Cons)


fun longest_common_prefix :: "'a list ⇒ 'a list ⇒ 'a list" where
"longest_common_prefix (x#xs) (y#ys) =
  (if x=y then x # longest_common_prefix xs ys else [])" |
"longest_common_prefix _ _ = []"

lemma longest_common_prefix_prefix1:
  "prefix (longest_common_prefix xs ys) xs"
by(induction xs ys rule: longest_common_prefix.induct) auto

lemma longest_common_prefix_prefix2:
  "prefix (longest_common_prefix xs ys) ys"
by(induction xs ys rule: longest_common_prefix.induct) auto

lemma longest_common_prefix_max_prefix:
  "⟦ prefix ps xs; prefix ps ys ⟧
   ⟹ prefix ps (longest_common_prefix xs ys)"
by(induction xs ys arbitrary: ps rule: longest_common_prefix.induct)
  (auto simp: prefix_Cons)
```

9.5 Parallel lists

```
definition parallel :: "'a list ⇒ 'a list ⇒ bool"  (infixl "∥" 50)
  where "(xs ∥ ys) = (¬ prefix xs ys ∧ ¬ prefix ys xs)"

lemma parallelI [intro]: "¬ prefix xs ys ⟹ ¬ prefix ys xs ⟹ xs ∥ ys"
  unfolding parallel_def by blast

lemma parallelE [elim]:
  assumes "xs ∥ ys"
  obtains "¬ prefix xs ys ∧ ¬ prefix ys xs"
  using assms unfolding parallel_def by blast

theorem prefix_cases:
  obtains "prefix xs ys" | "strict_prefix ys xs" | "xs ∥ ys"
  unfolding parallel_def strict_prefix_def by blast

theorem parallel_decomp:
  "xs ∥ ys ⟹ ∃as b bs c cs. b ≠ c ∧ xs = as @ b # bs ∧ ys = as @ c # cs"
proof (induct xs rule: rev_induct)
  case Nil
  then have False by auto
  then show ?case ..
next
  case (snoc x xs)
  show ?case
  proof (rule prefix_cases)
    assume le: "prefix xs ys"
    then obtain ys' where ys: "ys = xs @ ys'" ..
    show ?thesis
    proof (cases ys')
      assume "ys' = []"
      then show ?thesis by (metis append_Nil2 parallelE prefixI snoc.prems ys)
    next
      fix c cs assume ys': "ys' = c # cs"
      have "x ≠ c" using snoc.prems ys ys' by fastforce
      thus "∃as b bs c cs. b ≠ c ∧ xs @ [x] = as @ b # bs ∧ ys = as @ c # cs"
        using ys ys' by blast
```

```
    qed
  next
    assume "strict_prefix ys xs"
    then have "prefix ys (xs @ [x])" by (simp add: strict_prefix_def)
    with snoc have False by blast
    then show ?thesis ..
  next
    assume "xs ∥ ys"
    with snoc obtain as b bs c cs where neq: "(b::'a) ≠ c"
      and xs: "xs = as @ b # bs" and ys: "ys = as @ c # cs"
      by blast
    from xs have "xs @ [x] = as @ b # (bs @ [x])" by simp
    with neq ys show ?thesis by blast
  qed
qed

lemma parallel_append: "a ∥ b ⟹ a @ c ∥ b @ d"
  apply (rule parallelI)
    apply (erule parallelE, erule conjE,
      induct rule: not_prefix_induct, simp+)+
  done

lemma parallel_appendI: "xs ∥ ys ⟹ x = xs @ xs' ⟹ y = ys @ ys' ⟹ x ∥ y"
  by (simp add: parallel_append)

lemma parallel_commute: "a ∥ b ⟷ b ∥ a"
  unfolding parallel_def by auto
```

9.6 Suffix order on lists

```
definition suffix :: "'a list ⇒ 'a list ⇒ bool"
  where "suffix xs ys = (∃zs. ys = zs @ xs)"

definition strict_suffix :: "'a list ⇒ 'a list ⇒ bool"
  where "strict_suffix xs ys ⟷ suffix xs ys ∧ xs ≠ ys"

interpretation suffix_order: order suffix strict_suffix
  by standard (auto simp: suffix_def strict_suffix_def)

interpretation suffix_bot: order_bot Nil suffix strict_suffix
  by standard (simp add: suffix_def)

lemma suffixI [intro?]: "ys = zs @ xs ⟹ suffix xs ys"
  unfolding suffix_def by blast

lemma suffixE [elim?]:
  assumes "suffix xs ys"
  obtains zs where "ys = zs @ xs"
  using assms unfolding suffix_def by blast

lemma suffix_tl [simp]: "suffix (tl xs) xs"
  by (induct xs) (auto simp: suffix_def)

lemma strict_suffix_tl [simp]: "xs ≠ [] ⟹ strict_suffix (tl xs) xs"
  by (induct xs) (auto simp: strict_suffix_def suffix_def)

lemma Nil_suffix [simp]: "suffix [] xs"
  by (simp add: suffix_def)

lemma suffix_Nil [simp]: "(suffix xs []) = (xs = [])"
  by (auto simp add: suffix_def)
```

```
lemma suffix_ConsI: "suffix xs ys ⟹ suffix xs (y # ys)"
  by (auto simp add: suffix_def)

lemma suffix_ConsD: "suffix (x # xs) ys ⟹ suffix xs ys"
  by (auto simp add: suffix_def)

lemma suffix_appendI: "suffix xs ys ⟹ suffix xs (zs @ ys)"
  by (auto simp add: suffix_def)

lemma suffix_appendD: "suffix (zs @ xs) ys ⟹ suffix xs ys"
  by (auto simp add: suffix_def)

lemma strict_suffix_set_subset: "strict_suffix xs ys ⟹ set xs ⊆ set ys"
  by (auto simp: strict_suffix_def suffix_def)

lemma set_mono_suffix: "suffix xs ys ⟹ set xs ⊆ set ys"
by (auto simp: suffix_def)

lemma sorted_antimono_suffix: "suffix xs ys ⟹ sorted ys ⟹ sorted xs"
by (metis sorted_append suffix_def)

lemma suffix_ConsD2: "suffix (x # xs) (y # ys) ⟹ suffix xs ys"
proof -
  assume "suffix (x # xs) (y # ys)"
  then obtain zs where "y # ys = zs @ x # xs" ..
  then show ?thesis
    by (induct zs) (auto intro!: suffix_appendI suffix_ConsI)
qed

lemma suffix_to_prefix [code]: "suffix xs ys ⟷ prefix (rev xs) (rev ys)"
proof
  assume "suffix xs ys"
  then obtain zs where "ys = zs @ xs" ..
  then have "rev ys = rev xs @ rev zs" by simp
  then show "prefix (rev xs) (rev ys)" ..
next
  assume "prefix (rev xs) (rev ys)"
  then obtain zs where "rev ys = rev xs @ zs" ..
  then have "rev (rev ys) = rev zs @ rev (rev xs)" by simp
  then have "ys = rev zs @ xs" by simp
  then show "suffix xs ys" ..
qed

lemma strict_suffix_to_prefix [code]: "strict_suffix xs ys ⟷ strict_prefix (rev xs) (rev ys)"
  by (auto simp: suffix_to_prefix strict_suffix_def strict_prefix_def)

lemma distinct_suffix: "distinct ys ⟹ suffix xs ys ⟹ distinct xs"
  by (clarsimp elim!: suffixE)

lemma map_mono_suffix: "suffix xs ys ⟹ suffix (map f xs) (map f ys)"
by (auto elim!: suffixE intro: suffixI)

lemma filter_mono_suffix: "suffix xs ys ⟹ suffix (filter P xs) (filter P ys)"
by (auto simp: suffix_def)

lemma suffix_drop: "suffix (drop n as) as"
  unfolding suffix_def by (rule exI [where x = "take n as"]) simp

lemma suffix_take: "suffix xs ys ⟹ ys = take (length ys - length xs) ys @ xs"
  by (auto elim!: suffixE)
```

lemma *strict_suffix_reflclp_conv: "strict_suffix$^{==}$ = suffix"*
  by *(intro ext) (auto simp: suffix_def strict_suffix_def)*

lemma *suffix_lists: "suffix xs ys ⟹ ys ∈ lists A ⟹ xs ∈ lists A"*
  unfolding *suffix_def* **by** *auto*

lemma *suffix_snoc [simp]: "suffix xs (ys @ [y]) ⟷ xs = [] ∨ (∃zs. xs = zs @ [y] ∧ suffix zs ys)"*
  by *(cases xs rule: rev_cases) (auto simp: suffix_def)*

lemma *snoc_suffix_snoc [simp]: "suffix (xs @ [x]) (ys @ [y]) = (x = y ∧ suffix xs ys)"*
  by *(auto simp add: suffix_def)*

lemma *same_suffix_suffix [simp]: "suffix (ys @ xs) (zs @ xs) = suffix ys zs"*
  by *(simp add: suffix_to_prefix)*

lemma *same_suffix_nil [simp]: "suffix (ys @ xs) xs = (ys = [])"*
  by *(simp add: suffix_to_prefix)*

theorem *suffix_Cons: "suffix xs (y # ys) ⟷ xs = y # ys ∨ suffix xs ys"*
  unfolding *suffix_def* **by** *(auto simp: Cons_eq_append_conv)*

theorem *suffix_append:*
  "suffix xs (ys @ zs) ⟷ suffix xs zs ∨ (∃xs'. xs = xs' @ zs ∧ suffix xs' ys)"
  by *(auto simp: suffix_def append_eq_append_conv2)*

theorem *suffix_length_le: "suffix xs ys ⟹ length xs ≤ length ys"*
  by *(auto simp add: suffix_def)*

lemma *suffix_same_cases:*
  "suffix (xs_1::'a list) ys ⟹ suffix xs_2 ys ⟹ suffix xs_1 xs_2 ∨ suffix xs_2 xs_1"
  unfolding *suffix_def* **by** *(force simp: append_eq_append_conv2)*

lemma *suffix_length_suffix:*
  "suffix ps xs ⟹ suffix qs xs ⟹ length ps ≤ length qs ⟹ suffix ps qs"
  by *(auto simp: suffix_to_prefix intro: prefix_length_prefix)*

lemma *suffix_length_less: "strict_suffix xs ys ⟹ length xs < length ys"*
  by *(auto simp: strict_suffix_def suffix_def)*

lemma *suffix_ConsD': "suffix (x#xs) ys ⟹ strict_suffix xs ys"*
  by *(auto simp: strict_suffix_def suffix_def)*

lemma *drop_strict_suffix: "strict_suffix xs ys ⟹ strict_suffix (drop n xs) ys"*
proof *(induct n arbitrary: xs ys)*
  case *0*
  then show *?case* **by** *(cases ys) simp_all*
next
  case *(Suc n)*
  then show *?case*
    by *(cases xs) (auto intro: Suc dest: suffix_ConsD' suffix_order.less_imp_le)*
qed

lemma *not_suffix_cases:*
  assumes *pfx: "¬ suffix ps ls"*
  obtains
    (c1) "ps ≠ []" **and** *"ls = []"*
  | *(c2) a as x xs* **where** *"ps = as@[a]"* **and** *"ls = xs@[x]"* **and** *"x = a"* **and** *"¬ suffix as xs"*
  | *(c3) a as x xs* **where** *"ps = as@[a]"* **and** *"ls = xs@[x]"* **and** *"x ≠ a"*
proof *(cases ps rule: rev_cases)*
  case *Nil*

```
  then show ?thesis using pfx by simp
next
  case (snoc as a)
  note c = ‹ps = as@[a]›
  show ?thesis
  proof (cases ls rule: rev_cases)
    case Nil then show ?thesis by (metis append_Nil2 pfx c1 same_suffix_nil)
  next
    case (snoc xs x)
    show ?thesis
    proof (cases "x = a")
      case True
      have "¬ suffix as xs" using pfx c snoc True by simp
      with c snoc True show ?thesis by (rule c2)
    next
      case False
      with c snoc show ?thesis by (rule c3)
    qed
  qed
qed

lemma not_suffix_induct [consumes 1, case_names Nil Neq Eq]:
  assumes np: "¬ suffix ps ls"
    and base: "⋀x xs. P (xs@[x]) []"
    and r1: "⋀x xs y ys. x ≠ y ⟹ P (xs@[x]) (ys@[y])"
    and r2: "⋀x xs y ys. ⟦ x = y; ¬ suffix xs ys; P xs ys ⟧ ⟹ P (xs@[x]) (ys@[y])"
  shows "P ps ls" using np
proof (induct ls arbitrary: ps rule: rev_induct)
  case Nil
  then show ?case by (cases ps rule: rev_cases) (auto intro: base)
next
  case (snoc y ys ps)
  then have npfx: "¬ suffix ps (ys @ [y])" by simp
  then obtain x xs where pv: "ps = xs @ [x]"
    by (rule not_suffix_cases) auto
  show ?case by (metis snoc.hyps snoc_suffix_snoc npfx pv r1 r2)
qed


lemma parallelD1: "x ∥ y ⟹ ¬ prefix x y"
  by blast

lemma parallelD2: "x ∥ y ⟹ ¬ prefix y x"
  by blast

lemma parallel_Nil1 [simp]: "¬ x ∥ []"
  unfolding parallel_def by simp

lemma parallel_Nil2 [simp]: "¬ [] ∥ x"
  unfolding parallel_def by simp

lemma Cons_parallelI1: "a ≠ b ⟹ a # as ∥ b # bs"
  by auto

lemma Cons_parallelI2: "⟦ a = b; as ∥ bs ⟧ ⟹ a # as ∥ b # bs"
  by (metis Cons_prefix_Cons parallelE parallelI)

lemma not_equal_is_parallel:
  assumes neq: "xs ≠ ys"
    and len: "length xs = length ys"
  shows "xs ∥ ys"
```

```
  using len neq
proof (induct rule: list_induct2)
  case Nil
  then show ?case by simp
next
  case (Cons a as b bs)
  have ih: "as ≠ bs ⟹ as ∥ bs" by fact
  show ?case
  proof (cases "a = b")
    case True
    then have "as ≠ bs" using Cons by simp
    then show ?thesis by (rule Cons_parallelI2 [OF True ih])
  next
    case False
    then show ?thesis by (rule Cons_parallelI1)
  qed
qed
```

9.7 Suffixes

```
primrec suffixes where
  "suffixes [] = [[]]"
| "suffixes (x#xs) = suffixes xs @ [x # xs]"

lemma in_set_suffixes [simp]: "xs ∈ set (suffixes ys) ⟷ suffix xs ys"
  by (induction ys) (auto simp: suffix_def Cons_eq_append_conv)

lemma distinct_suffixes [intro]: "distinct (suffixes xs)"
  by (induction xs) (auto simp: suffix_def)

lemma length_suffixes [simp]: "length (suffixes xs) = Suc (length xs)"
  by (induction xs) auto

lemma suffixes_snoc [simp]: "suffixes (xs @ [x]) = [] # map (λys. ys @ [x]) (suffixes xs)"
  by (induction xs) auto

lemma suffixes_not_Nil [simp]: "suffixes xs ≠ []"
  by (cases xs) auto

lemma hd_suffixes [simp]: "hd (suffixes xs) = []"
  by (induction xs) simp_all

lemma last_suffixes [simp]: "last (suffixes xs) = xs"
  by (cases xs) simp_all

lemma suffixes_append:
  "suffixes (xs @ ys) = suffixes ys @ map (λxs'. xs' @ ys) (tl (suffixes xs))"
proof (induction ys rule: rev_induct)
  case Nil
  thus ?case by (cases xs rule: rev_cases) auto
next
  case (snoc y ys)
  show ?case
    by (simp only: append.assoc [symmetric] suffixes_snoc snoc.IH) simp
qed

lemma suffixes_eq_snoc:
  "suffixes ys = xs @ [x] ⟷
     (ys = [] ∧ xs = [] ∨ (∃z zs. ys = z#zs ∧ xs = suffixes zs)) ∧ x = ys"
  by (cases ys) auto
```

```
lemma suffixes_tailrec [code]:
  "suffixes xs = rev (snd (foldl (λ(acc1, acc2) x. (x#acc1, (x#acc1)#acc2)) ([],[[]]) (rev xs)))"
proof -
  have "foldl (λ(acc1, acc2) x. (x#acc1, (x#acc1)#acc2)) (ys, ys # zs) (rev xs) =
        (xs @ ys, rev (map (λas. as @ ys) (suffixes xs)) @ zs)" for ys zs
  proof (induction xs arbitrary: ys zs)
    case (Cons x xs ys zs)
    from Cons.IH[of ys zs]
      show ?case by (simp add: o_def case_prod_unfold)
  qed simp_all
  from this [of "[]" "[]"] show ?thesis by simp
qed

lemma set_suffixes_eq: "set (suffixes xs) = {ys. suffix ys xs}"
  by auto

lemma card_set_suffixes [simp]: "card (set (suffixes xs)) = Suc (length xs)"
  by (subst distinct_card) auto

lemma set_suffixes_append:
  "set (suffixes (xs @ ys)) = set (suffixes ys) ∪ {xs' @ ys |xs'. xs' ∈ set (suffixes xs)}"
  by (subst suffixes_append, cases xs rule: rev_cases) auto


lemma suffixes_conv_prefixes: "suffixes xs = map rev (prefixes (rev xs))"
  by (induction xs) auto

lemma prefixes_conv_suffixes: "prefixes xs = map rev (suffixes (rev xs))"
  by (induction xs) auto

lemma prefixes_rev: "prefixes (rev xs) = map rev (suffixes xs)"
  by (induction xs) auto

lemma suffixes_rev: "suffixes (rev xs) = map rev (prefixes xs)"
  by (induction xs) auto
```

9.8 Homeomorphic embedding on lists

```
inductive list_emb :: "('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool"
  for P :: "('a ⇒ 'a ⇒ bool)"
where
  list_emb_Nil [intro, simp]: "list_emb P [] ys"
| list_emb_Cons [intro] : "list_emb P xs ys ⟹ list_emb P xs (y#ys)"
| list_emb_Cons2 [intro]: "P x y ⟹ list_emb P xs ys ⟹ list_emb P (x#xs) (y#ys)"

lemma list_emb_mono:
  assumes "⋀x y. P x y ⟶ Q x y"
  shows "list_emb P xs ys ⟶ list_emb Q xs ys"
proof
  assume "list_emb P xs ys"
  then show "list_emb Q xs ys" by (induct) (auto simp: assms)
qed

lemma list_emb_Nil2 [simp]:
  assumes "list_emb P xs []" shows "xs = []"
  using assms by (cases rule: list_emb.cases) auto

lemma list_emb_refl:
  assumes "⋀x. x ∈ set xs ⟹ P x x"
  shows "list_emb P xs xs"
  using assms by (induct xs) auto
```

```
lemma list_emb_Cons_Nil [simp]: "list_emb P (x#xs) [] = False"
proof -
  { assume "list_emb P (x#xs) []"
    from list_emb_Nil2 [OF this] have False by simp
  } moreover {
    assume False
    then have "list_emb P (x#xs) []" by simp
  } ultimately show ?thesis by blast
qed

lemma list_emb_append2 [intro]: "list_emb P xs ys ⟹ list_emb P xs (zs @ ys)"
  by (induct zs) auto

lemma list_emb_prefix [intro]:
  assumes "list_emb P xs ys" shows "list_emb P xs (ys @ zs)"
  using assms
  by (induct arbitrary: zs) auto

lemma list_emb_ConsD:
  assumes "list_emb P (x#xs) ys"
  shows "∃us v vs. ys = us @ v # vs ∧ P x v ∧ list_emb P xs vs"
using assms
proof (induct x ≡ "x # xs" ys arbitrary: x xs)
  case list_emb_Cons
  then show ?case by (metis append_Cons)
next
  case (list_emb_Cons2 x y xs ys)
  then show ?case by blast
qed

lemma list_emb_appendD:
  assumes "list_emb P (xs @ ys) zs"
  shows "∃us vs. zs = us @ vs ∧ list_emb P xs us ∧ list_emb P ys vs"
using assms
proof (induction xs arbitrary: ys zs)
  case Nil then show ?case by auto
next
  case (Cons x xs)
  then obtain us v vs where
    zs: "zs = us @ v # vs" and p: "P x v" and lh: "list_emb P (xs @ ys) vs"
    by (auto dest: list_emb_ConsD)
  obtain sk₀ :: "'a list ⇒ 'a list ⇒ 'a list" and sk₁ :: "'a list ⇒ 'a list ⇒ 'a list" where
    sk: "∀x₀ x₁. ¬ list_emb P (xs @ x₀) x₁ ∨ sk₀ x₀ x₁ @ sk₁ x₀ x₁ = x₁ ∧ list_emb P xs (sk₀ x₀ x₁)
∧ list_emb P x₀ (sk₁ x₀ x₁)"
    using Cons(1) by (metis (no_types))
  hence "∀x₂. list_emb P (x # xs) (x₂ @ v # sk₀ ys vs)" using p lh by auto
  thus ?case using lh zs sk by (metis (no_types) append_Cons append_assoc)
qed

lemma list_emb_strict_suffix:
  assumes "list_emb P xs ys" and "strict_suffix ys zs"
  shows "list_emb P xs zs"
  using assms(2) and list_emb_append2 [OF assms(1)] by (auto simp: strict_suffix_def suffix_def)

lemma list_emb_suffix:
  assumes "list_emb P xs ys" and "suffix ys zs"
  shows "list_emb P xs zs"
using assms and list_emb_strict_suffix
unfolding strict_suffix_reflclp_conv[symmetric] by auto
```

```
lemma list_emb_length: "list_emb P xs ys ⟹ length xs ≤ length ys"
  by (induct rule: list_emb.induct) auto

lemma list_emb_trans:
  assumes "⋀x y z. ⟦x ∈ set xs; y ∈ set ys; z ∈ set zs; P x y; P y z⟧ ⟹ P x z"
  shows "⟦list_emb P xs ys; list_emb P ys zs⟧ ⟹ list_emb P xs zs"
proof -
  assume "list_emb P xs ys" and "list_emb P ys zs"
  then show "list_emb P xs zs" using assms
  proof (induction arbitrary: zs)
    case list_emb_Nil show ?case by blast
  next
    case (list_emb_Cons xs ys y)
    from list_emb_ConsD [OF ‹list_emb P (y#ys) zs›] obtain us v vs
      where zs: "zs = us @ v # vs" and "P⁼⁼ y v" and "list_emb P ys vs" by blast
    then have "list_emb P ys (v#vs)" by blast
    then have "list_emb P ys zs" unfolding zs by (rule list_emb_append2)
    from list_emb_Cons.IH [OF this] and list_emb_Cons.prems show ?case by auto
  next
    case (list_emb_Cons2 x y xs ys)
    from list_emb_ConsD [OF ‹list_emb P (y#ys) zs›] obtain us v vs
      where zs: "zs = us @ v # vs" and "P y v" and "list_emb P ys vs" by blast
    with list_emb_Cons2 have "list_emb P xs vs" by auto
    moreover have "P x v"
    proof -
      from zs have "v ∈ set zs" by auto
      moreover have "x ∈ set (x#xs)" and "y ∈ set (y#ys)" by simp_all
      ultimately show ?thesis
        using ‹P x y› and ‹P y v› and list_emb_Cons2
        by blast
    qed
    ultimately have "list_emb P (x#xs) (v#vs)" by blast
    then show ?case unfolding zs by (rule list_emb_append2)
  qed
qed

lemma list_emb_set:
  assumes "list_emb P xs ys" and "x ∈ set xs"
  obtains y where "y ∈ set ys" and "P x y"
  using assms by (induct) auto

lemma list_emb_Cons_iff1 [simp]:
  assumes "P x y"
  shows   "list_emb P (x#xs) (y#ys) ⟷ list_emb P xs ys"
  using assms by (subst list_emb.simps) (auto dest: list_emb_ConsD)

lemma list_emb_Cons_iff2 [simp]:
  assumes "¬P x y"
  shows   "list_emb P (x#xs) (y#ys) ⟷ list_emb P (x#xs) ys"
  using assms by (subst list_emb.simps) auto

lemma list_emb_code [code]:
  "list_emb P [] ys ⟷ True"
  "list_emb P (x#xs) [] ⟷ False"
  "list_emb P (x#xs) (y#ys) ⟷ (if P x y then list_emb P xs ys else list_emb P (x#xs) ys)"
  by simp_all
```

9.9 Subsequences (special case of homeomorphic embedding)

```
abbreviation subseq :: "'a list ⇒ 'a list ⇒ bool"
  where "subseq xs ys ≡ list_emb (=) xs ys"
```

```
definition strict_subseq where "strict_subseq xs ys ⟷ xs ≠ ys ∧ subseq xs ys"

lemma subseq_Cons2: "subseq xs ys ⟹ subseq (x#xs) (x#ys)" by auto

lemma subseq_same_length:
  assumes "subseq xs ys" and "length xs = length ys" shows "xs = ys"
  using assms by (induct) (auto dest: list_emb_length)

lemma not_subseq_length [simp]: "length ys < length xs ⟹ ¬ subseq xs ys"
  by (metis list_emb_length linorder_not_less)

lemma subseq_Cons': "subseq (x#xs) ys ⟹ subseq xs ys"
  by (induct xs, simp, blast dest: list_emb_ConsD)

lemma subseq_Cons2':
  assumes "subseq (x#xs) (x#ys)" shows "subseq xs ys"
  using assms by (cases) (rule subseq_Cons')

lemma subseq_Cons2_neq:
  assumes "subseq (x#xs) (y#ys)"
  shows "x ≠ y ⟹ subseq (x#xs) ys"
  using assms by (cases) auto

lemma subseq_Cons2_iff [simp]:
  "subseq (x#xs) (y#ys) = (if x = y then subseq xs ys else subseq (x#xs) ys)"
  by simp

lemma subseq_append': "subseq (zs @ xs) (zs @ ys) ⟷ subseq xs ys"
  by (induct zs) simp_all

interpretation subseq_order: order subseq strict_subseq
proof
  fix xs ys :: "'a list"
  {
    assume "subseq xs ys" and "subseq ys xs"
    thus "xs = ys"
    proof (induct)
      case list_emb_Nil
      from list_emb_Nil2 [OF this] show ?case by simp
    next
      case list_emb_Cons2
      thus ?case by simp
    next
      case list_emb_Cons
      hence False using subseq_Cons' by fastforce
      thus ?case ..
    qed
  }
  thus "strict_subseq xs ys ⟷ (subseq xs ys ∧ ¬subseq ys xs)"
    by (auto simp: strict_subseq_def)
qed (auto simp: list_emb_refl intro: list_emb_trans)

lemma in_set_subseqs [simp]: "xs ∈ set (subseqs ys) ⟷ subseq xs ys"
proof
  assume "xs ∈ set (subseqs ys)"
  thus "subseq xs ys"
    by (induction ys arbitrary: xs) (auto simp: Let_def)
next
  have [simp]: "[] ∈ set (subseqs ys)" for ys :: "'a list"
    by (induction ys) (auto simp: Let_def)
```

```
  assume "subseq xs ys"
  thus "xs ∈ set (subseqs ys)"
    by (induction xs ys rule: list_emb.induct) (auto simp: Let_def)
qed

lemma set_subseqs_eq: "set (subseqs ys) = {xs. subseq xs ys}"
  by auto

lemma subseq_append_le_same_iff: "subseq (xs @ ys) ys ⟷ xs = []"
  by (auto dest: list_emb_length)

lemma subseq_singleton_left: "subseq [x] ys ⟷ x ∈ set ys"
  by (fastforce dest: list_emb_ConsD split_list_last)

lemma list_emb_append_mono:
  "⟦ list_emb P xs xs'; list_emb P ys ys' ⟧ ⟹ list_emb P (xs@ys) (xs'@ys')"
  by (induct rule: list_emb.induct) auto

lemma prefix_imp_subseq [intro]: "prefix xs ys ⟹ subseq xs ys"
  by (auto simp: prefix_def)

lemma suffix_imp_subseq [intro]: "suffix xs ys ⟹ subseq xs ys"
  by (auto simp: suffix_def)
```

9.10 Appending elements

```
lemma subseq_append [simp]:
  "subseq (xs @ zs) (ys @ zs) ⟷ subseq xs ys" (is "?l = ?r")
proof
  { fix xs' ys' xs ys zs :: "'a list" assume "subseq xs' ys'"
    then have "xs' = xs @ zs ∧ ys' = ys @ zs ⟶ subseq xs ys"
    proof (induct arbitrary: xs ys zs)
      case list_emb_Nil show ?case by simp
    next
      case (list_emb_Cons xs' ys' x)
      { assume "ys=[]" then have ?case using list_emb_Cons(1) by auto }
      moreover
      { fix us assume "ys = x#us"
        then have ?case using list_emb_Cons(2) by(simp add: list_emb.list_emb_Cons) }
      ultimately show ?case by (auto simp:Cons_eq_append_conv)
    next
      case (list_emb_Cons2 x y xs' ys')
      { assume "xs=[]" then have ?case using list_emb_Cons2(1) by auto }
      moreover
      { fix us vs assume "xs=x#us" "ys=x#vs" then have ?case using list_emb_Cons2 by auto}
      moreover
      { fix us assume "xs=x#us" "ys=[]" then have ?case using list_emb_Cons2(2) by bestsimp }
      ultimately show ?case using ‹(=) x y› by (auto simp: Cons_eq_append_conv)
    qed }
  moreover assume ?l
  ultimately show ?r by blast
next
  assume ?r then show ?l by (metis list_emb_append_mono subseq_order.order_refl)
qed

lemma subseq_append_iff:
  "subseq xs (ys @ zs) ⟷ (∃xs1 xs2. xs = xs1 @ xs2 ∧ subseq xs1 ys ∧ subseq xs2 zs)"
  (is "?lhs = ?rhs")
proof
  assume ?lhs thus ?rhs
  proof (induction xs "ys @ zs" arbitrary: ys zs rule: list_emb.induct)
```

```
    case (list_emb_Cons xs ws y ys zs)
    from list_emb_Cons(2)[of "tl ys" zs] and list_emb_Cons(2)[of "[]" "tl zs"] and list_emb_Cons(1,3)
      show ?case by (cases ys) auto
  next
    case (list_emb_Cons2 x y xs ws ys zs)
    from list_emb_Cons2(3)[of "tl ys" zs] and list_emb_Cons2(3)[of "[]" "tl zs"]
       and list_emb_Cons2(1,2,4)
    show ?case by (cases ys) (auto simp: Cons_eq_append_conv)
  qed auto
qed (auto intro: list_emb_append_mono)

lemma subseq_appendE [case_names append]:
  assumes "subseq xs (ys @ zs)"
  obtains xs1 xs2 where "xs = xs1 @ xs2" "subseq xs1 ys" "subseq xs2 zs"
  using assms by (subst (asm) subseq_append_iff) auto

lemma subseq_drop_many: "subseq xs ys ⟹ subseq xs (zs @ ys)"
  by (induct zs) auto

lemma subseq_rev_drop_many: "subseq xs ys ⟹ subseq xs (ys @ zs)"
  by (metis append_Nil2 list_emb_Nil list_emb_append_mono)
```

9.11 Relation to standard list operations

```
lemma subseq_map:
  assumes "subseq xs ys" shows "subseq (map f xs) (map f ys)"
  using assms by (induct) auto

lemma subseq_filter_left [simp]: "subseq (filter P xs) xs"
  by (induct xs) auto

lemma subseq_filter [simp]:
  assumes "subseq xs ys" shows "subseq (filter P xs) (filter P ys)"
  using assms by induct auto

lemma subseq_conv_nths:
  "subseq xs ys ⟷ (∃N. xs = nths ys N)" (is "?L = ?R")
proof
  assume ?L
  then show ?R
  proof (induct)
    case list_emb_Nil show ?case by (metis nths_empty)
  next
    case (list_emb_Cons xs ys x)
    then obtain N where "xs = nths ys N" by blast
    then have "xs = nths (x#ys) (Suc ‘ N)"
      by (clarsimp simp add: nths_Cons inj_image_mem_iff)
    then show ?case by blast
  next
    case (list_emb_Cons2 x y xs ys)
    then obtain N where "xs = nths ys N" by blast
    then have "x#xs = nths (x#ys) (insert 0 (Suc ‘ N))"
      by (clarsimp simp add: nths_Cons inj_image_mem_iff)
    moreover from list_emb_Cons2 have "x = y" by simp
    ultimately show ?case by blast
  qed
next
  assume ?R
  then obtain N where "xs = nths ys N" ..
  moreover have "subseq (nths ys N) ys"
  proof (induct ys arbitrary: N)
```

```
    case Nil show ?case by simp
  next
    case Cons then show ?case by (auto simp: nths_Cons)
  qed
  ultimately show ?L by simp
qed
```

9.12 Contiguous sublists

```
definition sublist :: "'a list ⇒ 'a list ⇒ bool" where
  "sublist xs ys = (∃ps ss. ys = ps @ xs @ ss)"

definition strict_sublist :: "'a list ⇒ 'a list ⇒ bool" where
  "strict_sublist xs ys ⟷ sublist xs ys ∧ xs ≠ ys"

interpretation sublist_order: order sublist strict_sublist
proof
  fix xs ys zs :: "'a list"
  assume "sublist xs ys" "sublist ys zs"
  then obtain xs1 xs2 ys1 ys2 where "ys = xs1 @ xs @ xs2" "zs = ys1 @ ys @ ys2"
    by (auto simp: sublist_def)
  hence "zs = (ys1 @ xs1) @ xs @ (xs2 @ ys2)" by simp
  thus "sublist xs zs" unfolding sublist_def by blast
next
  fix xs ys :: "'a list"
  {
    assume "sublist xs ys" "sublist ys xs"
    then obtain as bs cs ds
      where xs: "xs = as @ ys @ bs" and ys: "ys = cs @ xs @ ds"
      by (auto simp: sublist_def)
    have "xs = as @ cs @ xs @ ds @ bs" by (subst xs, subst ys) auto
    also have "length ... = length as + length cs + length xs + length bs + length ds"
      by simp
    finally have "as = []" "bs = []" by simp_all
    with xs show "xs = ys" by simp
  }
  thus "strict_sublist xs ys ⟷ (sublist xs ys ∧ ¬sublist ys xs)"
    by (auto simp: strict_sublist_def)
qed (auto simp: strict_sublist_def sublist_def intro: exI[of _ "[]"])

lemma sublist_Nil_left [simp, intro]: "sublist [] ys"
  by (auto simp: sublist_def)

lemma sublist_Cons_Nil [simp]: "¬sublist (x#xs) []"
  by (auto simp: sublist_def)

lemma sublist_Nil_right [simp]: "sublist xs [] ⟷ xs = []"
  by (cases xs) auto

lemma sublist_appendI [simp, intro]: "sublist xs (ps @ xs @ ss)"
  by (auto simp: sublist_def)

lemma sublist_append_leftI [simp, intro]: "sublist xs (ps @ xs)"
  by (auto simp: sublist_def intro: exI[of _ "[]"])

lemma sublist_append_rightI [simp, intro]: "sublist xs (xs @ ss)"
  by (auto simp: sublist_def intro: exI[of _ "[]"])

lemma sublist_altdef: "sublist xs ys ⟷ (∃ys'. prefix ys' ys ∧ suffix xs ys')"
proof safe
  assume "sublist xs ys"
```

```
  then obtain ps ss where "ys = ps @ xs @ ss" by (auto simp: sublist_def)
  thus "∃ys'. prefix ys' ys ∧ suffix xs ys'"
    by (intro exI[of _ "ps @ xs"] conjI suffix_appendI) auto
next
  fix ys'
  assume "prefix ys' ys" "suffix xs ys'"
  thus "sublist xs ys" by (auto simp: prefix_def suffix_def)
qed

lemma sublist_altdef': "sublist xs ys ⟷ (∃ys'. suffix ys' ys ∧ prefix xs ys')"
proof safe
  assume "sublist xs ys"
  then obtain ps ss where "ys = ps @ xs @ ss" by (auto simp: sublist_def)
  thus "∃ys'. suffix ys' ys ∧ prefix xs ys'"
    by (intro exI[of _ "xs @ ss"] conjI suffixI) auto
next
  fix ys'
  assume "suffix ys' ys" "prefix xs ys'"
  thus "sublist xs ys" by (auto simp: prefix_def suffix_def)
qed

lemma sublist_Cons_right: "sublist xs (y # ys) ⟷ prefix xs (y # ys) ∨ sublist xs ys"
  by (auto simp: sublist_def prefix_def Cons_eq_append_conv)

lemma sublist_code [code]:
  "sublist [] ys ⟷ True"
  "sublist (x # xs) [] ⟷ False"
  "sublist (x # xs) (y # ys) ⟷ prefix (x # xs) (y # ys) ∨ sublist (x # xs) ys"
  by (simp_all add: sublist_Cons_right)


lemma sublist_append:
  "sublist xs (ys @ zs) ⟷
     sublist xs ys ∨ sublist xs zs ∨ (∃xs1 xs2. xs = xs1 @ xs2 ∧ suffix xs1 ys ∧ prefix xs2 zs)"
  by (auto simp: sublist_altdef prefix_append suffix_append)

primrec sublists :: "'a list ⇒ 'a list list" where
  "sublists [] = [[]]"
| "sublists (x # xs) = sublists xs @ map ((#) x) (prefixes xs)"

lemma in_set_sublists [simp]: "xs ∈ set (sublists ys) ⟷ sublist xs ys"
  by (induction ys arbitrary: xs) (auto simp: sublist_Cons_right prefix_Cons)

lemma set_sublists_eq: "set (sublists xs) = {ys. sublist ys xs}"
  by auto

lemma length_sublists [simp]: "length (sublists xs) = Suc (length xs * Suc (length xs) div 2)"
  by (induction xs) simp_all

lemma sublist_length_le: "sublist xs ys ⟹ length xs ≤ length ys"
  by (auto simp add: sublist_def)

lemma set_mono_sublist: "sublist xs ys ⟹ set xs ⊆ set ys"
  by (auto simp add: sublist_def)

lemma prefix_imp_sublist [simp, intro]: "prefix xs ys ⟹ sublist xs ys"
  by (auto simp: sublist_def prefix_def intro: exI[of _ "[]"])

lemma suffix_imp_sublist [simp, intro]: "suffix xs ys ⟹ sublist xs ys"
  by (auto simp: sublist_def suffix_def intro: exI[of _ "[]"])
```

```
lemma sublist_take [simp, intro]: "sublist (take n xs) xs"
  by (rule prefix_imp_sublist) (simp_all add: take_is_prefix)

lemma sublist_drop [simp, intro]: "sublist (drop n xs) xs"
  by (rule suffix_imp_sublist) (simp_all add: suffix_drop)

lemma sublist_tl [simp, intro]: "sublist (tl xs) xs"
  by (rule suffix_imp_sublist) (simp_all add: suffix_drop)

lemma sublist_butlast [simp, intro]: "sublist (butlast xs) xs"
  by (rule prefix_imp_sublist) (simp_all add: prefixeq_butlast)

lemma sublist_rev [simp]: "sublist (rev xs) (rev ys) = sublist xs ys"
proof
  assume "sublist (rev xs) (rev ys)"
  then obtain as bs where "rev ys = as @ rev xs @ bs"
    by (auto simp: sublist_def)
  also have "rev ... = rev bs @ xs @ rev as" by simp
  finally show "sublist xs ys" by simp
next
  assume "sublist xs ys"
  then obtain as bs where "ys = as @ xs @ bs"
    by (auto simp: sublist_def)
  also have "rev ... = rev bs @ rev xs @ rev as" by simp
  finally show "sublist (rev xs) (rev ys)" by simp
qed

lemma sublist_rev_left: "sublist (rev xs) ys = sublist xs (rev ys)"
  by (subst sublist_rev [symmetric]) (simp only: rev_rev_ident)

lemma sublist_rev_right: "sublist xs (rev ys) = sublist (rev xs) ys"
  by (subst sublist_rev [symmetric]) (simp only: rev_rev_ident)

lemma snoc_sublist_snoc:
  "sublist (xs @ [x]) (ys @ [y]) ⟷
     (x = y ∧ suffix xs ys ∨ sublist (xs @ [x]) ys) "
  by (subst (1 2) sublist_rev [symmetric])
     (simp del: sublist_rev add: sublist_Cons_right suffix_to_prefix)

lemma sublist_snoc:
  "sublist xs (ys @ [y]) ⟷ suffix xs (ys @ [y]) ∨ sublist xs ys"
  by (subst (1 2) sublist_rev [symmetric])
     (simp del: sublist_rev add: sublist_Cons_right suffix_to_prefix)

lemma sublist_imp_subseq [intro]: "sublist xs ys ⟹ subseq xs ys"
  by (auto simp: sublist_def)
```

9.13 Parametricity

```
context includes lifting_syntax
begin

private lemma prefix_primrec:
  "prefix = rec_list (λxs. True) (λx xs xsa ys.
              case ys of [] ⇒ False | y # ys ⇒ x = y ∧ xsa ys)"
proof (intro ext, goal_cases)
  case (1 xs ys)
  show ?case by (induction xs arbitrary: ys) (auto simp: prefix_Cons split: list.splits)
qed

private lemma sublist_primrec:
```

```
  "sublist = (λxs ys. rec_list (λxs. xs = []) (λy ys ysa xs. prefix xs (y # ys) ∨ ysa xs) ys xs)"
proof (intro ext, goal_cases)
  case (1 xs ys)
  show ?case by (induction ys) (auto simp: sublist_Cons_right)
qed

private lemma list_emb_primrec:
  "list_emb = (λuu uua uuaa. rec_list (λP xs. List.null xs) (λy ys ysa P xs. case xs of [] ⇒ True
     | x # xs ⇒ if P x y then ysa P xs else ysa P (x # xs)) uuaa uu uua)"
proof (intro ext, goal_cases)
  case (1 P xs ys)
  show ?case
    by (induction ys arbitrary: xs)
       (auto simp: list_emb_code List.null_def split: list.splits)
qed

lemma prefix_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) prefix prefix"
  unfolding prefix_primrec by transfer_prover

lemma suffix_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) suffix suffix"
  unfolding suffix_to_prefix [abs_def] by transfer_prover

lemma sublist_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) sublist sublist"
  unfolding sublist_primrec by transfer_prover

lemma parallel_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) parallel parallel"
  unfolding parallel_def by transfer_prover



lemma list_emb_transfer [transfer_rule]:
  "((A ===> A ===> (=)) ===> list_all2 A ===> list_all2 A ===> (=)) list_emb list_emb"
  unfolding list_emb_primrec by transfer_prover

lemma strict_prefix_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) strict_prefix strict_prefix"
  unfolding strict_prefix_def by transfer_prover

lemma strict_suffix_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) strict_suffix strict_suffix"
  unfolding strict_suffix_def by transfer_prover

lemma strict_subseq_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) strict_subseq strict_subseq"
  unfolding strict_subseq_def by transfer_prover

lemma strict_sublist_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 A ===> (=)) strict_sublist strict_sublist"
  unfolding strict_sublist_def by transfer_prover
```

```
lemma prefixes_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 (list_all2 A)) prefixes prefixes"
  unfolding prefixes_def by transfer_prover

lemma suffixes_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 (list_all2 A)) suffixes suffixes"
  unfolding suffixes_def by transfer_prover

lemma sublists_transfer [transfer_rule]:
  assumes [transfer_rule]: "bi_unique A"
  shows   "(list_all2 A ===> list_all2 (list_all2 A)) sublists sublists"
  unfolding sublists_def by transfer_prover

end

end
```

10 Infinite Sets and Related Concepts

```
theory Infinite_Set
  imports Main
begin
```

10.1 The set of natural numbers is infinite

```
lemma infinite_nat_iff_unbounded_le: "infinite S ⟷ (∀m. ∃n≥m. n ∈ S)"
  for S :: "nat set"
  using frequently_cofinite[of "λx. x ∈ S"]
  by (simp add: cofinite_eq_sequentially frequently_def eventually_sequentially)

lemma infinite_nat_iff_unbounded: "infinite S ⟷ (∀m. ∃n>m. n ∈ S)"
  for S :: "nat set"
  using frequently_cofinite[of "λx. x ∈ S"]
  by (simp add: cofinite_eq_sequentially frequently_def eventually_at_top_dense)

lemma finite_nat_iff_bounded: "finite S ⟷ (∃k. S ⊆ {..<k})"
  for S :: "nat set"
  using infinite_nat_iff_unbounded_le[of S] by (simp add: subset_eq) (metis not_le)

lemma finite_nat_iff_bounded_le: "finite S ⟷ (∃k. S ⊆ {.. k})"
  for S :: "nat set"
  using infinite_nat_iff_unbounded[of S] by (simp add: subset_eq) (metis not_le)

lemma finite_nat_bounded: "finite S ⟹ ∃k. S ⊆ {..<k}"
  for S :: "nat set"
  by (simp add: finite_nat_iff_bounded)
```

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some `k`, there is some larger number that is an element of the set.

```
lemma unbounded_k_infinite: "∀m>k. ∃n>m. n ∈ S ⟹ infinite (S::nat set)"
  apply (clarsimp simp add: finite_nat_set_iff_bounded)
  apply (drule_tac x="Suc (max m k)" in spec)
  using less_Suc_eq apply fastforce
  done

lemma nat_not_finite: "finite (UNIV::nat set) ⟹ R"
  by simp
```

```
lemma range_inj_infinite:
  fixes f :: "nat ⇒ 'a"
  assumes "inj f"
  shows "infinite (range f)"
proof
  assume "finite (range f)"
  from this assms have "finite (UNIV::nat set)"
    by (rule finite_imageD)
  then show False by simp
qed
```

10.2 The set of integers is also infinite

```
lemma infinite_int_iff_infinite_nat_abs: "infinite S ⟷ infinite ((nat ∘ abs) ` S)"
  for S :: "int set"
proof -
  have "inj_on nat (abs ` A)" for A
    by (rule inj_onI) auto
  then show ?thesis
    by (auto simp flip: image_comp dest: finite_image_absD finite_imageD)
qed

proposition infinite_int_iff_unbounded_le: "infinite S ⟷ (∀m. ∃n. |n| ≥ m ∧ n ∈ S)"
  for S :: "int set"
  by (simp add: infinite_int_iff_infinite_nat_abs infinite_nat_iff_unbounded_le o_def image_def)
    (metis abs_ge_zero nat_le_eq_zle le_nat_iff)

proposition infinite_int_iff_unbounded: "infinite S ⟷ (∀m. ∃n. |n| > m ∧ n ∈ S)"
  for S :: "int set"
  by (simp add: infinite_int_iff_infinite_nat_abs infinite_nat_iff_unbounded o_def image_def)
    (metis (full_types) nat_le_iff nat_mono not_le)

proposition finite_int_iff_bounded: "finite S ⟷ (∃k. abs ` S ⊆ {..<k})"
  for S :: "int set"
  using infinite_int_iff_unbounded_le[of S] by (simp add: subset_eq) (metis not_le)

proposition finite_int_iff_bounded_le: "finite S ⟷ (∃k. abs ` S ⊆ {.. k})"
  for S :: "int set"
  using infinite_int_iff_unbounded[of S] by (simp add: subset_eq) (metis not_le)
```

10.3 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

```
lemma not_INFM [simp]: "¬ (INFM x. P x) ⟷ (MOST x. ¬ P x)"
  by (rule not_frequently)

lemma not_MOST [simp]: "¬ (MOST x. P x) ⟷ (INFM x. ¬ P x)"
  by (rule not_eventually)

lemma INFM_const [simp]: "(INFM x::'a. P) ⟷ P ∧ infinite (UNIV::'a set)"
  by (simp add: frequently_const_iff)

lemma MOST_const [simp]: "(MOST x::'a. P) ⟷ P ∨ finite (UNIV::'a set)"
  by (simp add: eventually_const_iff)

lemma INFM_imp_distrib: "(INFM x. P x ⟶ Q x) ⟷ ((MOST x. P x) ⟶ (INFM x. Q x))"
  by (rule frequently_imp_iff)
```

lemma *MOST_imp_iff: "MOST x. P x ⟹ (MOST x. P x ⟶ Q x) ⟷ (MOST x. Q x)"*
by *(auto intro: eventually_rev_mp eventually_mono)*

lemma *INFM_conjI: "INFM x. P x ⟹ MOST x. Q x ⟹ INFM x. P x ∧ Q x"*
by *(rule frequently_rev_mp[of P]) (auto elim: eventually_mono)*

Properties of quantifiers with injective functions.

lemma *INFM_inj: "INFM x. P (f x) ⟹ inj f ⟹ INFM x. P x"*
using *finite_vimageI[of "{x. P x}" f]* **by** *(auto simp: frequently_cofinite)*

lemma *MOST_inj: "MOST x. P x ⟹ inj f ⟹ MOST x. P (f x)"*
using *finite_vimageI[of "{x. ¬ P x}" f]* **by** *(auto simp: eventually_cofinite)*

Properties of quantifiers with singletons.

lemma *not_INFM_eq [simp]:*
"¬ (INFM x. x = a)"
"¬ (INFM x. a = x)"
unfolding *frequently_cofinite* **by** *simp_all*

lemma *MOST_neq [simp]:*
"MOST x. x ≠ a"
"MOST x. a ≠ x"
unfolding *eventually_cofinite* **by** *simp_all*

lemma *INFM_neq [simp]:*
"(INFM x::'a. x ≠ a) ⟷ infinite (UNIV::'a set)"
"(INFM x::'a. a ≠ x) ⟷ infinite (UNIV::'a set)"
unfolding *frequently_cofinite* **by** *simp_all*

lemma *MOST_eq [simp]:*
"(MOST x::'a. x = a) ⟷ finite (UNIV::'a set)"
"(MOST x::'a. a = x) ⟷ finite (UNIV::'a set)"
unfolding *eventually_cofinite* **by** *simp_all*

lemma *MOST_eq_imp:*
"MOST x. x = a ⟶ P x"
"MOST x. a = x ⟶ P x"
unfolding *eventually_cofinite* **by** *simp_all*

Properties of quantifiers over the naturals.

lemma *MOST_nat: "(∀∞n. P n) ⟷ (∃m. ∀n>m. P n)"*
for *P :: "nat ⇒ bool"*
by *(auto simp add: eventually_cofinite finite_nat_iff_bounded_le subset_eq simp flip: not_le)*

lemma *MOST_nat_le: "(∀∞n. P n) ⟷ (∃m. ∀n≥m. P n)"*
for *P :: "nat ⇒ bool"*
by *(auto simp add: eventually_cofinite finite_nat_iff_bounded subset_eq simp flip: not_le)*

lemma *INFM_nat: "(∃∞n. P n) ⟷ (∀m. ∃n>m. P n)"*
for *P :: "nat ⇒ bool"*
by *(simp add: frequently_cofinite infinite_nat_iff_unbounded)*

lemma *INFM_nat_le: "(∃∞n. P n) ⟷ (∀m. ∃n≥m. P n)"*
for *P :: "nat ⇒ bool"*
by *(simp add: frequently_cofinite infinite_nat_iff_unbounded_le)*

lemma *MOST_INFM: "infinite (UNIV::'a set) ⟹ MOST x::'a. P x ⟹ INFM x::'a. P x"*
by *(simp add: eventually_frequently)*

lemma *MOST_Suc_iff: "(MOST n. P (Suc n)) ⟷ (MOST n. P n)"*

by *(simp add: cofinite_eq_sequentially)*

lemma *MOST_SucI: "MOST n. P n ⟹ MOST n. P (Suc n)"*
and *MOST_SucD: "MOST n. P (Suc n) ⟹ MOST n. P n"*
by *(simp_all add: MOST_Suc_iff)*

lemma *MOST_ge_nat: "MOST n::nat. m ≤ n"*
by *(simp add: cofinite_eq_sequentially)*

— legacy names
lemma *Inf_many_def: "Inf_many P ⟷ infinite {x. P x}"* **by** *(fact frequently_cofinite)*
lemma *Alm_all_def: "Alm_all P ⟷ ¬ (INFM x. ¬ P x)"* **by** *simp*
lemma *INFM_iff_infinite: "(INFM x. P x) ⟷ infinite {x. P x}"* **by** *(fact frequently_cofinite)*
lemma *MOST_iff_cofinite: "(MOST x. P x) ⟷ finite {x. ¬ P x}"* **by** *(fact eventually_cofinite)*
lemma *INFM_EX: "(∃∞x. P x) ⟹ (∃x. P x)"* **by** *(fact frequently_ex)*
lemma *ALL_MOST: "∀x. P x ⟹ ∀∞x. P x"* **by** *(fact always_eventually)*
lemma *INFM_mono: "∃∞x. P x ⟹ (⋀x. P x ⟹ Q x) ⟹ ∃∞x. Q x"* **by** *(fact frequently_elim1)*
lemma *MOST_mono: "∀∞x. P x ⟹ (⋀x. P x ⟹ Q x) ⟹ ∀∞x. Q x"* **by** *(fact eventually_mono)*
lemma *INFM_disj_distrib: "(∃∞x. P x ∨ Q x) ⟷ (∃∞x. P x) ∨ (∃∞x. Q x)"* **by** *(fact frequently_disj_iff)*
lemma *MOST_rev_mp: "∀∞x. P x ⟹ ∀∞x. P x ⟶ Q x ⟹ ∀∞x. Q x"* **by** *(fact eventually_rev_mp)*
lemma *MOST_conj_distrib: "(∀∞x. P x ∧ Q x) ⟷ (∀∞x. P x) ∧ (∀∞x. Q x)"* **by** *(fact eventually_conj_iff)*
lemma *MOST_conjI: "MOST x. P x ⟹ MOST x. Q x ⟹ MOST x. P x ∧ Q x"* **by** *(fact eventually_conj)*
lemma *INFM_finite_Bex_distrib: "finite A ⟹ (INFM y. ∃x∈A. P x y) ⟷ (∃x∈A. INFM y. P x y)"* **by** *(fact frequently_bex_finite_distrib)*
lemma *MOST_finite_Ball_distrib: "finite A ⟹ (MOST y. ∀x∈A. P x y) ⟷ (∀x∈A. MOST y. P x y)"* **by** *(fact eventually_ball_finite_distrib)*
lemma *INFM_E: "INFM x. P x ⟹ (⋀x. P x ⟹ thesis) ⟹ thesis"* **by** *(fact frequentlyE)*
lemma *MOST_I: "(⋀x. P x) ⟹ MOST x. P x"* **by** *(rule eventuallyI)*
lemmas *MOST_iff_finiteNeg = MOST_iff_cofinite*

10.4 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

Could be generalized to *enumerate' S n = (SOME t. t ∈ s ∧ finite {s ∈ S. s < t} ∧ card {s ∈ S. s < t} = n)*.

primrec *(in wellorder) enumerate :: "'a set ⇒ nat ⇒ 'a"*
where
enumerate_0: "enumerate S 0 = (LEAST n. n ∈ S)"
| enumerate_Suc: "enumerate S (Suc n) = enumerate (S - {LEAST n. n ∈ S}) n"

lemma *enumerate_Suc': "enumerate S (Suc n) = enumerate (S - {enumerate S 0}) n"*
by *simp*

lemma *enumerate_in_set: "infinite S ⟹ enumerate S n ∈ S"*
proof *(induct n arbitrary: S)*
case *0*
then show *?case*
by *(fastforce intro: LeastI dest!: infinite_imp_nonempty)*
next
case *(Suc n)*
then show *?case*
by *simp (metis DiffE infinite_remove)*
qed

declare *enumerate_0 [simp del] enumerate_Suc [simp del]*

lemma *enumerate_step: "infinite S ⟹ enumerate S n < enumerate S (Suc n)"*
apply *(induct n arbitrary: S)*
apply *(rule order_le_neq_trans)*
apply *(simp add: enumerate_0 Least_le enumerate_in_set)*

```
   apply (simp only: enumerate_Suc')
   apply (subgoal_tac "enumerate (S - {enumerate S 0}) 0 ∈ S - {enumerate S 0}")
    apply (blast intro: sym)
   apply (simp add: enumerate_in_set del: Diff_iff)
  apply (simp add: enumerate_Suc')
  done

lemma enumerate_mono: "m < n ⟹ infinite S ⟹ enumerate S m < enumerate S n"
  by (induct m n rule: less_Suc_induct) (auto intro: enumerate_step)

lemma le_enumerate:
  assumes S: "infinite S"
  shows "n ≤ enumerate S n"
  using S
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have "n ≤ enumerate S n" by simp
  also note enumerate_mono[of n "Suc n", OF _ ‹infinite S›]
  finally show ?case by simp
qed

lemma enumerate_Suc'':
  fixes S :: "'a::wellorder set"
  assumes "infinite S"
  shows "enumerate S (Suc n) = (LEAST s. s ∈ S ∧ enumerate S n < s)"
  using assms
proof (induct n arbitrary: S)
  case 0
  then have "∀s ∈ S. enumerate S 0 ≤ s"
    by (auto simp: enumerate.simps intro: Least_le)
  then show ?case
    unfolding enumerate_Suc' enumerate_0[of "S - {enumerate S 0}"]
    by (intro arg_cong[where f = Least] ext) auto
next
  case (Suc n S)
  show ?case
    using enumerate_mono[OF zero_less_Suc ‹infinite S›, of n] ‹infinite S›
    apply (subst (1 2) enumerate_Suc')
    apply (subst Suc)
     apply (use ‹infinite S› in simp)
    apply (intro arg_cong[where f = Least] ext)
    apply (auto simp flip: enumerate_Suc')
    done
qed

lemma enumerate_Ex:
  fixes S :: "nat set"
  assumes S: "infinite S"
    and s: "s ∈ S"
  shows "∃n. enumerate S n = s"
  using s
proof (induct s rule: less_induct)
  case (less s)
  show ?case
  proof (cases "∃y∈S. y < s")
    case True
    let ?y = "Max {s'∈S. s' < s}"
    from True have y: "⋀x. ?y < x ⟷ (∀s'∈S. s' < s ⟶ s' < x)"
```

```
    by (subst Max_less_iff) auto
  then have y_in: "?y ∈ {s'∈S. s' < s}"
    by (intro Max_in) auto
  with less.hyps[of ?y] obtain n where "enumerate S n = ?y"
    by auto
  with S have "enumerate S (Suc n) = s"
    by (auto simp: y less enumerate_Suc'' intro!: Least_equality)
  then show ?thesis by auto
next
  case False
  then have "∀t∈S. s ≤ t" by auto
  with ‹s ∈ S› show ?thesis
    by (auto intro!: exI[of _ 0] Least_equality simp: enumerate_0)
qed
qed

lemma bij_enumerate:
  fixes S :: "nat set"
  assumes S: "infinite S"
  shows "bij_betw (enumerate S) UNIV S"
proof -
  have "⋀n m. n ≠ m ⟹ enumerate S n ≠ enumerate S m"
    using enumerate_mono[OF _ ‹infinite S›] by (auto simp: neq_iff)
  then have "inj (enumerate S)"
    by (auto simp: inj_on_def)
  moreover have "∀s ∈ S. ∃i. enumerate S i = s"
    using enumerate_Ex[OF S] by auto
  moreover note ‹infinite S›
  ultimately show ?thesis
    unfolding bij_betw_def by (auto intro: enumerate_in_set)
qed
```

A pair of weird and wonderful lemmas from HOL Light.

```
lemma finite_transitivity_chain:
  assumes "finite A"
    and R: "⋀x. ¬ R x x" "⋀x y z. ⟦R x y; R y z⟧ ⟹ R x z"
    and A: "⋀x. x ∈ A ⟹ ∃y. y ∈ A ∧ R x y"
  shows "A = {}"
  using ‹finite A› A
proof (induct A)
  case empty
  then show ?case by simp
next
  case (insert a A)
  with R show ?case
    by (metis empty_iff insert_iff)
qed

corollary Union_maximal_sets:
  assumes "finite ℱ"
  shows "⋃{T ∈ ℱ. ∀U∈ℱ. ¬ T ⊂ U} = ⋃ℱ"
    (is "?lhs = ?rhs")
proof
  show "?lhs ⊆ ?rhs" by force
  show "?rhs ⊆ ?lhs"
  proof (rule Union_subsetI)
    fix S
    assume "S ∈ ℱ"
    have "{T ∈ ℱ. S ⊆ T} = {}"
      if "¬ (∃y. y ∈ {T ∈ ℱ. ∀U∈ℱ. ¬ T ⊂ U} ∧ S ⊆ y)"
      apply (rule finite_transitivity_chain [of _ "λT U. S ⊆ T ∧ T ⊂ U"])
```

```
      apply (use assms that in auto)
    apply (blast intro: dual_order.trans psubset_imp_subset)
    done
  with ‹S ∈ ℱ› show "∃y. y ∈ {T ∈ ℱ. ∀U∈ℱ. ¬ T ⊂ U} ∧ S ⊆ y"
    by blast
 qed
qed

end
```

11 Countable sets

```
theory Countable_Set
imports Countable Infinite_Set
begin
```

11.1 Predicate for countable sets

```
definition countable :: "'a set ⇒ bool" where
  "countable S ⟷ (∃f::'a ⇒ nat. inj_on f S)"

lemma countableE:
  assumes S: "countable S" obtains f :: "'a ⇒ nat" where "inj_on f S"
  using S by (auto simp: countable_def)

lemma countableI: "inj_on (f::'a ⇒ nat) S ⟹ countable S"
  by (auto simp: countable_def)

lemma countableI': "inj_on (f::'a ⇒ 'b::countable) S ⟹ countable S"
  using comp_inj_on[of f S to_nat] by (auto intro: countableI)

lemma countableE_bij:
  assumes S: "countable S" obtains f :: "nat ⇒ 'a" and C :: "nat set" where "bij_betw f C S"
  using S by (blast elim: countableE dest: inj_on_imp_bij_betw bij_betw_inv)

lemma countableI_bij: "bij_betw f (C::nat set) S ⟹ countable S"
  by (blast intro: countableI bij_betw_inv_into bij_betw_imp_inj_on)

lemma countable_finite: "finite S ⟹ countable S"
  by (blast dest: finite_imp_inj_to_nat_seg countableI)

lemma countableI_bij1: "bij_betw f A B ⟹ countable A ⟹ countable B"
  by (blast elim: countableE_bij intro: bij_betw_trans countableI_bij)

lemma countableI_bij2: "bij_betw f B A ⟹ countable A ⟹ countable B"
  by (blast elim: countableE_bij intro: bij_betw_trans bij_betw_inv_into countableI_bij)

lemma countable_iff_bij[simp]: "bij_betw f A B ⟹ countable A ⟷ countable B"
  by (blast intro: countableI_bij1 countableI_bij2)

lemma countable_subset: "A ⊆ B ⟹ countable B ⟹ countable A"
  by (auto simp: countable_def intro: subset_inj_on)

lemma countableI_type[intro, simp]: "countable (A:: 'a :: countable set)"
  using countableI[of to_nat A] by auto
```

11.2 Enumerate a countable set

```
lemma countableE_infinite:
  assumes "countable S" "infinite S"
```

```
  obtains e :: "'a ⇒ nat" where "bij_betw e S UNIV"
proof -
  obtain f :: "'a ⇒ nat" where "inj_on f S"
    using ⟨countable S⟩ by (rule countableE)
  then have "bij_betw f S (f`S)"
    unfolding bij_betw_def by simp
  moreover
  from ⟨inj_on f S⟩ ⟨infinite S⟩ have inf_fS: "infinite (f`S)"
    by (auto dest: finite_imageD)
  then have "bij_betw (the_inv_into UNIV (enumerate (f`S))) (f`S) UNIV"
    by (intro bij_betw_the_inv_into bij_enumerate)
  ultimately have "bij_betw (the_inv_into UNIV (enumerate (f`S)) ∘ f) S UNIV"
    by (rule bij_betw_trans)
  then show thesis ..
qed

lemma countable_enum_cases:
  assumes "countable S"
  obtains (finite) f :: "'a ⇒ nat" where "finite S" "bij_betw f S {..<card S}"
        | (infinite) f :: "'a ⇒ nat" where "infinite S" "bij_betw f S UNIV"
  using ex_bij_betw_finite_nat[of S] countableE_infinite ⟨countable S⟩
  by (cases "finite S") (auto simp add: atLeast0LessThan)

definition to_nat_on :: "'a set ⇒ 'a ⇒ nat" where
  "to_nat_on S = (SOME f. if finite S then bij_betw f S {..< card S} else bij_betw f S UNIV)"

definition from_nat_into :: "'a set ⇒ nat ⇒ 'a" where
  "from_nat_into S n = (if n ∈ to_nat_on S ` S then inv_into S (to_nat_on S) n else SOME s. s∈S)"

lemma to_nat_on_finite: "finite S ⟹ bij_betw (to_nat_on S) S {..< card S}"
  using ex_bij_betw_finite_nat unfolding to_nat_on_def
  by (intro someI2_ex[where Q="λf. bij_betw f S {..<card S}"]) (auto simp add: atLeast0LessThan)

lemma to_nat_on_infinite: "countable S ⟹ infinite S ⟹ bij_betw (to_nat_on S) S UNIV"
  using countableE_infinite unfolding to_nat_on_def
  by (intro someI2_ex[where Q="λf. bij_betw f S UNIV"]) auto

lemma bij_betw_from_nat_into_finite: "finite S ⟹ bij_betw (from_nat_into S) {..< card S} S"
  unfolding from_nat_into_def[abs_def]
  using to_nat_on_finite[of S]
  apply (subst bij_betw_cong)
  apply (split if_split)
  apply (simp add: bij_betw_def)
  apply (auto cong: bij_betw_cong
              intro: bij_betw_inv_into to_nat_on_finite)
  done

lemma bij_betw_from_nat_into: "countable S ⟹ infinite S ⟹ bij_betw (from_nat_into S) UNIV S"
  unfolding from_nat_into_def[abs_def]
  using to_nat_on_infinite[of S, unfolded bij_betw_def]
  by (auto cong: bij_betw_cong intro: bij_betw_inv_into to_nat_on_infinite)

lemma countable_as_injective_image:
  assumes "countable A" "infinite A"
  obtains f :: "nat ⇒ 'a" where "A = range f" "inj f"
by (metis bij_betw_def bij_betw_from_nat_into [OF assms])

lemma inj_on_to_nat_on[intro]: "countable A ⟹ inj_on (to_nat_on A) A"
  using to_nat_on_infinite[of A] to_nat_on_finite[of A]
  by (cases "finite A") (auto simp: bij_betw_def)
```

lemma *to_nat_on_inj[simp]:*
"countable A ⟹ a ∈ A ⟹ b ∈ A ⟹ to_nat_on A a = to_nat_on A b ⟷ a = b"
using *inj_on_to_nat_on[of A]* **by** *(auto dest: inj_onD)*

lemma *from_nat_into_to_nat_on[simp]: "countable A ⟹ a ∈ A ⟹ from_nat_into A (to_nat_on A a) = a"*
by *(auto simp: from_nat_into_def intro!: inv_into_f_f)*

lemma *subset_range_from_nat_into: "countable A ⟹ A ⊆ range (from_nat_into A)"*
by *(auto intro: from_nat_into_to_nat_on[symmetric])*

lemma *from_nat_into: "A ≠ {} ⟹ from_nat_into A n ∈ A"*
unfolding *from_nat_into_def* **by** *(metis equals0I inv_into_into someI_ex)*

lemma *range_from_nat_into_subset: "A ≠ {} ⟹ range (from_nat_into A) ⊆ A"*
using *from_nat_into[of A]* **by** *auto*

lemma *range_from_nat_into[simp]: "A ≠ {} ⟹ countable A ⟹ range (from_nat_into A) = A"*
by *(metis equalityI range_from_nat_into_subset subset_range_from_nat_into)*

lemma *image_to_nat_on: "countable A ⟹ infinite A ⟹ to_nat_on A ` A = UNIV"*
using *to_nat_on_infinite[of A]* **by** *(simp add: bij_betw_def)*

lemma *to_nat_on_surj: "countable A ⟹ infinite A ⟹ ∃a∈A. to_nat_on A a = n"*
by *(metis (no_types) image_iff iso_tuple_UNIV_I image_to_nat_on)*

lemma *to_nat_on_from_nat_into[simp]: "n ∈ to_nat_on A ` A ⟹ to_nat_on A (from_nat_into A n) = n"*
by *(simp add: f_inv_into_f from_nat_into_def)*

lemma *to_nat_on_from_nat_into_infinite[simp]:*
"countable A ⟹ infinite A ⟹ to_nat_on A (from_nat_into A n) = n"
by *(metis image_iff to_nat_on_surj to_nat_on_from_nat_into)*

lemma *from_nat_into_inj:*
"countable A ⟹ m ∈ to_nat_on A ` A ⟹ n ∈ to_nat_on A ` A ⟹
from_nat_into A m = from_nat_into A n ⟷ m = n"
by *(subst to_nat_on_inj[symmetric, of A]) auto*

lemma *from_nat_into_inj_infinite[simp]:*
"countable A ⟹ infinite A ⟹ from_nat_into A m = from_nat_into A n ⟷ m = n"
using *image_to_nat_on[of A] from_nat_into_inj[of A m n]* **by** *simp*

lemma *eq_from_nat_into_iff:*
"countable A ⟹ x ∈ A ⟹ i ∈ to_nat_on A ` A ⟹ x = from_nat_into A i ⟷ i = to_nat_on A x"
by *auto*

lemma *from_nat_into_surj: "countable A ⟹ a ∈ A ⟹ ∃n. from_nat_into A n = a"*
by *(rule exI[of _ "to_nat_on A a"]) simp*

lemma *from_nat_into_inject[simp]:*
"A ≠ {} ⟹ countable A ⟹ B ≠ {} ⟹ countable B ⟹ from_nat_into A = from_nat_into B ⟷ A = B"
by *(metis range_from_nat_into)*

lemma *inj_on_from_nat_into: "inj_on from_nat_into ({A. A ≠ {} ∧ countable A})"*
unfolding *inj_on_def* **by** *auto*

11.3 Closure properties of countability

lemma *countable_SIGMA[intro, simp]:*
"countable I ⟹ (⋀i. i ∈ I ⟹ countable (A i)) ⟹ countable (SIGMA i : I. A i)"

```
  by (intro countableI'[of "λ(i, a). (to_nat_on I i, to_nat_on (A i) a)"]) (auto simp: inj_on_def)

lemma countable_image[intro, simp]:
  assumes "countable A"
  shows "countable (f`A)"
proof -
  obtain g :: "'a ⇒ nat" where "inj_on g A"
    using assms by (rule countableE)
  moreover have "inj_on (inv_into A f) (f`A)" "inv_into A f ` f ` A ⊆ A"
    by (auto intro: inj_on_inv_into inv_into_into)
  ultimately show ?thesis
    by (blast dest: comp_inj_on subset_inj_on intro: countableI)
qed

lemma countable_image_inj_on: "countable (f ` A) ⟹ inj_on f A ⟹ countable A"
  by (metis countable_image the_inv_into_onto)

lemma countable_UN[intro, simp]:
  fixes I :: "'i set" and A :: "'i => 'a set"
  assumes I: "countable I"
  assumes A: "⋀i. i ∈ I ⟹ countable (A i)"
  shows "countable (⋃i∈I. A i)"
proof -
  have "(⋃i∈I. A i) = snd ` (SIGMA i : I. A i)" by (auto simp: image_iff)
  then show ?thesis by (simp add: assms)
qed

lemma countable_Un[intro]: "countable A ⟹ countable B ⟹ countable (A ∪ B)"
  by (rule countable_UN[of "{True, False}" "λTrue ⇒ A | False ⇒ B", simplified])
     (simp split: bool.split)

lemma countable_Un_iff[simp]: "countable (A ∪ B) ⟷ countable A ∧ countable B"
  by (metis countable_Un countable_subset inf_sup_ord(3,4))

lemma countable_Plus[intro, simp]:
  "countable A ⟹ countable B ⟹ countable (A <+> B)"
  by (simp add: Plus_def)

lemma countable_empty[intro, simp]: "countable {}"
  by (blast intro: countable_finite)

lemma countable_insert[intro, simp]: "countable A ⟹ countable (insert a A)"
  using countable_Un[of "{a}" A] by (auto simp: countable_finite)

lemma countable_Int1[intro, simp]: "countable A ⟹ countable (A ∩ B)"
  by (force intro: countable_subset)

lemma countable_Int2[intro, simp]: "countable B ⟹ countable (A ∩ B)"
  by (blast intro: countable_subset)

lemma countable_INT[intro, simp]: "i ∈ I ⟹ countable (A i) ⟹ countable (⋂i∈I. A i)"
  by (blast intro: countable_subset)

lemma countable_Diff[intro, simp]: "countable A ⟹ countable (A - B)"
  by (blast intro: countable_subset)

lemma countable_insert_eq [simp]: "countable (insert x A) = countable A"
    by auto (metis Diff_insert_absorb countable_Diff insert_absorb)

lemma countable_vimage: "B ⊆ range f ⟹ countable (f -` B) ⟹ countable B"
  by (metis Int_absorb2 countable_image image_vimage_eq)
```

lemma *surj_countable_vimage: "surj f ⟹ countable (f -‘ B) ⟹ countable B"*
by *(metis countable_vimage top_greatest)*

lemma *countable_Collect[simp]: "countable A ⟹ countable {a ∈ A. φ a}"*
by *(metis Collect_conj_eq Int_absorb Int_commute Int_def countable_Int1)*

lemma *countable_Image:*
assumes *"⋀y. y ∈ Y ⟹ countable (X `` {y})"*
assumes *"countable Y"*
shows *"countable (X `` Y)"*
proof -
have *"countable (X `` (⋃y∈Y. {y}))"*
unfolding *Image_UN* **by** *(intro countable_UN assms)*
then show *?thesis* **by** *simp*
qed

lemma *countable_relpow:*
fixes *X :: "'a rel"*
assumes *Image_X: "⋀Y. countable Y ⟹ countable (X `` Y)"*
assumes *Y: "countable Y"*
shows *"countable ((X ^^ i) `` Y)"*
using *Y* **by** *(induct i arbitrary: Y) (auto simp: relcomp_Image Image_X)*

lemma *countable_funpow:*
fixes *f :: "'a set ⇒ 'a set"*
assumes *"⋀A. countable A ⟹ countable (f A)"*
and *"countable A"*
shows *"countable ((f ^^ n) A)"*
by*(induction n)(simp_all add: assms)*

lemma *countable_rtrancl:*
"(⋀Y. countable Y ⟹ countable (X `` Y)) ⟹ countable Y ⟹ countable (X\* `` Y)"
unfolding *rtrancl_is_UN_relpow UN_Image* **by** *(intro countable_UN countableI_type countable_relpow)*

lemma *countable_lists[intro, simp]:*
assumes *A: "countable A"* **shows** *"countable (lists A)"*
proof -
have *"countable (lists (range (from_nat_into A)))"*
by *(auto simp: lists_image)*
with *A* **show** *?thesis*
by *(auto dest: subset_range_from_nat_into countable_subset lists_mono)*
qed

lemma *Collect_finite_eq_lists: "Collect finite = set ‘ lists UNIV"*
using *finite_list* **by** *auto*

lemma *countable_Collect_finite: "countable (Collect (finite::'a::countable set⇒bool))"*
by *(simp add: Collect_finite_eq_lists)*

lemma *countable_int: "countable ℤ"*
unfolding *Ints_def* **by** *auto*

lemma *countable_rat: "countable ℚ"*
unfolding *Rats_def* **by** *auto*

lemma *Collect_finite_subset_eq_lists: "{A. finite A ∧ A ⊆ T} = set ‘ lists T"*
using *finite_list* **by** *(auto simp: lists_eq_set)*

lemma *countable_Collect_finite_subset:*
"countable T ⟹ countable {A. finite A ∧ A ⊆ T}"

```
  unfolding Collect_finite_subset_eq_lists by auto

lemma countable_set_option [simp]: "countable (set_option x)"
by(cases x) auto
```

11.4 Misc lemmas

```
lemma countable_subset_image:
   "countable B ∧ B ⊆ (f ‘ A) ⟷ (∃A’. countable A’ ∧ A’ ⊆ A ∧ (B = f ‘ A’))"
   (is "?lhs = ?rhs")
proof
  assume ?lhs
  show ?rhs
    by (rule exI [where x="inv_into A f ‘ B"])
      (use ‹?lhs› in ‹auto simp: f_inv_into_f subset_iff image_inv_into_cancel inv_into_into›)
next
  assume ?rhs
  then show ?lhs by force
qed

lemma infinite_countable_subset’:
  assumes X: "infinite X" shows "∃C⊆X. countable C ∧ infinite C"
proof -
  from infinite_countable_subset[OF X] guess f ..
  then show ?thesis
    by (intro exI[of _ "range f"]) (auto simp: range_inj_infinite)
qed

lemma countable_all:
  assumes S: "countable S"
  shows "(∀s∈S. P s) ⟷ (∀n::nat. from_nat_into S n ∈ S ⟶ P (from_nat_into S n))"
  using S[THEN subset_range_from_nat_into] by auto

lemma finite_sequence_to_countable_set:
   assumes "countable X" obtains F where "⋀i. F i ⊆ X" "⋀i. F i ⊆ F (Suc i)" "⋀i. finite (F i)"
"(⋃i. F i) = X"
proof - show thesis
    apply (rule that[of "λi. if X = {} then {} else from_nat_into X ‘ {..i}"])
    apply (auto simp: image_iff Ball_def intro: from_nat_into split: if_split_asm)
  proof -
    fix x n assume "x ∈ X" "∀i m. m ≤ i ⟶ x ≠ from_nat_into X m"
    with from_nat_into_surj[OF ‹countable X› ‹x ∈ X›]
    show False
      by auto
  qed
qed

lemma transfer_countable[transfer_rule]:
  "bi_unique R ⟹ rel_fun (rel_set R) (=) countable countable"
  by (rule rel_funI, erule (1) bi_unique_rel_set_lemma)
     (auto dest: countable_image_inj_on)
```

11.5 Uncountable

```
abbreviation uncountable where
  "uncountable A ≡ ¬ countable A"

lemma uncountable_def: "uncountable A ⟷ A ≠ {} ∧ ¬ (∃f::(nat ⇒ ’a). range f = A)"
  by (auto intro: inj_on_inv_into simp: countable_def)
     (metis all_not_in_conv inj_on_iff_surj subset_UNIV)
```

lemma *uncountable_bij_betw: "bij_betw f A B ⟹ uncountable B ⟹ uncountable A"*
unfolding *bij_betw_def* **by** *(metis countable_image)*

lemma *uncountable_infinite: "uncountable A ⟹ infinite A"*
by *(metis countable_finite)*

lemma *uncountable_minus_countable:*
"uncountable A ⟹ countable B ⟹ uncountable (A - B)"
using *countable_Un[of B "A - B"]* **by** *auto*

lemma *countable_Diff_eq [simp]: "countable (A - {x}) = countable A"*
by *(meson countable_Diff countable_empty countable_insert uncountable_minus_countable)*

end

12 Countable Complete Lattices

theory *Countable_Complete_Lattices*
imports *Main Countable_Set*
begin

lemma *UNIV_nat_eq: "UNIV = insert 0 (range Suc)"*
by *(metis UNIV_eq_I nat.nchotomy insertCI rangeI)*

class *countable_complete_lattice = lattice + Inf + Sup + bot + top +*
assumes *ccInf_lower: "countable A ⟹ x ∈ A ⟹ Inf A ≤ x"*
assumes *ccInf_greatest: "countable A ⟹ (⋀x. x ∈ A ⟹ z ≤ x) ⟹ z ≤ Inf A"*
assumes *ccSup_upper: "countable A ⟹ x ∈ A ⟹ x ≤ Sup A"*
assumes *ccSup_least: "countable A ⟹ (⋀x. x ∈ A ⟹ x ≤ z) ⟹ Sup A ≤ z"*
assumes *ccInf_empty [simp]: "Inf {} = top"*
assumes *ccSup_empty [simp]: "Sup {} = bot"*
begin

subclass *bounded_lattice*
proof
fix *a*
show *"bot ≤ a"* **by** *(auto intro: ccSup_least simp only: ccSup_empty [symmetric])*
show *"a ≤ top"* **by** *(auto intro: ccInf_greatest simp only: ccInf_empty [symmetric])*
qed

lemma *ccINF_lower: "countable A ⟹ i ∈ A ⟹ (INF i :A. f i) ≤ f i"*
using *ccInf_lower [of "f ‘ A"]* **by** *simp*

lemma *ccINF_greatest: "countable A ⟹ (⋀i. i ∈ A ⟹ u ≤ f i) ⟹ u ≤ (INF i :A. f i)"*
using *ccInf_greatest [of "f ‘ A"]* **by** *auto*

lemma *ccSUP_upper: "countable A ⟹ i ∈ A ⟹ f i ≤ (SUP i :A. f i)"*
using *ccSup_upper [of "f ‘ A"]* **by** *simp*

lemma *ccSUP_least: "countable A ⟹ (⋀i. i ∈ A ⟹ f i ≤ u) ⟹ (SUP i :A. f i) ≤ u"*
using *ccSup_least [of "f ‘ A"]* **by** *auto*

lemma *ccInf_lower2: "countable A ⟹ u ∈ A ⟹ u ≤ v ⟹ Inf A ≤ v"*
using *ccInf_lower [of A u]* **by** *auto*

lemma *ccINF_lower2: "countable A ⟹ i ∈ A ⟹ f i ≤ u ⟹ (INF i :A. f i) ≤ u"*
using *ccINF_lower [of A i f]* **by** *auto*

lemma *ccSup_upper2: "countable A ⟹ u ∈ A ⟹ v ≤ u ⟹ v ≤ Sup A"*
using *ccSup_upper [of A u]* **by** *auto*

lemma *ccSUP_upper2: "countable A ⟹ i ∈ A ⟹ u ≤ f i ⟹ u ≤ (SUP i :A. f i)"*
using *ccSUP_upper [of A i f]* **by** *auto*

lemma *le_ccInf_iff: "countable A ⟹ b ≤ Inf A ⟷ (∀a∈A. b ≤ a)"*
by *(auto intro: ccInf_greatest dest: ccInf_lower)*

lemma *le_ccINF_iff: "countable A ⟹ u ≤ (INF i :A. f i) ⟷ (∀i∈A. u ≤ f i)"*
using *le_ccInf_iff [of "f ‘ A"]* **by** *simp*

lemma *ccSup_le_iff: "countable A ⟹ Sup A ≤ b ⟷ (∀a∈A. a ≤ b)"*
by *(auto intro: ccSup_least dest: ccSup_upper)*

lemma *ccSUP_le_iff: "countable A ⟹ (SUP i :A. f i) ≤ u ⟷ (∀i∈A. f i ≤ u)"*
using *ccSup_le_iff [of "f ‘ A"]* **by** *simp*

lemma *ccInf_insert [simp]: "countable A ⟹ Inf (insert a A) = inf a (Inf A)"*
by *(force intro: le_infI le_infI1 le_infI2 antisym ccInf_greatest ccInf_lower)*

lemma *ccINF_insert [simp]: "countable A ⟹ (INF x:insert a A. f x) = inf (f a) (INFIMUM A f)"*
unfolding *image_insert* **by** *simp*

lemma *ccSup_insert [simp]: "countable A ⟹ Sup (insert a A) = sup a (Sup A)"*
by *(force intro: le_supI le_supI1 le_supI2 antisym ccSup_least ccSup_upper)*

lemma *ccSUP_insert [simp]: "countable A ⟹ (SUP x:insert a A. f x) = sup (f a) (SUPREMUM A f)"*
unfolding *image_insert* **by** *simp*

lemma *ccINF_empty [simp]: "(INF x:{}. f x) = top"*
unfolding *image_empty* **by** *simp*

lemma *ccSUP_empty [simp]: "(SUP x:{}. f x) = bot"*
unfolding *image_empty* **by** *simp*

lemma *ccInf_superset_mono: "countable A ⟹ B ⊆ A ⟹ Inf A ≤ Inf B"*
by *(auto intro: ccInf_greatest ccInf_lower countable_subset)*

lemma *ccSup_subset_mono: "countable B ⟹ A ⊆ B ⟹ Sup A ≤ Sup B"*
by *(auto intro: ccSup_least ccSup_upper countable_subset)*

lemma *ccInf_mono:*
assumes *[intro]: "countable B" "countable A"*
assumes *"⋀b. b ∈ B ⟹ ∃a∈A. a ≤ b"*
shows *"Inf A ≤ Inf B"*
proof *(rule ccInf_greatest)*
fix *b* **assume** *"b ∈ B"*
with *assms* **obtain** *a* **where** *"a ∈ A"* **and** *"a ≤ b"* **by** *blast*
from ‹a ∈ A› **have** *"Inf A ≤ a"* **by** *(rule ccInf_lower[rotated]) auto*
with ‹a ≤ b› **show** *"Inf A ≤ b"* **by** *auto*
qed *auto*

lemma *ccINF_mono:*
"countable A ⟹ countable B ⟹ (⋀m. m ∈ B ⟹ ∃n∈A. f n ≤ g m) ⟹ (INF n:A. f n) ≤ (INF n:B. g n)"
using *ccInf_mono [of "g ‘ B" "f ‘ A"]* **by** *auto*

lemma *ccSup_mono:*
assumes *[intro]: "countable B" "countable A"*
assumes *"⋀a. a ∈ A ⟹ ∃b∈B. a ≤ b"*
shows *"Sup A ≤ Sup B"*
proof *(rule ccSup_least)*

fix a **assume** "a ∈ A"
with assms **obtain** b **where** "b ∈ B" **and** "a ≤ b" **by** blast
from ‹b ∈ B› **have** "b ≤ Sup B" **by** (rule ccSup_upper[rotated]) auto
with ‹a ≤ b› **show** "a ≤ Sup B" **by** auto
qed auto

lemma ccSUP_mono:
"countable A ⟹ countable B ⟹ (⋀n. n ∈ A ⟹ ∃m∈B. f n ≤ g m) ⟹ (SUP n:A. f n) ≤ (SUP n:B. g n)"
using ccSup_mono [of "g ‘ B" "f ‘ A"] **by** auto

lemma ccINF_superset_mono:
"countable A ⟹ B ⊆ A ⟹ (⋀x. x ∈ B ⟹ f x ≤ g x) ⟹ (INF x:A. f x) ≤ (INF x:B. g x)"
by (blast intro: ccINF_mono countable_subset dest: subsetD)

lemma ccSUP_subset_mono:
"countable B ⟹ A ⊆ B ⟹ (⋀x. x ∈ A ⟹ f x ≤ g x) ⟹ (SUP x:A. f x) ≤ (SUP x:B. g x)"
by (blast intro: ccSUP_mono countable_subset dest: subsetD)

lemma less_eq_ccInf_inter: "countable A ⟹ countable B ⟹ sup (Inf A) (Inf B) ≤ Inf (A ∩ B)"
by (auto intro: ccInf_greatest ccInf_lower)

lemma ccSup_inter_less_eq: "countable A ⟹ countable B ⟹ Sup (A ∩ B) ≤ inf (Sup A) (Sup B)"
by (auto intro: ccSup_least ccSup_upper)

lemma ccInf_union_distrib: "countable A ⟹ countable B ⟹ Inf (A ∪ B) = inf (Inf A) (Inf B)"
by (rule antisym) (auto intro: ccInf_greatest ccInf_lower le_infI1 le_infI2)

lemma ccINF_union:
"countable A ⟹ countable B ⟹ (INF i:A ∪ B. M i) = inf (INF i:A. M i) (INF i:B. M i)"
by (auto intro!: antisym ccINF_mono intro: le_infI1 le_infI2 ccINF_greatest ccINF_lower)

lemma ccSup_union_distrib: "countable A ⟹ countable B ⟹ Sup (A ∪ B) = sup (Sup A) (Sup B)"
by (rule antisym) (auto intro: ccSup_least ccSup_upper le_supI1 le_supI2)

lemma ccSUP_union:
"countable A ⟹ countable B ⟹ (SUP i:A ∪ B. M i) = sup (SUP i:A. M i) (SUP i:B. M i)"
by (auto intro!: antisym ccSUP_mono intro: le_supI1 le_supI2 ccSUP_least ccSUP_upper)

lemma ccINF_inf_distrib: "countable A ⟹ inf (INF a:A. f a) (INF a:A. g a) = (INF a:A. inf (f a) (g a))"
by (rule antisym) (rule ccINF_greatest, auto intro: le_infI1 le_infI2 ccINF_lower ccINF_mono)

lemma ccSUP_sup_distrib: "countable A ⟹ sup (SUP a:A. f a) (SUP a:A. g a) = (SUP a:A. sup (f a) (g a))"
by (rule antisym[rotated]) (rule ccSUP_least, auto intro: le_supI1 le_supI2 ccSUP_upper ccSUP_mono)

lemma ccINF_const [simp]: "A ≠ {} ⟹ (INF i :A. f) = f"
unfolding image_constant_conv **by** auto

lemma ccSUP_const [simp]: "A ≠ {} ⟹ (SUP i :A. f) = f"
unfolding image_constant_conv **by** auto

lemma ccINF_top [simp]: "(INF x:A. top) = top"
by (cases "A = {}") simp_all

lemma ccSUP_bot [simp]: "(SUP x:A. bot) = bot"
by (cases "A = {}") simp_all

lemma ccINF_commute: "countable A ⟹ countable B ⟹ (INF i:A. INF j:B. f i j) = (INF j:B. INF i:A.

```
f i j)"
  by (iprover intro: ccINF_lower ccINF_greatest order_trans antisym)

lemma ccSUP_commute: "countable A ⟹ countable B ⟹ (SUP i:A. SUP j:B. f i j) = (SUP j:B. SUP i:A.
f i j)"
  by (iprover intro: ccSUP_upper ccSUP_least order_trans antisym)

end

context
  fixes a :: "'a::{countable_complete_lattice, linorder}"
begin

lemma less_ccSup_iff: "countable S ⟹ a < Sup S ⟷ (∃x∈S. a < x)"
  unfolding not_le [symmetric] by (subst ccSup_le_iff) auto

lemma less_ccSUP_iff: "countable A ⟹ a < (SUP i:A. f i) ⟷ (∃x∈A. a < f x)"
  using less_ccSup_iff [of "f ‘ A"] by simp

lemma ccInf_less_iff: "countable S ⟹ Inf S < a ⟷ (∃x∈S. x < a)"
  unfolding not_le [symmetric] by (subst le_ccInf_iff) auto

lemma ccINF_less_iff: "countable A ⟹ (INF i:A. f i) < a ⟷ (∃x∈A. f x < a)"
  using ccInf_less_iff [of "f ‘ A"] by simp

end

class countable_complete_distrib_lattice = countable_complete_lattice +
  assumes sup_ccInf: "countable B ⟹ sup a (Inf B) = (INF b:B. sup a b)"
  assumes inf_ccSup: "countable B ⟹ inf a (Sup B) = (SUP b:B. inf a b)"
begin

lemma sup_ccINF:
  "countable B ⟹ sup a (INF b:B. f b) = (INF b:B. sup a (f b))"
  by (simp only: sup_ccInf image_image countable_image)

lemma inf_ccSUP:
  "countable B ⟹ inf a (SUP b:B. f b) = (SUP b:B. inf a (f b))"
  by (simp only: inf_ccSup image_image countable_image)

subclass distrib_lattice
proof
  fix a b c
  from sup_ccInf[of "{b, c}" a] have "sup a (Inf {b, c}) = (INF d:{b, c}. sup a d)"
    by simp
  then show "sup a (inf b c) = inf (sup a b) (sup a c)"
    by simp
qed

lemma ccInf_sup:
  "countable B ⟹ sup (Inf B) a = (INF b:B. sup b a)"
  by (simp add: sup_ccInf sup_commute)

lemma ccSup_inf:
  "countable B ⟹ inf (Sup B) a = (SUP b:B. inf b a)"
  by (simp add: inf_ccSup inf_commute)

lemma ccINF_sup:
  "countable B ⟹ sup (INF b:B. f b) a = (INF b:B. sup (f b) a)"
  by (simp add: sup_ccINF sup_commute)
```

lemma *ccSUP_inf:*
"countable B ⟹ inf (SUP b:B. f b) a = (SUP b:B. inf (f b) a)"
by *(simp add: inf_ccSUP inf_commute)*

lemma *ccINF_sup_distrib2:*
"countable A ⟹ countable B ⟹ sup (INF a:A. f a) (INF b:B. g b) = (INF a:A. INF b:B. sup (f a) (g b))"
by *(subst ccINF_commute) (simp_all add: sup_ccINF ccINF_sup)*

lemma *ccSUP_inf_distrib2:*
"countable A ⟹ countable B ⟹ inf (SUP a:A. f a) (SUP b:B. g b) = (SUP a:A. SUP b:B. inf (f a) (g b))"
by *(subst ccSUP_commute) (simp_all add: inf_ccSUP ccSUP_inf)*

context
fixes *f :: "'a ⇒ 'b::countable_complete_lattice"*
assumes *"mono f"*
begin

lemma *mono_ccInf:*
"countable A ⟹ f (Inf A) ≤ (INF x:A. f x)"
using ‹*mono f*›
by *(auto intro!: countable_complete_lattice_class.ccINF_greatest intro: ccInf_lower dest: monoD)*

lemma *mono_ccSup:*
"countable A ⟹ (SUP x:A. f x) ≤ f (Sup A)"
using ‹*mono f*› **by** *(auto intro: countable_complete_lattice_class.ccSUP_least ccSup_upper dest: monoD)*

lemma *mono_ccINF:*
"countable I ⟹ f (INF i : I. A i) ≤ (INF x : I. f (A x))"
by *(intro countable_complete_lattice_class.ccINF_greatest monoD[OF ‹mono f›] ccINF_lower)*

lemma *mono_ccSUP:*
"countable I ⟹ (SUP x : I. f (A x)) ≤ f (SUP i : I. A i)"
by *(intro countable_complete_lattice_class.ccSUP_least monoD[OF ‹mono f›] ccSUP_upper)*

end

end

12.0.1 Instances of countable complete lattices

instance *"fun" :: (type, countable_complete_lattice) countable_complete_lattice*
by *standard*
(auto simp: le_fun_def intro!: ccSUP_upper ccSUP_least ccINF_lower ccINF_greatest)

subclass *(in complete_lattice) countable_complete_lattice*
by *standard (auto intro: Sup_upper Sup_least Inf_lower Inf_greatest)*

subclass *(in complete_distrib_lattice) countable_complete_distrib_lattice*
by *standard (auto intro: sup_Inf inf_Sup)*

end

13 Continuity and iterations

theory *Order_Continuity*
imports *Complex_Main Countable_Complete_Lattices*
begin

lemma *SUP_nat_binary:*
"(SUP n::nat. if n = 0 then A else B) = (sup A B::'a::countable_complete_lattice)"
apply *(auto intro!: antisym ccSUP_least)*
apply *(rule ccSUP_upper2[***where** *i=0])*
apply *simp_all*
apply *(rule ccSUP_upper2[***where** *i=1])*
apply *simp_all*
done

lemma *INF_nat_binary:*
"(INF n::nat. if n = 0 then A else B) = (inf A B::'a::countable_complete_lattice)"
apply *(auto intro!: antisym ccINF_greatest)*
apply *(rule ccINF_lower2[***where** *i=0])*
apply *simp_all*
apply *(rule ccINF_lower2[***where** *i=1])*
apply *simp_all*
done

The name *continuous* is already taken in *Complex_Main*, so we use *sup_continuous* and *inf_continuous*. These names appear sometimes in literature and have the advantage that these names are duals.

named_theorems *order_continuous_intros*

13.1 Continuity for complete lattices

definition
sup_continuous :: "('a::countable_complete_lattice ⇒ 'b::countable_complete_lattice) ⇒ bool"
where
"sup_continuous F ⟷ (∀M::nat ⇒ 'a. mono M ⟶ F (SUP i. M i) = (SUP i. F (M i)))"

lemma *sup_continuousD: "sup_continuous F ⟹ mono M ⟹ F (SUP i::nat. M i) = (SUP i. F (M i))"*
by *(auto simp: sup_continuous_def)*

lemma *sup_continuous_mono:*
assumes *[simp]: "sup_continuous F"* **shows** *"mono F"*
proof
fix *A B :: "'a"* **assume** *[simp]: "A ≤ B"*
have *"F B = F (SUP n::nat. if n = 0 then A else B)"*
by *(simp add: sup_absorb2 SUP_nat_binary)*
also have *"... = (SUP n::nat. if n = 0 then F A else F B)"*
by *(auto simp: sup_continuousD mono_def intro!: SUP_cong)*
finally show *"F A ≤ F B"*
by *(simp add: SUP_nat_binary le_iff_sup)*
qed

lemma *[order_continuous_intros]:*
shows *sup_continuous_const: "sup_continuous (λx. c)"*
and *sup_continuous_id: "sup_continuous (λx. x)"*
and *sup_continuous_apply: "sup_continuous (λf. f x)"*
and *sup_continuous_fun: "(⋀s. sup_continuous (λx. P x s)) ⟹ sup_continuous P"*
and *sup_continuous_If: "sup_continuous F ⟹ sup_continuous G ⟹ sup_continuous (λf. if C then F f else G f)"*
by *(auto simp: sup_continuous_def)*

lemma *sup_continuous_compose:*
assumes *f: "sup_continuous f"* **and** *g: "sup_continuous g"*
shows *"sup_continuous (λx. f (g x))"*
unfolding *sup_continuous_def*
proof *safe*
fix *M :: "nat ⇒ 'c"*

```
  assume M: "mono M"
  then have "mono (λi. g (M i))"
    using sup_continuous_mono[OF g] by (auto simp: mono_def)
  with M show "f (g (SUPREMUM UNIV M)) = (SUP i. f (g (M i)))"
    by (auto simp: sup_continuous_def g[THEN sup_continuousD] f[THEN sup_continuousD])
qed

lemma sup_continuous_sup[order_continuous_intros]:
  "sup_continuous f ⟹ sup_continuous g ⟹ sup_continuous (λx. sup (f x) (g x))"
  by (simp add: sup_continuous_def ccSUP_sup_distrib)

lemma sup_continuous_inf[order_continuous_intros]:
  fixes P Q :: "'a :: countable_complete_lattice ⇒ 'b :: countable_complete_distrib_lattice"
  assumes P: "sup_continuous P" and Q: "sup_continuous Q"
  shows "sup_continuous (λx. inf (P x) (Q x))"
  unfolding sup_continuous_def
proof (safe intro!: antisym)
  fix M :: "nat ⇒ 'a" assume M: "incseq M"
  have "inf (P (SUP i. M i)) (Q (SUP i. M i)) ≤ (SUP j i. inf (P (M i)) (Q (M j)))"
    by (simp add: sup_continuousD[OF P M] sup_continuousD[OF Q M] inf_ccSUP ccSUP_inf)
  also have "... ≤ (SUP i. inf (P (M i)) (Q (M i)))"
  proof (intro ccSUP_least)
    fix i j from M assms[THEN sup_continuous_mono] show "inf (P (M i)) (Q (M j)) ≤ (SUP i. inf (P (M
i)) (Q (M i)))"
      by (intro ccSUP_upper2[of _ "sup i j"] inf_mono) (auto simp: mono_def)
  qed auto
  finally show "inf (P (SUP i. M i)) (Q (SUP i. M i)) ≤ (SUP i. inf (P (M i)) (Q (M i)))" .

  show "(SUP i. inf (P (M i)) (Q (M i))) ≤ inf (P (SUP i. M i)) (Q (SUP i. M i))"
    unfolding sup_continuousD[OF P M] sup_continuousD[OF Q M] by (intro ccSUP_least inf_mono ccSUP_upper)
auto
qed

lemma sup_continuous_and[order_continuous_intros]:
  "sup_continuous P ⟹ sup_continuous Q ⟹ sup_continuous (λx. P x ∧ Q x)"
  using sup_continuous_inf[of P Q] by simp

lemma sup_continuous_or[order_continuous_intros]:
  "sup_continuous P ⟹ sup_continuous Q ⟹ sup_continuous (λx. P x ∨ Q x)"
  by (auto simp: sup_continuous_def)

lemma sup_continuous_lfp:
  assumes "sup_continuous F" shows "lfp F = (SUP i. (F ^^ i) bot)" (is "lfp F = ?U")
proof (rule antisym)
  note mono = sup_continuous_mono[OF ‹sup_continuous F›]
  show "?U ≤ lfp F"
  proof (rule SUP_least)
    fix i show "(F ^^ i) bot ≤ lfp F"
    proof (induct i)
      case (Suc i)
      have "(F ^^ Suc i) bot = F ((F ^^ i) bot)" by simp
      also have "... ≤ F (lfp F)" by (rule monoD[OF mono Suc])
      also have "... = lfp F" by (simp add: lfp_fixpoint[OF mono])
      finally show ?case .
    qed simp
  qed
  show "lfp F ≤ ?U"
  proof (rule lfp_lowerbound)
    have "mono (λi::nat. (F ^^ i) bot)"
    proof -
      { fix i::nat have "(F ^^ i) bot ≤ (F ^^ (Suc i)) bot"
```

```
      proof (induct i)
        case 0 show ?case by simp
      next
        case Suc thus ?case using monoD[OF mono Suc] by auto
      qed }
    thus ?thesis by (auto simp add: mono_iff_le_Suc)
  qed
  hence "F ?U = (SUP i. (F ^^ Suc i) bot)"
    using ‹sup_continuous F› by (simp add: sup_continuous_def)
  also have "... ≤ ?U"
    by (fast intro: SUP_least SUP_upper)
  finally show "F ?U ≤ ?U" .
 qed
qed

lemma lfp_transfer_bounded:
 assumes P: "P bot" "⋀x. P x ⟹ P (f x)" "⋀M. (⋀i. P (M i)) ⟹ P (SUP i::nat. M i)"
 assumes α: "⋀M. mono M ⟹ (⋀i::nat. P (M i)) ⟹ α (SUP i. M i) = (SUP i. α (M i))"
 assumes f: "sup_continuous f" and g: "sup_continuous g"
 assumes [simp]: "⋀x. P x ⟹ x ≤ lfp f ⟹ α (f x) = g (α x)"
 assumes g_bound: "⋀x. α bot ≤ g x"
 shows "α (lfp f) = lfp g"
proof (rule antisym)
 note mono_g = sup_continuous_mono[OF g]
 note mono_f = sup_continuous_mono[OF f]
 have lfp_bound: "α bot ≤ lfp g"
   by (subst lfp_unfold[OF mono_g]) (rule g_bound)

 have P_pow: "P ((f ^^ i) bot)" for i
   by (induction i) (auto intro!: P)
 have incseq_pow: "mono (λi. (f ^^ i) bot)"
   unfolding mono_iff_le_Suc
 proof
   fix i show "(f ^^ i) bot ≤ (f ^^ (Suc i)) bot"
   proof (induct i)
     case Suc thus ?case using monoD[OF sup_continuous_mono[OF f] Suc] by auto
   qed (simp add: le_fun_def)
 qed
 have P_lfp: "P (lfp f)"
   using P_pow unfolding sup_continuous_lfp[OF f] by (auto intro!: P)

 have iter_le_lfp: "(f ^^ n) bot ≤ lfp f" for n
   apply (induction n)
   apply simp
   apply (subst lfp_unfold[OF mono_f])
   apply (auto intro!: monoD[OF mono_f])
   done

 have "α (lfp f) = (SUP i. α ((f^^i) bot))"
   unfolding sup_continuous_lfp[OF f] using incseq_pow P_pow by (rule α)
 also have "... ≤ lfp g"
 proof (rule SUP_least)
   fix i show "α ((f^^i) bot) ≤ lfp g"
   proof (induction i)
     case (Suc n) then show ?case
       by (subst lfp_unfold[OF mono_g]) (simp add: monoD[OF mono_g] P_pow iter_le_lfp)
   qed (simp add: lfp_bound)
 qed
 finally show "α (lfp f) ≤ lfp g" .

 show "lfp g ≤ α (lfp f)"
```

```
  proof (induction rule: lfp_ordinal_induct[OF mono_g])
    case (1 S) then show ?case
      by (subst lfp_unfold[OF sup_continuous_mono[OF f]])
         (simp add: monoD[OF mono_g] P_lfp)
  qed (auto intro: Sup_least)
qed

lemma lfp_transfer:
  "sup_continuous α ⟹ sup_continuous f ⟹ sup_continuous g ⟹
    (⋀x. α bot ≤ g x) ⟹ (⋀x. x ≤ lfp f ⟹ α (f x) = g (α x)) ⟹ α (lfp f) = lfp g"
  by (rule lfp_transfer_bounded[where P=top]) (auto dest: sup_continuousD)

definition
  inf_continuous :: "('a::countable_complete_lattice ⇒ 'b::countable_complete_lattice) ⇒ bool"
where
  "inf_continuous F ⟷ (∀M::nat ⇒ 'a. antimono M ⟶ F (INF i. M i) = (INF i. F (M i)))"

lemma inf_continuousD: "inf_continuous F ⟹ antimono M ⟹ F (INF i::nat. M i) = (INF i. F (M i))"
  by (auto simp: inf_continuous_def)

lemma inf_continuous_mono:
  assumes [simp]: "inf_continuous F" shows "mono F"
proof
  fix A B :: "'a" assume [simp]: "A ≤ B"
  have "F A = F (INF n::nat. if n = 0 then B else A)"
    by (simp add: inf_absorb2 INF_nat_binary)
  also have "... = (INF n::nat. if n = 0 then F B else F A)"
    by (auto simp: inf_continuousD antimono_def intro!: INF_cong)
  finally show "F A ≤ F B"
    by (simp add: INF_nat_binary le_iff_inf inf_commute)
qed

lemma [order_continuous_intros]:
  shows inf_continuous_const: "inf_continuous (λx. c)"
    and inf_continuous_id: "inf_continuous (λx. x)"
    and inf_continuous_apply: "inf_continuous (λf. f x)"
    and inf_continuous_fun: "(⋀s. inf_continuous (λx. P x s)) ⟹ inf_continuous P"
    and inf_continuous_If: "inf_continuous F ⟹ inf_continuous G ⟹ inf_continuous (λf. if C then
F f else G f)"
  by (auto simp: inf_continuous_def)

lemma inf_continuous_inf[order_continuous_intros]:
  "inf_continuous f ⟹ inf_continuous g ⟹ inf_continuous (λx. inf (f x) (g x))"
  by (simp add: inf_continuous_def ccINF_inf_distrib)

lemma inf_continuous_sup[order_continuous_intros]:
  fixes P Q :: "'a :: countable_complete_lattice ⇒ 'b :: countable_complete_distrib_lattice"
  assumes P: "inf_continuous P" and Q: "inf_continuous Q"
  shows "inf_continuous (λx. sup (P x) (Q x))"
  unfolding inf_continuous_def
proof (safe intro!: antisym)
  fix M :: "nat ⇒ 'a" assume M: "decseq M"
  show "sup (P (INF i. M i)) (Q (INF i. M i)) ≤ (INF i. sup (P (M i)) (Q (M i)))"
    unfolding inf_continuousD[OF P M] inf_continuousD[OF Q M] by (intro ccINF_greatest sup_mono ccINF_lower)
auto

  have "(INF i. sup (P (M i)) (Q (M i))) ≤ (INF j i. sup (P (M i)) (Q (M j)))"
  proof (intro ccINF_greatest)
    fix i j from M assms[THEN inf_continuous_mono] show "sup (P (M i)) (Q (M j)) ≥ (INF i. sup (P (M
i)) (Q (M i)))"
      by (intro ccINF_lower2[of _ "sup i j"] sup_mono) (auto simp: mono_def antimono_def)
```

```
  qed auto
  also have "... ≤ sup (P (INF i. M i)) (Q (INF i. M i))"
    by (simp add: inf_continuousD[OF P M] inf_continuousD[OF Q M] ccINF_sup sup_ccINF)
  finally show "sup (P (INF i. M i)) (Q (INF i. M i)) ≥ (INF i. sup (P (M i)) (Q (M i)))" .
qed

lemma inf_continuous_and[order_continuous_intros]:
  "inf_continuous P ⟹ inf_continuous Q ⟹ inf_continuous (λx. P x ∧ Q x)"
  using inf_continuous_inf[of P Q] by simp

lemma inf_continuous_or[order_continuous_intros]:
  "inf_continuous P ⟹ inf_continuous Q ⟹ inf_continuous (λx. P x ∨ Q x)"
  using inf_continuous_sup[of P Q] by simp

lemma inf_continuous_compose:
  assumes f: "inf_continuous f" and g: "inf_continuous g"
  shows "inf_continuous (λx. f (g x))"
  unfolding inf_continuous_def
proof safe
  fix M :: "nat ⇒ 'c"
  assume M: "antimono M"
  then have "antimono (λi. g (M i))"
    using inf_continuous_mono[OF g] by (auto simp: mono_def antimono_def)
  with M show "f (g (INFIMUM UNIV M)) = (INF i. f (g (M i)))"
    by (auto simp: inf_continuous_def g[THEN inf_continuousD] f[THEN inf_continuousD])
qed

lemma inf_continuous_gfp:
  assumes "inf_continuous F" shows "gfp F = (INF i. (F ^^ i) top)" (is "gfp F = ?U")
proof (rule antisym)
  note mono = inf_continuous_mono[OF ‹inf_continuous F›]
  show "gfp F ≤ ?U"
  proof (rule INF_greatest)
    fix i show "gfp F ≤ (F ^^ i) top"
    proof (induct i)
      case (Suc i)
      have "gfp F = F (gfp F)" by (simp add: gfp_fixpoint[OF mono])
      also have "... ≤ F ((F ^^ i) top)" by (rule monoD[OF mono Suc])
      also have "... = (F ^^ Suc i) top" by simp
      finally show ?case .
    qed simp
  qed
  show "?U ≤ gfp F"
  proof (rule gfp_upperbound)
    have *: "antimono (λi::nat. (F ^^ i) top)"
    proof -
      { fix i::nat have "(F ^^ Suc i) top ≤ (F ^^ i) top"
        proof (induct i)
          case 0 show ?case by simp
        next
          case Suc thus ?case using monoD[OF mono Suc] by auto
        qed }
      thus ?thesis by (auto simp add: antimono_iff_le_Suc)
    qed
    have "?U ≤ (INF i. (F ^^ Suc i) top)"
      by (fast intro: INF_greatest INF_lower)
    also have "... ≤ F ?U"
      by (simp add: inf_continuousD ‹inf_continuous F› *)
    finally show "?U ≤ F ?U" .
  qed
qed
```

```
lemma gfp_transfer:
  assumes α: "inf_continuous α" and f: "inf_continuous f" and g: "inf_continuous g"
  assumes [simp]: "α top = top" "⋀x. α (f x) = g (α x)"
  shows "α (gfp f) = gfp g"
proof -
  have "α (gfp f) = (INF i. α ((f^^i) top))"
    unfolding inf_continuous_gfp[OF f] by (intro f α inf_continuousD antimono_funpow inf_continuous_mono)
  moreover have "α ((f^^i) top) = (g^^i) top" for i
    by (induction i; simp)
  ultimately show ?thesis
    unfolding inf_continuous_gfp[OF g] by simp
qed

lemma gfp_transfer_bounded:
  assumes P: "P (f top)" "⋀x. P x ⟹ P (f x)" "⋀M. antimono M ⟹ (⋀i. P (M i)) ⟹ P (INF i::nat.
M i)"
  assumes α: "⋀M. antimono M ⟹ (⋀i::nat. P (M i)) ⟹ α (INF i. M i) = (INF i. α (M i))"
  assumes f: "inf_continuous f" and g: "inf_continuous g"
  assumes [simp]: "⋀x. P x ⟹ α (f x) = g (α x)"
  assumes g_bound: "⋀x. g x ≤ α (f top)"
  shows "α (gfp f) = gfp g"
proof (rule antisym)
  note mono_g = inf_continuous_mono[OF g]

  have P_pow: "P ((f ^^ i) (f top))" for i
    by (induction i) (auto intro!: P)

  have antimono_pow: "antimono (λi. (f ^^ i) top)"
    unfolding antimono_iff_le_Suc
  proof
    fix i show "(f ^^ Suc i) top ≤ (f ^^ i) top"
    proof (induct i)
      case Suc thus ?case using monoD[OF inf_continuous_mono[OF f] Suc] by auto
    qed (simp add: le_fun_def)
  qed
  have antimono_pow2: "antimono (λi. (f ^^ i) (f top))"
  proof
    show "x ≤ y ⟹ (f ^^ y) (f top) ≤ (f ^^ x) (f top)" for x y
      using antimono_pow[THEN antimonoD, of "Suc x" "Suc y"]
      unfolding funpow_Suc_right by simp
  qed

  have gfp_f: "gfp f = (INF i. (f ^^ i) (f top))"
    unfolding inf_continuous_gfp[OF f]
  proof (rule INF_eq)
    show "∃j∈UNIV. (f ^^ j) (f top) ≤ (f ^^ i) top" for i
      by (intro bexI[of _ "i - 1"]) (auto simp: diff_Suc funpow_Suc_right simp del: funpow.simps(2)
split: nat.split)
    show "∃j∈UNIV. (f ^^ j) top ≤ (f ^^ i) (f top)" for i
      by (intro bexI[of _ "Suc i"]) (auto simp: funpow_Suc_right simp del: funpow.simps(2))
  qed

  have P_lfp: "P (gfp f)"
    unfolding gfp_f by (auto intro!: P P_pow antimono_pow2)

  have "α (gfp f) = (INF i. α ((f^^i) (f top)))"
    unfolding gfp_f by (rule α) (auto intro!: P_pow antimono_pow2)
  also have "... ≥ gfp g"
  proof (rule INF_greatest)
    fix i show "gfp g ≤ α ((f^^i) (f top))"
```

```
    proof (induction i)
      case (Suc n) then show ?case
        by (subst gfp_unfold[OF mono_g]) (simp add: monoD[OF mono_g] P_pow)
    next
      case 0
      have "gfp g ≤ α (f top)"
        by (subst gfp_unfold[OF mono_g]) (rule g_bound)
      then show ?case
        by simp
    qed
  qed
  finally show "gfp g ≤ α (gfp f)" .

  show "α (gfp f) ≤ gfp g"
  proof (induction rule: gfp_ordinal_induct[OF mono_g])
    case (1 S) then show ?case
      by (subst gfp_unfold[OF inf_continuous_mono[OF f]])
         (simp add: monoD[OF mono_g] P_lfp)
  qed (auto intro: Inf_greatest)
qed
```

13.1.1 Least fixed points in countable complete lattices

```
definition (in countable_complete_lattice) cclfp :: "('a ⇒ 'a) ⇒ 'a"
  where "cclfp f = (SUP i. (f ^^ i) bot)"

lemma cclfp_unfold:
  assumes "sup_continuous F" shows "cclfp F = F (cclfp F)"
proof -
  have "cclfp F = (SUP i. F ((F ^^ i) bot))"
    unfolding cclfp_def by (subst UNIV_nat_eq) auto
  also have "... = F (cclfp F)"
    unfolding cclfp_def
    by (intro sup_continuousD[symmetric] assms mono_funpow sup_continuous_mono)
  finally show ?thesis .
qed

lemma cclfp_lowerbound: assumes f: "mono f" and A: "f A ≤ A" shows "cclfp f ≤ A"
  unfolding cclfp_def
proof (intro ccSUP_least)
  fix i show "(f ^^ i) bot ≤ A"
  proof (induction i)
    case (Suc i) from monoD[OF f this] A show ?case
      by auto
  qed simp
qed simp

lemma cclfp_transfer:
  assumes "sup_continuous α" "mono f"
  assumes "α bot = bot" "⋀x. α (f x) = g (α x)"
  shows "α (cclfp f) = cclfp g"
proof -
  have "α (cclfp f) = (SUP i. α ((f ^^ i) bot))"
    unfolding cclfp_def by (intro sup_continuousD assms mono_funpow sup_continuous_mono)
  moreover have "α ((f ^^ i) bot) = (g ^^ i) bot" for i
    by (induction i) (simp_all add: assms)
  ultimately show ?thesis
    by (simp add: cclfp_def)
qed

end
```

14 Extended natural numbers (i.e. with infinity)

```
theory Extended_Nat
imports Main Countable Order_Continuity
begin

class infinity =
  fixes infinity :: "'a"  ("∞")

context
  fixes f :: "nat ⇒ 'a::{canonically_ordered_monoid_add, linorder_topology, complete_linorder}"
begin

lemma sums_SUP[simp, intro]: "f sums (SUP n. ∑i<n. f i)"
  unfolding sums_def by (intro LIMSEQ_SUP monoI sum_mono2 zero_le) auto

lemma suminf_eq_SUP: "suminf f = (SUP n. ∑i<n. f i)"
  using sums_SUP by (rule sums_unique[symmetric])

end
```

14.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

```
typedef enat = "UNIV :: nat option set" ..
```

TODO: introduce enat as coinductive datatype, enat is just `of_nat`

```
definition enat :: "nat ⇒ enat" where
  "enat n = Abs_enat (Some n)"

instantiation enat :: infinity
begin

definition "∞ = Abs_enat None"
instance ..

end

instance enat :: countable
proof
  show "∃to_nat::enat ⇒ nat. inj to_nat"
    by (rule exI[of _ "to_nat ∘ Rep_enat"]) (simp add: inj_on_def Rep_enat_inject)
qed

old_rep_datatype enat "∞ :: enat"
proof -
  fix P i assume "⋀j. P (enat j)" "P ∞"
  then show "P i"
  proof induct
    case (Abs_enat y) then show ?case
      by (cases y rule: option.exhaust)
         (auto simp: enat_def infinity_enat_def)
  qed
qed (auto simp add: enat_def infinity_enat_def Abs_enat_inject)

declare [[coercion "enat::nat⇒enat"]]

lemmas enat2_cases = enat.exhaust[case_product enat.exhaust]
lemmas enat3_cases = enat.exhaust[case_product enat.exhaust enat.exhaust]
```

lemma *not_infinity_eq [iff]: "(x ≠ ∞) = (∃i. x = enat i)"*
by *(cases x) auto*

lemma *not_enat_eq [iff]: "(∀y. x ≠ enat y) = (x = ∞)"*
by *(cases x) auto*

lemma *enat_ex_split: "(∃c::enat. P c) ⟷ P ∞ ∨ (∃c::nat. P c)"*
by *(metis enat.exhaust)*

primrec *the_enat :: "enat ⇒ nat"*
where *"the_enat (enat n) = n"*

14.2 Constructors and numbers

instantiation *enat :: zero_neq_one*
begin

definition
"0 = enat 0"

definition
"1 = enat 1"

instance
proof qed *(simp add: zero_enat_def one_enat_def)*

end

definition *eSuc :: "enat ⇒ enat"* **where**
"eSuc i = (case i of enat n ⇒ enat (Suc n) | ∞ ⇒ ∞)"

lemma *enat_0 [code_post]: "enat 0 = 0"*
by *(simp add: zero_enat_def)*

lemma *enat_1 [code_post]: "enat 1 = 1"*
by *(simp add: one_enat_def)*

lemma *enat_0_iff: "enat x = 0 ⟷ x = 0" "0 = enat x ⟷ x = 0"*
by *(auto simp add: zero_enat_def)*

lemma *enat_1_iff: "enat x = 1 ⟷ x = 1" "1 = enat x ⟷ x = 1"*
by *(auto simp add: one_enat_def)*

lemma *one_eSuc: "1 = eSuc 0"*
by *(simp add: zero_enat_def one_enat_def eSuc_def)*

lemma *infinity_ne_i0 [simp]: "(∞::enat) ≠ 0"*
by *(simp add: zero_enat_def)*

lemma *i0_ne_infinity [simp]: "0 ≠ (∞::enat)"*
by *(simp add: zero_enat_def)*

lemma *zero_one_enat_neq:*
"¬ 0 = (1::enat)"
"¬ 1 = (0::enat)"
unfolding *zero_enat_def one_enat_def* **by** *simp_all*

lemma *infinity_ne_i1 [simp]: "(∞::enat) ≠ 1"*
by *(simp add: one_enat_def)*

lemma *i1_ne_infinity [simp]: "1 ≠ (∞::enat)"*

by *(simp add: one_enat_def)*

lemma *eSuc_enat: "eSuc (enat n) = enat (Suc n)"*
by *(simp add: eSuc_def)*

lemma *eSuc_infinity [simp]: "eSuc ∞ = ∞"*
by *(simp add: eSuc_def)*

lemma *eSuc_ne_0 [simp]: "eSuc n ≠ 0"*
by *(simp add: eSuc_def zero_enat_def split: enat.splits)*

lemma *zero_ne_eSuc [simp]: "0 ≠ eSuc n"*
by *(rule eSuc_ne_0 [symmetric])*

lemma *eSuc_inject [simp]: "eSuc m = eSuc n ⟷ m = n"*
by *(simp add: eSuc_def split: enat.splits)*

lemma *eSuc_enat_iff: "eSuc x = enat y ⟷ (∃n. y = Suc n ∧ x = enat n)"*
by *(cases y) (auto simp: enat_0 eSuc_enat[symmetric])*

lemma *enat_eSuc_iff: "enat y = eSuc x ⟷ (∃n. y = Suc n ∧ enat n = x)"*
by *(cases y) (auto simp: enat_0 eSuc_enat[symmetric])*

14.3 Addition

instantiation *enat :: comm_monoid_add*
begin

definition *[nitpick_simp]:*
"m + n = (case m of ∞ ⇒ ∞ | enat m ⇒ (case n of ∞ ⇒ ∞ | enat n ⇒ enat (m + n)))"

lemma *plus_enat_simps [simp, code]:*
fixes *q :: enat*
shows *"enat m + enat n = enat (m + n)"*
and *"∞ + q = ∞"*
and *"q + ∞ = ∞"*
by *(simp_all add: plus_enat_def split: enat.splits)*

instance
proof
fix *n m q :: enat*
show *"n + m + q = n + (m + q)"*
by *(cases n m q rule: enat3_cases) auto*
show *"n + m = m + n"*
by *(cases n m rule: enat2_cases) auto*
show *"0 + n = n"*
by *(cases n) (simp_all add: zero_enat_def)*
qed

end

lemma *eSuc_plus_1:*
"eSuc n = n + 1"
by *(cases n) (simp_all add: eSuc_enat one_enat_def)*

lemma *plus_1_eSuc:*
"1 + q = eSuc q"
"q + 1 = eSuc q"
by *(simp_all add: eSuc_plus_1 ac_simps)*

lemma *iadd_Suc: "eSuc m + n = eSuc (m + n)"*

```
  by (simp_all add: eSuc_plus_1 ac_simps)

lemma iadd_Suc_right: "m + eSuc n = eSuc (m + n)"
  by (simp only: add.commute[of m] iadd_Suc)
```

14.4 Multiplication

```
instantiation enat :: "{comm_semiring_1, semiring_no_zero_divisors}"
begin

definition times_enat_def [nitpick_simp]:
  "m * n = (case m of ∞ ⇒ if n = 0 then 0 else ∞ | enat m ⇒
    (case n of ∞ ⇒ if m = 0 then 0 else ∞ | enat n ⇒ enat (m * n)))"

lemma times_enat_simps [simp, code]:
  "enat m * enat n = enat (m * n)"
  "∞ * ∞ = (∞::enat)"
  "∞ * enat n = (if n = 0 then 0 else ∞)"
  "enat m * ∞ = (if m = 0 then 0 else ∞)"
  unfolding times_enat_def zero_enat_def
  by (simp_all split: enat.split)

instance
proof
  fix a b c :: enat
  show "(a * b) * c = a * (b * c)"
    unfolding times_enat_def zero_enat_def
    by (simp split: enat.split)
  show comm: "a * b = b * a"
    unfolding times_enat_def zero_enat_def
    by (simp split: enat.split)
  show "1 * a = a"
    unfolding times_enat_def zero_enat_def one_enat_def
    by (simp split: enat.split)
  show distr: "(a + b) * c = a * c + b * c"
    unfolding times_enat_def zero_enat_def
    by (simp split: enat.split add: distrib_right)
  show "0 * a = 0"
    unfolding times_enat_def zero_enat_def
    by (simp split: enat.split)
  show "a * 0 = 0"
    unfolding times_enat_def zero_enat_def
    by (simp split: enat.split)
  show "a * (b + c) = a * b + a * c"
    by (cases a b c rule: enat3_cases) (auto simp: times_enat_def zero_enat_def distrib_left)
  show "a ≠ 0 ⟹ b ≠ 0 ⟹ a * b ≠ 0"
    by (cases a b rule: enat2_cases) (auto simp: times_enat_def zero_enat_def)
qed

end

lemma mult_eSuc: "eSuc m * n = n + m * n"
  unfolding eSuc_plus_1 by (simp add: algebra_simps)

lemma mult_eSuc_right: "m * eSuc n = m + m * n"
  unfolding eSuc_plus_1 by (simp add: algebra_simps)

lemma of_nat_eq_enat: "of_nat n = enat n"
  apply (induct n)
  apply (simp add: enat_0)
  apply (simp add: plus_1_eSuc eSuc_enat)
```

```
  done

instance enat :: semiring_char_0
proof
  have "inj enat" by (rule injI) simp
  then show "inj (λn. of_nat n :: enat)" by (simp add: of_nat_eq_enat)
qed

lemma imult_is_infinity: "((a::enat) * b = ∞) = (a = ∞ ∧ b ≠ 0 ∨ b = ∞ ∧ a ≠ 0)"
  by (auto simp add: times_enat_def zero_enat_def split: enat.split)
```

14.5 Numerals

```
lemma numeral_eq_enat:
  "numeral k = enat (numeral k)"
  using of_nat_eq_enat [of "numeral k"] by simp

lemma enat_numeral [code_abbrev]:
  "enat (numeral k) = numeral k"
  using numeral_eq_enat ..

lemma infinity_ne_numeral [simp]: "(∞::enat) ≠ numeral k"
  by (simp add: numeral_eq_enat)

lemma numeral_ne_infinity [simp]: "numeral k ≠ (∞::enat)"
  by (simp add: numeral_eq_enat)

lemma eSuc_numeral [simp]: "eSuc (numeral k) = numeral (k + Num.One)"
  by (simp only: eSuc_plus_1 numeral_plus_one)
```

14.6 Subtraction

```
instantiation enat :: minus
begin

definition diff_enat_def:
"a - b = (case a of (enat x) ⇒ (case b of (enat y) ⇒ enat (x - y) | ∞ ⇒ 0)
          | ∞ ⇒ ∞)"

instance ..

end

lemma idiff_enat_enat [simp, code]: "enat a - enat b = enat (a - b)"
  by (simp add: diff_enat_def)

lemma idiff_infinity [simp, code]: "∞ - n = (∞::enat)"
  by (simp add: diff_enat_def)

lemma idiff_infinity_right [simp, code]: "enat a - ∞ = 0"
  by (simp add: diff_enat_def)

lemma idiff_0 [simp]: "(0::enat) - n = 0"
  by (cases n, simp_all add: zero_enat_def)

lemmas idiff_enat_0 [simp] = idiff_0 [unfolded zero_enat_def]

lemma idiff_0_right [simp]: "(n::enat) - 0 = n"
  by (cases n) (simp_all add: zero_enat_def)

lemmas idiff_enat_0_right [simp] = idiff_0_right [unfolded zero_enat_def]
```

lemma *idiff_self [simp]: "n ≠ ∞ ⟹ (n::enat) - n = 0"*
by *(auto simp: zero_enat_def)*

lemma *eSuc_minus_eSuc [simp]: "eSuc n - eSuc m = n - m"*
by *(simp add: eSuc_def split: enat.split)*

lemma *eSuc_minus_1 [simp]: "eSuc n - 1 = n"*
by *(simp add: one_enat_def flip: eSuc_enat zero_enat_def)*

14.7 Ordering

instantiation *enat :: linordered_ab_semigroup_add*
begin

definition *[nitpick_simp]:*
"m ≤ n = (case n of enat n1 ⇒ (case m of enat m1 ⇒ m1 ≤ n1 | ∞ ⇒ False)
| ∞ ⇒ True)"

definition *[nitpick_simp]:*
"m < n = (case m of enat m1 ⇒ (case n of enat n1 ⇒ m1 < n1 | ∞ ⇒ True)
| ∞ ⇒ False)"

lemma *enat_ord_simps [simp]:*
"enat m ≤ enat n ⟷ m ≤ n"
"enat m < enat n ⟷ m < n"
"q ≤ (∞::enat)"
"q < (∞::enat) ⟷ q ≠ ∞"
"(∞::enat) ≤ q ⟷ q = ∞"
"(∞::enat) < q ⟷ False"
by *(simp_all add: less_eq_enat_def less_enat_def split: enat.splits)*

lemma *numeral_le_enat_iff[simp]:*
shows *"numeral m ≤ enat n ⟷ numeral m ≤ n"*
by *(auto simp: numeral_eq_enat)*

lemma *numeral_less_enat_iff[simp]:*
shows *"numeral m < enat n ⟷ numeral m < n"*
by *(auto simp: numeral_eq_enat)*

lemma *enat_ord_code [code]:*
"enat m ≤ enat n ⟷ m ≤ n"
"enat m < enat n ⟷ m < n"
"q ≤ (∞::enat) ⟷ True"
"enat m < ∞ ⟷ True"
"∞ ≤ enat n ⟷ False"
"(∞::enat) < q ⟷ False"
by *simp_all*

instance
by *standard (auto simp add: less_eq_enat_def less_enat_def plus_enat_def split: enat.splits)*

end

instance *enat :: dioid*
proof
fix *a b :: enat* **show** *"(a ≤ b) = (∃c. b = a + c)"*
by *(cases a b rule: enat2_cases) (auto simp: le_iff_add enat_ex_split)*
qed

instance *enat :: "{linordered_nonzero_semiring, strict_ordered_comm_monoid_add}"*

```
proof
  fix a b c :: enat
  show "a ≤ b ⟹ 0 ≤ c ⟹c * a ≤ c * b"
    unfolding times_enat_def less_eq_enat_def zero_enat_def
    by (simp split: enat.splits)
  show "a < b ⟹ c < d ⟹ a + c < b + d" for a b c d :: enat
    by (cases a b c d rule: enat2_cases[case_product enat2_cases]) auto
  show "a < b ⟹ a + 1 < b + 1"
    by (metis add_right_mono eSuc_minus_1 eSuc_plus_1 less_le)
qed (simp add: zero_enat_def one_enat_def)



lemma enat_ord_number [simp]:
  "(numeral m :: enat) ≤ numeral n ⟷ (numeral m :: nat) ≤ numeral n"
  "(numeral m :: enat) < numeral n ⟷ (numeral m :: nat) < numeral n"
  by (simp_all add: numeral_eq_enat)

lemma infinity_ileE [elim!]: "∞ ≤ enat m ⟹ R"
  by (simp add: zero_enat_def less_eq_enat_def split: enat.splits)

lemma infinity_ilessE [elim!]: "∞ < enat m ⟹ R"
  by simp

lemma eSuc_ile_mono [simp]: "eSuc n ≤ eSuc m ⟷ n ≤ m"
  by (simp add: eSuc_def less_eq_enat_def split: enat.splits)

lemma eSuc_mono [simp]: "eSuc n < eSuc m ⟷ n < m"
  by (simp add: eSuc_def less_enat_def split: enat.splits)

lemma ile_eSuc [simp]: "n ≤ eSuc n"
  by (simp add: eSuc_def less_eq_enat_def split: enat.splits)

lemma not_eSuc_ilei0 [simp]: "¬ eSuc n ≤ 0"
  by (simp add: zero_enat_def eSuc_def less_eq_enat_def split: enat.splits)

lemma i0_iless_eSuc [simp]: "0 < eSuc n"
  by (simp add: zero_enat_def eSuc_def less_enat_def split: enat.splits)

lemma iless_eSuc0[simp]: "(n < eSuc 0) = (n = 0)"
  by (simp add: zero_enat_def eSuc_def less_enat_def split: enat.split)

lemma ileI1: "m < n ⟹ eSuc m ≤ n"
  by (simp add: eSuc_def less_eq_enat_def less_enat_def split: enat.splits)

lemma Suc_ile_eq: "enat (Suc m) ≤ n ⟷ enat m < n"
  by (cases n) auto

lemma iless_Suc_eq [simp]: "enat m < eSuc n ⟷ enat m ≤ n"
  by (auto simp add: eSuc_def less_enat_def split: enat.splits)

lemma imult_infinity: "(0::enat) < n ⟹ ∞ * n = ∞"
  by (simp add: zero_enat_def less_enat_def split: enat.splits)

lemma imult_infinity_right: "(0::enat) < n ⟹ n * ∞ = ∞"
  by (simp add: zero_enat_def less_enat_def split: enat.splits)

lemma enat_0_less_mult_iff: "(0 < (m::enat) * n) = (0 < m ∧ 0 < n)"
  by (simp only: zero_less_iff_neq_zero mult_eq_0_iff, simp)

lemma mono_eSuc: "mono eSuc"
```

```
  by (simp add: mono_def)

lemma min_enat_simps [simp]:
  "min (enat m) (enat n) = enat (min m n)"
  "min q 0 = 0"
  "min 0 q = 0"
  "min q (∞::enat) = q"
  "min (∞::enat) q = q"
  by (auto simp add: min_def)

lemma max_enat_simps [simp]:
  "max (enat m) (enat n) = enat (max m n)"
  "max q 0 = q"
  "max 0 q = q"
  "max q ∞ = (∞::enat)"
  "max ∞ q = (∞::enat)"
  by (simp_all add: max_def)

lemma enat_ile: "n ≤ enat m ⟹ ∃k. n = enat k"
  by (cases n) simp_all

lemma enat_iless: "n < enat m ⟹ ∃k. n = enat k"
  by (cases n) simp_all

lemma iadd_le_enat_iff:
  "x + y ≤ enat n ⟷ (∃y' x'. x = enat x' ∧ y = enat y' ∧ x' + y' ≤ n)"
by(cases x y rule: enat.exhaust[case_product enat.exhaust]) simp_all

lemma chain_incr: "∀i. ∃j. Y i < Y j ⟹ ∃j. enat k < Y j"
apply (induct_tac k)
 apply (simp (no_asm) only: enat_0)
 apply (fast intro: le_less_trans [OF zero_le])
apply (erule exE)
apply (drule spec)
apply (erule exE)
apply (drule ileI1)
apply (rule eSuc_enat [THEN subst])
apply (rule exI)
apply (erule (1) le_less_trans)
done

lemma eSuc_max: "eSuc (max x y) = max (eSuc x) (eSuc y)"
  by (simp add: eSuc_def split: enat.split)

lemma eSuc_Max:
  assumes "finite A" "A ≠ {}"
  shows "eSuc (Max A) = Max (eSuc ‘ A)"
using assms proof induction
  case (insert x A)
  thus ?case by(cases "A = {}")(simp_all add: eSuc_max)
qed simp

instantiation enat :: "{order_bot, order_top}"
begin

definition bot_enat :: enat where "bot_enat = 0"
definition top_enat :: enat where "top_enat = ∞"

instance
  by standard (simp_all add: bot_enat_def top_enat_def)
```

end

lemma *finite_enat_bounded:*
  assumes *le_fin: "⋀y. y ∈ A ⟹ y ≤ enat n"*
  shows *"finite A"*
proof *(rule finite_subset)*
  show *"finite (enat ` {..n})"* **by** *blast*
  have *"A ⊆ {..enat n}"* **using** *le_fin* **by** *fastforce*
  also have *"... ⊆ enat ` {..n}"*
    apply *(rule subsetI)*
    subgoal for *x* **by** *(cases x) auto*
    done
  finally show *"A ⊆ enat ` {..n}"* **.**
qed

14.8 Cancellation simprocs

lemma *enat_add_left_cancel: "a + b = a + c ⟷ a = (∞::enat) ∨ b = c"*
  unfolding *plus_enat_def* **by** *(simp split: enat.split)*

lemma *enat_add_left_cancel_le: "a + b ≤ a + c ⟷ a = (∞::enat) ∨ b ≤ c"*
  unfolding *plus_enat_def* **by** *(simp split: enat.split)*

lemma *enat_add_left_cancel_less: "a + b < a + c ⟷ a ≠ (∞::enat) ∧ b < c"*
  unfolding *plus_enat_def* **by** *(simp split: enat.split)*

ML ‹
```
structure Cancel_Enat_Common =
struct
  (* copied from src/HOL/Tools/nat_numeral_simprocs.ML *)
  fun find_first_t _    _ []         = raise TERM("find_first_t", [])
    | find_first_t past u (t::terms) =
          if u aconv t then (rev past @ terms)
          else find_first_t (t::past) u terms

  fun dest_summing (Const (@{const_name Groups.plus}, _) $ t $ u, ts) =
        dest_summing (t, dest_summing (u, ts))
    | dest_summing (t, ts) = t :: ts

  val mk_sum = Arith_Data.long_mk_sum
  fun dest_sum t = dest_summing (t, [])
  val find_first = find_first_t []
  val trans_tac = Numeral_Simprocs.trans_tac
  val norm_ss =
    simpset_of (put_simpset HOL_basic_ss @{context}
      addsimps @{thms ac_simps add_0_left add_0_right})
  fun norm_tac ctxt = ALLGOALS (simp_tac (put_simpset norm_ss ctxt))
  fun simplify_meta_eq ctxt cancel_th th =
    Arith_Data.simplify_meta_eq [] ctxt
      ([th, cancel_th] MRS trans)
  fun mk_eq (a, b) = HOLogic.mk_Trueprop (HOLogic.mk_eq (a, b))
end

structure Eq_Enat_Cancel = ExtractCommonTermFun
(open Cancel_Enat_Common
  val mk_bal = HOLogic.mk_eq
  val dest_bal = HOLogic.dest_bin @{const_name HOL.eq} @{typ enat}
  fun simp_conv _ _ = SOME @{thm enat_add_left_cancel}
)

structure Le_Enat_Cancel = ExtractCommonTermFun
```

```
(open Cancel_Enat_Common
  val mk_bal = HOLogic.mk_binrel @{const_name Orderings.less_eq}
  val dest_bal = HOLogic.dest_bin @{const_name Orderings.less_eq} @{typ enat}
  fun simp_conv _ _ = SOME @{thm enat_add_left_cancel_le}
)

structure Less_Enat_Cancel = ExtractCommonTermFun
(open Cancel_Enat_Common
  val mk_bal = HOLogic.mk_binrel @{const_name Orderings.less}
  val dest_bal = HOLogic.dest_bin @{const_name Orderings.less} @{typ enat}
  fun simp_conv _ _ = SOME @{thm enat_add_left_cancel_less}
)
⟩

simproc_setup enat_eq_cancel
  ("(l::enat) + m = n" | "(l::enat) = m + n") =
  ⟨fn phi => fn ctxt => fn ct => Eq_Enat_Cancel.proc ctxt (Thm.term_of ct)⟩

simproc_setup enat_le_cancel
  ("(l::enat) + m ≤ n" | "(l::enat) ≤ m + n") =
  ⟨fn phi => fn ctxt => fn ct => Le_Enat_Cancel.proc ctxt (Thm.term_of ct)⟩

simproc_setup enat_less_cancel
  ("(l::enat) + m < n" | "(l::enat) < m + n") =
  ⟨fn phi => fn ctxt => fn ct => Less_Enat_Cancel.proc ctxt (Thm.term_of ct)⟩
```

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

14.9 Well-ordering

```
lemma less_enatE:
  "[| n < enat m; !!k. n = enat k ==> k < m ==> P |] ==> P"
by (induct n) auto

lemma less_infinityE:
  "[| n < ∞; !!k. n = enat k ==> P |] ==> P"
by (induct n) auto

lemma enat_less_induct:
  assumes prem: "⋀n. ∀m::enat. m < n ⟶ P m ⟹ P n" shows "P n"
proof -
  have P_enat: "⋀k. P (enat k)"
    apply (rule nat_less_induct)
    apply (rule prem, clarify)
    apply (erule less_enatE, simp)
    done
  show ?thesis
  proof (induct n)
    fix nat
    show "P (enat nat)" by (rule P_enat)
  next
    show "P ∞"
      apply (rule prem, clarify)
      apply (erule less_infinityE)
      apply (simp add: P_enat)
      done
  qed
qed
```

```
instance enat :: wellorder
proof
  fix P and n
  assume hyp: "(⋀n::enat. (⋀m::enat. m < n ⟹ P m) ⟹ P n)"
  show "P n" by (blast intro: enat_less_induct hyp)
qed
```

14.10 Complete Lattice

```
instantiation enat :: complete_lattice
begin

definition inf_enat :: "enat ⇒ enat ⇒ enat" where
  "inf_enat = min"

definition sup_enat :: "enat ⇒ enat ⇒ enat" where
  "sup_enat = max"

definition Inf_enat :: "enat set ⇒ enat" where
  "Inf_enat A = (if A = {} then ∞ else (LEAST x. x ∈ A))"

definition Sup_enat :: "enat set ⇒ enat" where
  "Sup_enat A = (if A = {} then 0 else if finite A then Max A else ∞)"
instance
proof
  fix x :: "enat" and A :: "enat set"
  { assume "x ∈ A" then show "Inf A ≤ x"
      unfolding Inf_enat_def by (auto intro: Least_le) }
  { assume "⋀y. y ∈ A ⟹ x ≤ y" then show "x ≤ Inf A"
      unfolding Inf_enat_def
      by (cases "A = {}") (auto intro: LeastI2_ex) }
  { assume "x ∈ A" then show "x ≤ Sup A"
      unfolding Sup_enat_def by (cases "finite A") auto }
  { assume "⋀y. y ∈ A ⟹ y ≤ x" then show "Sup A ≤ x"
      unfolding Sup_enat_def using finite_enat_bounded by auto }
qed (simp_all add:
 inf_enat_def sup_enat_def bot_enat_def top_enat_def Inf_enat_def Sup_enat_def)
end

instance enat :: complete_linorder ..

lemma eSuc_Sup: "A ≠ {} ⟹ eSuc (Sup A) = Sup (eSuc ‘ A)"
  by(auto simp add: Sup_enat_def eSuc_Max inj_on_def dest: finite_imageD)

lemma sup_continuous_eSuc: "sup_continuous f ⟹ sup_continuous (λx. eSuc (f x))"
  using  eSuc_Sup[of "_ ‘ UNIV"] by (auto simp: sup_continuous_def)
```

14.11 Traditional theorem names

```
lemmas enat_defs = zero_enat_def one_enat_def eSuc_def
  plus_enat_def less_eq_enat_def less_enat_def

lemma iadd_is_0: "(m + n = (0::enat)) = (m = 0 ∧ n = 0)"
  by (rule add_eq_0_iff_both_eq_0)

lemma i0_lb : "(0::enat) ≤ n"
  by (rule zero_le)

lemma ile0_eq: "n ≤ (0::enat) ⟷ n = 0"
  by (rule le_zero_eq)
```

lemma *not_iless0:* "¬ n < (0::enat)"
  by *(rule not_less_zero)*

lemma *i0_less[simp]:* "(0::enat) < n ⟷ n ≠ 0"
  by *(rule zero_less_iff_neq_zero)*

lemma *imult_is_0:* "((m::enat) * n = 0) = (m = 0 ∨ n = 0)"
  by *(rule mult_eq_0_iff)*

end

15 Linear Temporal Logic on Streams

theory *Linear_Temporal_Logic_on_Streams*
  imports *Stream Sublist Extended_Nat Infinite_Set*
begin

16 Preliminaries

lemma *shift_prefix:*
assumes *"xl @- xs = yl @- ys"* **and** *"length xl ≤ length yl"*
shows *"prefix xl yl"*
using *assms* **proof**(*induct xl arbitrary: yl xs ys*)
  case *(Cons x xl yl xs ys)*
  thus *?case* **by** *(cases yl) auto*
qed *auto*

lemma *shift_prefix_cases:*
assumes *"xl @- xs = yl @- ys"*
shows *"prefix xl yl ∨ prefix yl xl"*
using *shift_prefix[OF assms]*
by *(cases "length xl ≤ length yl") (metis, metis assms nat_le_linear shift_prefix)*

17 Linear temporal logic

Propositional connectives:

abbreviation *(input) IMPL* (**infix** *"impl" 60*)
where *"φ impl ψ ≡ λ xs. φ xs ⟶ ψ xs"*

abbreviation *(input) OR* (**infix** *"or" 60*)
where *"φ or ψ ≡ λ xs. φ xs ∨ ψ xs"*

abbreviation *(input) AND* (**infix** *"aand" 60*)
where *"φ aand ψ ≡ λ xs. φ xs ∧ ψ xs"*

abbreviation *(input)* "not φ ≡ λ xs. ¬ φ xs"

abbreviation *(input)* "true ≡ λ xs. True"

abbreviation *(input)* "false ≡ λ xs. False"

lemma *impl_not_or:* "φ impl ψ = (not φ) or ψ"
by *blast*

lemma *not_or:* "not (φ or ψ) = (not φ) aand (not ψ)"
by *blast*

lemma *not_aand:* "not (φ aand ψ) = (not φ) or (not ψ)"

by *blast*

lemma *non_not[simp]: "not (not φ) = φ"* **by** *simp*

Temporal (LTL) connectives:

fun *holds* **where** *"holds P xs ⟷ P (shd xs)"*
fun *nxt* **where** *"nxt φ xs = φ (stl xs)"*

definition *"HLD s = holds (λx. x ∈ s)"*

abbreviation *HLD_nxt* (**infixr** *"·" 65*) **where**
"s · P ≡ HLD s aand nxt P"

context
notes *[[inductive_internals]]*
begin

inductive *ev* **for** *φ* **where**
base: "φ xs ⟹ ev φ xs"
|
step: "ev φ (stl xs) ⟹ ev φ xs"

coinductive *alw* **for** *φ* **where**
alw: "⟦φ xs; alw φ (stl xs)⟧ ⟹ alw φ xs"

— weak until:
coinductive *UNTIL* (**infix** *"until" 60*) **for** *φ ψ* **where**
base: "ψ xs ⟹ (φ until ψ) xs"
|
step: "⟦φ xs; (φ until ψ) (stl xs)⟧ ⟹ (φ until ψ) xs"

end

lemma *holds_mono:*
assumes *holds: "holds P xs"* **and** *0: "⋀ x. P x ⟹ Q x"*
shows *"holds Q xs"*
using *assms* **by** *auto*

lemma *holds_aand:*
"(holds P aand holds Q) steps ⟷ holds (λ step. P step ∧ Q step) steps" **by** *auto*

lemma *HLD_iff: "HLD s ω ⟷ shd ω ∈ s"*
by *(simp add: HLD_def)*

lemma *HLD_Stream[simp]: "HLD X (x ## ω) ⟷ x ∈ X"*
by *(simp add: HLD_iff)*

lemma *nxt_mono:*
assumes *nxt: "nxt φ xs"* **and** *0: "⋀ xs. φ xs ⟹ ψ xs"*
shows *"nxt ψ xs"*
using *assms* **by** *auto*

declare *ev.intros[intro]*
declare *alw.cases[elim]*

lemma *ev_induct_strong[consumes 1, case_names base step]:*
"ev φ x ⟹ (⋀xs. φ xs ⟹ P xs) ⟹ (⋀xs. ev φ (stl xs) ⟹ ¬ φ xs ⟹ P (stl xs) ⟹ P xs)
⟹ P x"
by *(induct rule: ev.induct) auto*

lemma *alw_coinduct[consumes 1, case_names alw stl]:*

```
  "X x ⟹ (⋀x. X x ⟹ φ x) ⟹ (⋀x. X x ⟹ ¬ alw φ (stl x) ⟹ X (stl x)) ⟹ alw φ x"
  using alw.coinduct[of X x φ] by auto

lemma ev_mono:
assumes ev: "ev φ xs" and 0: "⋀ xs. φ xs ⟹ ψ xs"
shows "ev ψ xs"
using ev by induct (auto simp: 0)

lemma alw_mono:
assumes alw: "alw φ xs" and 0: "⋀ xs. φ xs ⟹ ψ xs"
shows "alw ψ xs"
using alw by coinduct (auto simp: 0)

lemma until_monoL:
assumes until: "(φ1 until ψ) xs" and 0: "⋀ xs. φ1 xs ⟹ φ2 xs"
shows "(φ2 until ψ) xs"
using until by coinduct (auto elim: UNTIL.cases simp: 0)

lemma until_monoR:
assumes until: "(φ until ψ1) xs" and 0: "⋀ xs. ψ1 xs ⟹ ψ2 xs"
shows "(φ until ψ2) xs"
using until by coinduct (auto elim: UNTIL.cases simp: 0)

lemma until_mono:
assumes until: "(φ1 until ψ1) xs" and
0: "⋀ xs. φ1 xs ⟹ φ2 xs" "⋀ xs. ψ1 xs ⟹ ψ2 xs"
shows "(φ2 until ψ2) xs"
using until by coinduct (auto elim: UNTIL.cases simp: 0)

lemma until_false: "φ until false = alw φ"
proof-
  {fix xs assume "(φ until false) xs" hence "alw φ xs"
   by coinduct (auto elim: UNTIL.cases)
  }
  moreover
  {fix xs assume "alw φ xs" hence "(φ until false) xs"
   by coinduct auto
  }
  ultimately show ?thesis by blast
qed

lemma ev_nxt: "ev φ = (φ or nxt (ev φ))"
by (rule ext) (metis ev.simps nxt.simps)

lemma alw_nxt: "alw φ = (φ aand nxt (alw φ))"
by (rule ext) (metis alw.simps nxt.simps)

lemma ev_ev[simp]: "ev (ev φ) = ev φ"
proof-
  {fix xs
   assume "ev (ev φ) xs" hence "ev φ xs"
   by induct auto
  }
  thus ?thesis by auto
qed

lemma alw_alw[simp]: "alw (alw φ) = alw φ"
proof-
  {fix xs
   assume "alw φ xs" hence "alw (alw φ) xs"
   by coinduct auto
```

```
  }
  thus ?thesis by auto
qed

lemma ev_shift:
assumes "ev φ xs"
shows "ev φ (xl @- xs)"
using assms by (induct xl) auto

lemma ev_imp_shift:
assumes "ev φ xs"  shows "∃ xl xs2. xs = xl @- xs2 ∧ φ xs2"
using assms by induct (metis shift.simps(1), metis shift.simps(2) stream.collapse)+

lemma alw_ev_shift: "alw φ xs1 ⟹ ev (alw φ) (xl @- xs1)"
by (auto intro: ev_shift)

lemma alw_shift:
assumes "alw φ (xl @- xs)"
shows "alw φ xs"
using assms by (induct xl) auto

lemma ev_ex_nxt:
assumes "ev φ xs"
shows "∃ n. (nxt ^^ n) φ xs"
using assms proof induct
  case (base xs) thus ?case by (intro exI[of _ 0]) auto
next
  case (step xs)
  then obtain n where "(nxt ^^ n) φ (stl xs)" by blast
  thus ?case by (intro exI[of _ "Suc n"]) (metis funpow.simps(2) nxt.simps o_def)
qed

lemma alw_sdrop:
assumes "alw φ xs"  shows "alw φ (sdrop n xs)"
by (metis alw_shift assms stake_sdrop)

lemma nxt_sdrop: "(nxt ^^ n) φ xs ⟷ φ (sdrop n xs)"
by (induct n arbitrary: xs) auto

definition "wait φ xs ≡ LEAST n. (nxt ^^ n) φ xs"

lemma nxt_wait:
assumes "ev φ xs"  shows "(nxt ^^ (wait φ xs)) φ xs"
unfolding wait_def using ev_ex_nxt[OF assms] by(rule LeastI_ex)

lemma nxt_wait_least:
assumes ev: "ev φ xs" and nxt: "(nxt ^^ n) φ xs"  shows "wait φ xs ≤ n"
unfolding wait_def using ev_ex_nxt[OF ev] by (metis Least_le nxt)

lemma sdrop_wait:
assumes "ev φ xs"  shows "φ (sdrop (wait φ xs) xs)"
using nxt_wait[OF assms] unfolding nxt_sdrop .

lemma sdrop_wait_least:
assumes ev: "ev φ xs" and nxt: "φ (sdrop n xs)"  shows "wait φ xs ≤ n"
using assms nxt_wait_least unfolding nxt_sdrop by auto

lemma nxt_ev: "(nxt ^^ n) φ xs ⟹ ev φ xs"
by (induct n arbitrary: xs) auto

lemma not_ev: "not (ev φ) = alw (not φ)"
```

```
proof(rule ext, safe)
  fix xs assume "not (ev φ) xs" thus "alw (not φ) xs"
  by (coinduct) auto
next
  fix xs assume "ev φ xs" and "alw (not φ) xs" thus False
  by (induct) auto
qed

lemma not_alw: "not (alw φ) = ev (not φ)"
proof-
  have "not (alw φ) = not (alw (not (not φ)))" by simp
  also have "... = ev (not φ)" unfolding not_ev[symmetric] by simp
  finally show ?thesis .
qed

lemma not_ev_not[simp]: "not (ev (not φ)) = alw φ"
unfolding not_ev by simp

lemma not_alw_not[simp]: "not (alw (not φ)) = ev φ"
unfolding not_alw by simp

lemma alw_ev_sdrop:
assumes "alw (ev φ) (sdrop m xs)"
shows "alw (ev φ) xs"
using assms
by coinduct (metis alw_nxt ev_shift funpow_swap1 nxt.simps nxt_sdrop stake_sdrop)

lemma ev_alw_imp_alw_ev:
assumes "ev (alw φ) xs"  shows "alw (ev φ) xs"
using assms by induct (metis (full_types) alw_mono ev.base, metis alw alw_nxt ev.step)

lemma alw_aand: "alw (φ aand ψ) = alw φ aand alw ψ"
proof-
  {fix xs assume "alw (φ aand ψ) xs" hence "(alw φ aand alw ψ) xs"
   by (auto elim: alw_mono)
  }
  moreover
  {fix xs assume "(alw φ aand alw ψ) xs" hence "alw (φ aand ψ) xs"
   by coinduct auto
  }
  ultimately show ?thesis by blast
qed

lemma ev_or: "ev (φ or ψ) = ev φ or ev ψ"
proof-
  {fix xs assume "(ev φ or ev ψ) xs" hence "ev (φ or ψ) xs"
   by (auto elim: ev_mono)
  }
  moreover
  {fix xs assume "ev (φ or ψ) xs" hence "(ev φ or ev ψ) xs"
   by induct auto
  }
  ultimately show ?thesis by blast
qed

lemma ev_alw_aand:
assumes φ: "ev (alw φ) xs" and ψ: "ev (alw ψ) xs"
shows "ev (alw (φ aand ψ)) xs"
proof-
  obtain xl xs1 where xs1: "xs = xl @- xs1" and φφ: "alw φ xs1"
  using φ by (metis ev_imp_shift)
```

```
  moreover obtain yl ys1 where xs2: "xs = yl @- ys1" and ψψ: "alw ψ ys1"
  using ψ by (metis ev_imp_shift)
  ultimately have 0: "xl @- xs1 = yl @- ys1" by auto
  hence "prefix xl yl ∨ prefix yl xl" using shift_prefix_cases by auto
  thus ?thesis proof
    assume "prefix xl yl"
    then obtain yl1 where yl: "yl = xl @ yl1" by (elim prefixE)
    have xs1': "xs1 = yl1 @- ys1" using 0 unfolding yl by simp
    have "alw φ ys1" using φφ unfolding xs1' by (metis alw_shift)
    hence "alw (φ aand ψ) ys1" using ψψ unfolding alw_aand by auto
    thus ?thesis unfolding xs2 by (auto intro: alw_ev_shift)
  next
    assume "prefix yl xl"
    then obtain xl1 where xl: "xl = yl @ xl1" by (elim prefixE)
    have ys1': "ys1 = xl1 @- xs1" using 0 unfolding xl by simp
    have "alw ψ xs1" using ψψ unfolding ys1' by (metis alw_shift)
    hence "alw (φ aand ψ) xs1" using φφ unfolding alw_aand by auto
    thus ?thesis unfolding xs1 by (auto intro: alw_ev_shift)
  qed
qed

lemma ev_alw_alw_impl:
assumes "ev (alw φ) xs" and "alw (alw φ impl ev ψ) xs"
shows "ev ψ xs"
using assms by induct auto

lemma ev_alw_stl[simp]: "ev (alw φ) (stl x) ⟷ ev (alw φ) x"
by (metis (full_types) alw_nxt ev_nxt nxt.simps)

lemma alw_alw_impl_ev:
"alw (alw φ impl ev ψ) = (ev (alw φ) impl alw (ev ψ))" (is "?A = ?B")
proof-
  {fix xs assume "?A xs ∧ ev (alw φ) xs" hence "alw (ev ψ) xs"
    by coinduct (auto elim: ev_alw_alw_impl)
  }
  moreover
  {fix xs assume "?B xs" hence "?A xs"
   by coinduct auto
  }
  ultimately show ?thesis by blast
qed

lemma ev_alw_impl:
assumes "ev φ xs" and "alw (φ impl ψ) xs"  shows "ev ψ xs"
using assms by induct auto

lemma ev_alw_impl_ev:
assumes "ev φ xs" and "alw (φ impl ev ψ) xs"  shows "ev ψ xs"
using ev_alw_impl[OF assms] by simp

lemma alw_mp:
assumes "alw φ xs" and "alw (φ impl ψ) xs"
shows "alw ψ xs"
proof-
  {assume "alw φ xs ∧ alw (φ impl ψ) xs" hence ?thesis
   by coinduct auto
  }
  thus ?thesis using assms by auto
qed

lemma all_imp_alw:
```

```
assumes "⋀ xs. φ xs"  shows "alw φ xs"
proof-
  {assume "∀ xs. φ xs"
   hence ?thesis by coinduct auto
  }
  thus ?thesis using assms by auto
qed

lemma alw_impl_ev_alw:
assumes "alw (φ impl ev ψ) xs"
shows "alw (ev φ impl ev ψ) xs"
using assms by coinduct (auto dest: ev_alw_impl)

lemma ev_holds_sset:
"ev (holds P) xs ⟷ (∃ x ∈ sset xs. P x)" (is "?L ⟷ ?R")
proof safe
  assume ?L thus ?R by induct (metis holds.simps stream.set_sel(1), metis stl_sset)
next
  fix x assume "x ∈ sset xs" "P x"
  thus ?L by (induct rule: sset_induct) (simp_all add: ev.base ev.step)
qed
```

LTL as a program logic:

```
lemma alw_invar:
assumes "φ xs" and "alw (φ impl nxt φ) xs"
shows "alw φ xs"
proof-
  {assume "φ xs ∧ alw (φ impl nxt φ) xs" hence ?thesis
   by coinduct auto
  }
  thus ?thesis using assms by auto
qed

lemma variance:
assumes 1: "φ xs" and 2: "alw (φ impl (ψ or nxt φ)) xs"
shows "(alw φ or ev ψ) xs"
proof-
  {assume "¬ ev ψ xs" hence "alw (not ψ) xs" unfolding not_ev[symmetric] .
   moreover have "alw (not ψ impl (φ impl nxt φ)) xs"
   using 2 by coinduct auto
   ultimately have "alw (φ impl nxt φ) xs" by(auto dest: alw_mp)
   with 1 have "alw φ xs" by(rule alw_invar)
  }
  thus ?thesis by blast
qed

lemma ev_alw_imp_nxt:
assumes e: "ev φ xs" and a: "alw (φ impl (nxt φ)) xs"
shows "ev (alw φ) xs"
proof-
  obtain xl xs1 where xs: "xs = xl @- xs1" and φ: "φ xs1"
  using e by (metis ev_imp_shift)
  have "φ xs1 ∧ alw (φ impl (nxt φ)) xs1" using a φ unfolding xs by (metis alw_shift)
  hence "alw φ xs1" by(coinduct xs1 rule: alw.coinduct) auto
  thus ?thesis unfolding xs by (auto intro: alw_ev_shift)
qed


inductive ev_at :: "('a stream ⇒ bool) ⇒ nat ⇒ 'a stream ⇒ bool" for P :: "'a stream ⇒ bool" where
  base: "P ω ⟹ ev_at P 0 ω"
| step:"¬ P ω ⟹ ev_at P n (stl ω) ⟹ ev_at P (Suc n) ω"
```

```
inductive_simps ev_at_0[simp]: "ev_at P 0 ω"
inductive_simps ev_at_Suc[simp]: "ev_at P (Suc n) ω"

lemma ev_at_imp_snth: "ev_at P n ω ⟹ P (sdrop n ω)"
  by (induction n arbitrary: ω) auto

lemma ev_at_HLD_imp_snth: "ev_at (HLD X) n ω ⟹ ω !! n ∈ X"
  by (auto dest!: ev_at_imp_snth simp: HLD_iff)

lemma ev_at_HLD_single_imp_snth: "ev_at (HLD {x}) n ω ⟹ ω !! n = x"
  by (drule ev_at_HLD_imp_snth) simp

lemma ev_at_unique: "ev_at P n ω ⟹ ev_at P m ω ⟹ n = m"
proof (induction arbitrary: m rule: ev_at.induct)
  case (base ω) then show ?case
    by (simp add: ev_at.simps[of _ _ ω])
next
  case (step ω n) from step.prems step.hyps step.IH[of "m - 1"] show ?case
    by (auto simp add: ev_at.simps[of _ _ ω])
qed

lemma ev_iff_ev_at: "ev P ω ⟷ (∃n. ev_at P n ω)"
proof
  assume "ev P ω" then show "∃n. ev_at P n ω"
    by (induction rule: ev_induct_strong) (auto intro: ev_at.intros)
next
  assume "∃n. ev_at P n ω"
  then obtain n where "ev_at P n ω"
    by auto
  then show "ev P ω"
    by induction auto
qed

lemma ev_at_shift: "ev_at (HLD X) i (stake (Suc i) ω @- ω' :: 's stream) ⟷ ev_at (HLD X) i ω"
  by (induction i arbitrary: ω) (auto simp: HLD_iff)

lemma ev_iff_ev_at_unqiue: "ev P ω ⟷ (∃!n. ev_at P n ω)"
  by (auto intro: ev_at_unique simp: ev_iff_ev_at)

lemma alw_HLD_iff_streams: "alw (HLD X) ω ⟷ ω ∈ streams X"
proof
  assume "alw (HLD X) ω" then show "ω ∈ streams X"
  proof (coinduction arbitrary: ω)
    case (streams ω) then show ?case by (cases ω) auto
  qed
next
  assume "ω ∈ streams X" then show "alw (HLD X) ω"
  proof (coinduction arbitrary: ω)
    case (alw ω) then show ?case by (cases ω) auto
  qed
qed

lemma not_HLD: "not (HLD X) = HLD (- X)"
  by (auto simp: HLD_iff)

lemma not_alw_iff: "¬ (alw P ω) ⟷ ev (not P) ω"
  using not_alw[of P] by (simp add: fun_eq_iff)

lemma not_ev_iff: "¬ (ev P ω) ⟷ alw (not P) ω"
  using not_alw_iff[of "not P" ω, symmetric]  by simp
```

```
lemma ev_Stream: "ev P (x ## s) ⟷ P (x ## s) ∨ ev P s"
  by (auto elim: ev.cases)

lemma alw_ev_imp_ev_alw:
  assumes "alw (ev P) ω" shows "ev (P aand alw (ev P)) ω"
proof -
  have "ev P ω" using assms by auto
  from this assms show ?thesis
    by induct auto
qed

lemma ev_False: "ev (λx. False) ω ⟷ False"
proof
  assume "ev (λx. False) ω" then show False
    by induct auto
qed auto

lemma alw_False: "alw (λx. False) ω ⟷ False"
  by auto

lemma ev_iff_sdrop: "ev P ω ⟷ (∃m. P (sdrop m ω))"
proof safe
  assume "ev P ω" then show "∃m. P (sdrop m ω)"
    by (induct rule: ev_induct_strong) (auto intro: exI[of _ 0] exI[of _ "Suc n" for n])
next
  fix m assume "P (sdrop m ω)" then show "ev P ω"
    by (induct m arbitrary: ω) auto
qed

lemma alw_iff_sdrop: "alw P ω ⟷ (∀m. P (sdrop m ω))"
proof safe
  fix m assume "alw P ω" then show "P (sdrop m ω)"
    by (induct m arbitrary: ω) auto
next
  assume "∀m. P (sdrop m ω)" then show "alw P ω"
    by (coinduction arbitrary: ω) (auto elim: allE[of _ 0] allE[of _ "Suc n" for n])
qed

lemma infinite_iff_alw_ev: "infinite {m. P (sdrop m ω)} ⟷ alw (ev P) ω"
  unfolding infinite_nat_iff_unbounded_le alw_iff_sdrop ev_iff_sdrop
  by simp (metis le_Suc_ex le_add1)

lemma alw_inv:
  assumes stl: "⋀s. f (stl s) = stl (f s)"
  shows "alw P (f s) ⟷ alw (λx. P (f x)) s"
proof
  assume "alw P (f s)" then show "alw (λx. P (f x)) s"
    by (coinduction arbitrary: s rule: alw_coinduct)
       (auto simp: stl)
next
  assume "alw (λx. P (f x)) s" then show "alw P (f s)"
    by (coinduction arbitrary: s rule: alw_coinduct) (auto simp flip: stl)
qed

lemma ev_inv:
  assumes stl: "⋀s. f (stl s) = stl (f s)"
  shows "ev P (f s) ⟷ ev (λx. P (f x)) s"
proof
  assume "ev P (f s)" then show "ev (λx. P (f x)) s"
    by (induction "f s" arbitrary: s) (auto simp: stl)
```

```
next
  assume "ev (λx. P (f x)) s" then show "ev P (f s)"
    by induction (auto simp flip: stl)
qed

lemma alw_smap: "alw P (smap f s) ⟷ alw (λx. P (smap f x)) s"
  by (rule alw_inv) simp

lemma ev_smap: "ev P (smap f s) ⟷ ev (λx. P (smap f x)) s"
  by (rule ev_inv) simp

lemma alw_cong:
  assumes P: "alw P ω" and eq: "⋀ω. P ω ⟹ Q1 ω ⟷ Q2 ω"
  shows "alw Q1 ω ⟷ alw Q2 ω"
proof -
  from eq have "(alw P aand Q1) = (alw P aand Q2)" by auto
  then have "alw (alw P aand Q1) ω = alw (alw P aand Q2) ω" by auto
  with P show "alw Q1 ω ⟷ alw Q2 ω"
    by (simp add: alw_aand)
qed

lemma ev_cong:
  assumes P: "alw P ω" and eq: "⋀ω. P ω ⟹ Q1 ω ⟷ Q2 ω"
  shows "ev Q1 ω ⟷ ev Q2 ω"
proof -
  from P have "alw (λxs. Q1 xs ⟶ Q2 xs) ω" by (rule alw_mono) (simp add: eq)
  moreover from P have "alw (λxs. Q2 xs ⟶ Q1 xs) ω" by (rule alw_mono) (simp add: eq)
  moreover note ev_alw_impl[of Q1 ω Q2] ev_alw_impl[of Q2 ω Q1]
  ultimately show "ev Q1 ω ⟷ ev Q2 ω"
    by auto
qed

lemma alwD: "alw P x ⟹ P x"
  by auto

lemma alw_alwD: "alw P ω ⟹ alw (alw P) ω"
  by simp

lemma alw_ev_stl: "alw (ev P) (stl ω) ⟷ alw (ev P) ω"
  by (auto intro: alw.intros)

lemma holds_Stream: "holds P (x ## s) ⟷ P x"
  by simp

lemma holds_eq1[simp]: "holds ((=) x) = HLD {x}"
  by rule (auto simp: HLD_iff)

lemma holds_eq2[simp]: "holds (λy. y = x) = HLD {x}"
  by rule (auto simp: HLD_iff)

lemma not_holds_eq[simp]: "holds (- (=) x) = not (HLD {x})"
  by rule (auto simp: HLD_iff)
```

Strong until

```
context
  notes [[inductive_internals]]
begin

inductive suntil (infix "suntil" 60) for φ ψ where
  base: "ψ ω ⟹ (φ suntil ψ) ω"
| step: "φ ω ⟹ (φ suntil ψ) (stl ω) ⟹ (φ suntil ψ) ω"
```

inductive_simps *suntil_Stream: "(φ suntil ψ) (x ## s)"*

end

lemma *suntil_induct_strong[consumes 1, case_names base step]:*
"(φ suntil ψ) x ⟹
(⋀ω. ψ ω ⟹ P ω) ⟹
(⋀ω. φ ω ⟹ ¬ ψ ω ⟹ (φ suntil ψ) (stl ω) ⟹ P (stl ω) ⟹ P ω) ⟹ P x"
using *suntil.induct[of φ ψ x P]* **by** *blast*

lemma *ev_suntil: "(φ suntil ψ) ω ⟹ ev ψ ω"*
by *(induct rule: suntil.induct) auto*

lemma *suntil_inv:*
assumes *stl: "⋀s. f (stl s) = stl (f s)"*
shows *"(P suntil Q) (f s) ⟷ ((λx. P (f x)) suntil (λx. Q (f x))) s"*
proof
assume *"(P suntil Q) (f s)"* **then show** *"((λx. P (f x)) suntil (λx. Q (f x))) s"*
by *(induction "f s" arbitrary: s) (auto simp: stl intro: suntil.intros)*
next
assume *"((λx. P (f x)) suntil (λx. Q (f x))) s"* **then show** *"(P suntil Q) (f s)"*
by *induction (auto simp flip: stl intro: suntil.intros)*
qed

lemma *suntil_smap: "(P suntil Q) (smap f s) ⟷ ((λx. P (smap f x)) suntil (λx. Q (smap f x))) s"*
by *(rule suntil_inv) simp*

lemma *hld_smap: "HLD x (smap f s) = holds (λy. f y ∈ x) s"*
by *(simp add: HLD_def)*

lemma *suntil_mono:*
assumes *eq: "⋀ω. P ω ⟹ Q1 ω ⟹ Q2 ω" "⋀ω. P ω ⟹ R1 ω ⟹ R2 ω"*
assumes *∗: "(Q1 suntil R1) ω" "alw P ω"* **shows** *"(Q2 suntil R2) ω"*
using *∗* **by** *induct (auto intro: eq suntil.intros)*

lemma *suntil_cong:*
"alw P ω ⟹ (⋀ω. P ω ⟹ Q1 ω ⟷ Q2 ω) ⟹ (⋀ω. P ω ⟹ R1 ω ⟷ R2 ω) ⟹
(Q1 suntil R1) ω ⟷ (Q2 suntil R2) ω"
using *suntil_mono[of P Q1 Q2 R1 R2 ω] suntil_mono[of P Q2 Q1 R2 R1 ω]* **by** *auto*

lemma *ev_suntil_iff: "ev (P suntil Q) ω ⟷ ev Q ω"*
proof
assume *"ev (P suntil Q) ω"* **then show** *"ev Q ω"*
by *induct (auto dest: ev_suntil)*
next
assume *"ev Q ω"* **then show** *"ev (P suntil Q) ω"*
by *induct (auto intro: suntil.intros)*
qed

lemma *true_suntil: "((λ_. True) suntil P) = ev P"*
by *(simp add: suntil_def ev_def)*

lemma *suntil_lfp: "(φ suntil ψ) = lfp (λP s. ψ s ∨ (φ s ∧ P (stl s)))"*
by *(simp add: suntil_def)*

lemma *sfilter_P[simp]: "P (shd s) ⟹ sfilter P s = shd s ## sfilter P (stl s)"*
using *sfilter_Stream[of P "shd s" "stl s"]* **by** *simp*

lemma *sfilter_not_P[simp]: "¬ P (shd s) ⟹ sfilter P s = sfilter P (stl s)"*
using *sfilter_Stream[of P "shd s" "stl s"]* **by** *simp*

```
lemma sfilter_eq:
  assumes "ev (holds P) s"
  shows "sfilter P s = x ## s' ⟷
    P x ∧ (not (holds P) suntil (HLD {x} aand nxt (λs. sfilter P s = s'))) s"
  using assms
  by (induct rule: ev_induct_strong)
     (auto simp add: HLD_iff intro: suntil.intros elim: suntil.cases)

lemma sfilter_streams:
  "alw (ev (holds P)) ω ⟹ ω ∈ streams A ⟹ sfilter P ω ∈ streams {x∈A. P x}"
proof (coinduction arbitrary: ω)
  case (streams ω)
  then have "ev (holds P) ω" by blast
  from this streams show ?case
    by (induct rule: ev_induct_strong) (auto elim: streamsE)
qed

lemma alw_sfilter:
  assumes *: "alw (ev (holds P)) s"
  shows "alw Q (sfilter P s) ⟷ alw (λx. Q (sfilter P x)) s"
proof
  assume "alw Q (sfilter P s)" with * show "alw (λx. Q (sfilter P x)) s"
  proof (coinduction arbitrary: s rule: alw_coinduct)
    case (stl s)
    then have "ev (holds P) s"
      by blast
    from this stl show ?case
      by (induct rule: ev_induct_strong) auto
  qed auto
next
  assume "alw (λx. Q (sfilter P x)) s" with * show "alw Q (sfilter P s)"
  proof (coinduction arbitrary: s rule: alw_coinduct)
    case (stl s)
    then have "ev (holds P) s"
      by blast
    from this stl show ?case
      by (induct rule: ev_induct_strong) auto
  qed auto
qed

lemma ev_sfilter:
  assumes *: "alw (ev (holds P)) s"
  shows "ev Q (sfilter P s) ⟷ ev (λx. Q (sfilter P x)) s"
proof
  assume "ev Q (sfilter P s)" from this * show "ev (λx. Q (sfilter P x)) s"
  proof (induction "sfilter P s" arbitrary: s rule: ev_induct_strong)
    case (step s)
    then have "ev (holds P) s"
      by blast
    from this step show ?case
      by (induct rule: ev_induct_strong) auto
  qed auto
next
  assume "ev (λx. Q (sfilter P x)) s" then show "ev Q (sfilter P s)"
  proof (induction rule: ev_induct_strong)
    case (step s) then show ?case
      by (cases "P (shd s)") auto
  qed auto
qed
```

```
lemma holds_sfilter:
  assumes "ev (holds Q) s" shows "holds P (sfilter Q s) ⟷ (not (holds Q) suntil (holds (Q aand P))) s"
proof
  assume "holds P (sfilter Q s)" with assms show "(not (holds Q) suntil (holds (Q aand P))) s"
    by (induct rule: ev_induct_strong) (auto intro: suntil.intros)
next
  assume "(not (holds Q) suntil (holds (Q aand P))) s" then show "holds P (sfilter Q s)"
    by induct auto
qed

lemma suntil_aand_nxt:
  "(φ suntil (φ aand nxt ψ)) ω ⟷ (φ aand nxt (φ suntil ψ)) ω"
proof
  assume "(φ suntil (φ aand nxt ψ)) ω" then show "(φ aand nxt (φ suntil ψ)) ω"
    by induction (auto intro: suntil.intros)
next
  assume "(φ aand nxt (φ suntil ψ)) ω"
  then have "(φ suntil ψ) (stl ω)" "φ ω"
    by auto
  then show "(φ suntil (φ aand nxt ψ)) ω"
    by (induction "stl ω" arbitrary: ω)
       (auto elim: suntil.cases intro: suntil.intros)
qed

lemma alw_sconst: "alw P (sconst x) ⟷ P (sconst x)"
proof
  assume "P (sconst x)" then show "alw P (sconst x)"
    by coinduction auto
qed auto

lemma ev_sconst: "ev P (sconst x) ⟷ P (sconst x)"
proof
  assume "ev P (sconst x)" then show "P (sconst x)"
    by (induction "sconst x") auto
qed auto

lemma suntil_sconst: "(φ suntil ψ) (sconst x) ⟷ ψ (sconst x)"
proof
  assume "(φ suntil ψ) (sconst x)" then show "ψ (sconst x)"
    by (induction "sconst x") auto
qed (auto intro: suntil.intros)

lemma hld_smap': "HLD x (smap f s) = HLD (f -‘ x) s"
  by (simp add: HLD_def)

lemma pigeonhole_stream:
  assumes "alw (HLD s) ω"
  assumes "finite s"
  shows "∃x∈s. alw (ev (HLD {x})) ω"
proof -
  have "∀i∈UNIV. ∃x∈s. ω !! i = x"
    using ‹alw (HLD s) ω› by (simp add: alw_iff_sdrop HLD_iff)
  from pigeonhole_infinite_rel[OF infinite_UNIV_nat ‹finite s› this]
  show ?thesis
    by (simp add: HLD_iff flip: infinite_iff_alw_ev)
qed

lemma ev_eq_suntil: "ev P ω ⟷ (not P suntil P) ω"
proof
  assume "ev P ω" then show "((λxs. ¬ P xs) suntil P) ω"
```

```
    by (induction rule: ev_induct_strong) (auto intro: suntil.intros)
qed (auto simp: ev_suntil)

end
theory EFSM_LTL
imports "../EFSM" "~~/src/HOL/Library/Linear_Temporal_Logic_on_Streams"
begin

datatype ior = ip | op | rg

record state =
  statename :: "nat option"
  datastate :: datastate
  event :: event
  "output" :: outputs

type_synonym full_observation = "state stream"
type_synonym property = "full_observation ⇒ bool"

abbreviation label :: "state ⇒ String.literal" where
  "label s ≡ fst (event s)"

abbreviation inputs :: "state ⇒ value list" where
  "inputs s ≡ snd (event s)"

fun ltl_step :: "transition_matrix ⇒ nat option ⇒ datastate ⇒ event ⇒ (nat option × outputs ×
datastate)" where
  "ltl_step _ None r _ = (None, [], r)" |
  "ltl_step e (Some s) r (l, i) = (let possibilities = possible_steps e s r l i in
                  if possibilities = {|||} then (None, [], r)
                  else
                    let (s', t) = Eps (λx. x |∈| possibilities) in
                    (Some s', (apply_outputs (Outputs t) (join_ir i r)), (apply_updates (Updates t)
(join_ir i r) r))
                 )"

primcorec make_full_observation :: "transition_matrix ⇒ nat option ⇒ datastate ⇒ event stream ⇒
full_observation" where
  "make_full_observation e s d i = (let (s', o', d') = ltl_step e s d (shd i) in (|statename = s, datastate
= d, event=(shd i), output = o'|)##(make_full_observation e s' d' (stl i)))"

lemma make_full_observation_unfold: "make_full_observation e s d i = (let (s', o', d') = ltl_step e
s d (shd i) in (|statename = s, datastate = d, event=(shd i), output = o'|)##(make_full_observation e
s' d' (stl i)))"
  using make_full_observation.code by blast

definition watch :: "transition_matrix ⇒ event stream ⇒ full_observation" where
  "watch e i ≡ (make_full_observation e (Some 0) <> i)"

abbreviation non_null :: "property" where
  "non_null s ≡ (statename (shd s) ≠ None)"

abbreviation null :: "property" where
  "null s ≡ (statename (shd s) = None)"

definition Outputs :: "nat ⇒ state stream ⇒ value option" where
  "Outputs n s ≡ nth (output (shd s)) n"

definition Inputs :: "nat ⇒ state stream ⇒ value" where
  "Inputs n s ≡ nth (inputs (shd s)) n"
```

```
definition Registers :: "nat ⇒ state stream ⇒ value option" where
  "Registers n s ≡ datastate (shd s) (R n)"

definition StateEq :: "nat option ⇒ state stream ⇒ bool" where
  "StateEq v s ≡ statename (shd s) = v"

definition LabelEq :: "string ⇒ state stream ⇒ bool" where
  "LabelEq v s ≡ fst (event (shd s)) = (String.implode v)"

lemma watch_label: "LabelEq l (watch e t) = (fst (shd t) = String.implode l)"
  by (simp add: LabelEq_def watch_def)

fun "checkInx" :: "ior ⇒ nat ⇒ (value option ⇒ value option ⇒ trilean) ⇒ value option ⇒ state
stream ⇒ bool" where
  "checkInx ior.ip n f v s = (f (Some (Inputs (n-1) s)) v = trilean.true)" |
  "checkInx ior.op n f v s = (f (Outputs n s) v = trilean.true)" |
  "checkInx ior.rg n f v s = (f (datastate (shd s) (vname.R n)) v = trilean.true)"

definition InputEq :: "value list ⇒ state stream ⇒ bool" where
  "InputEq v s ≡ inputs (shd s) = v"

definition OutputEq :: "value option list ⇒ state stream ⇒ bool" where
  "OutputEq v s ≡ output (shd s) = v"

definition InputLength :: "nat ⇒ state stream ⇒ bool" where
  "InputLength v s ≡ length (inputs (shd s)) = v"

definition OutputLength :: "nat ⇒ state stream ⇒ bool" where
  "OutputLength v s ≡ length (output (shd s)) = v"

abbreviation some_state :: "full_observation ⇒ bool" where
  "some_state s ≡ (∃state. statename (shd s) = Some state)"

lemma non_null_equiv: "non_null = some_state"
  by simp

lemma start_some_state:  "s = make_full_observation e (Some 0) <> t ⟹ some_state s"
  by simp

lemma some_until_none: "s = make_full_observation e (Some 0) <> t ⟹ (some_state until null) s "
proof (coinduction)
  case UNTIL
  then show ?case
    by (smt UNTIL.coinduct non_null_equiv)
qed

lemma shd_state_is_none: "(StateEq None) (make_full_observation e None r t)"
  by (simp add: StateEq_def)

lemma unfold_observe_none: "make_full_observation e None d t = (⦇statename = None, datastate = d, event=(shd
t), output = []⦈##(make_full_observation e None d (stl t)))"
  by (simp add: stream.expand)

lemma once_none_always_none: "alw (StateEq None) (make_full_observation e None r t)"
proof -
have f1: "(¬ StateEq None (make_full_observation e None r t) ∨ StateEq None (make_full_observation
e None r (v3_0 (λs. StateEq None (make_full_observation e None r s)) (λs. StateEq None (make_full_observation
e None r s)))) ∧ ¬ StateEq None (make_full_observation e None r (v3_0 (λs. StateEq None (make_full_observation
e None r s)) (λs. StateEq None (make_full_observation e None r s)))) ∨ StateEq None (make_full_observation
e None r (v3_1 (λs. StateEq None (make_full_observation e None r s)) (λs. StateEq None (make_full_observation
e None r s)))) ∧ ¬ alw (λs. StateEq None (make_full_observation e None r s)) (stl (v3_1 (λs. StateEq
```

```
None (make_full_observation e None r s)) (λs. StateEq None (make_full_observation e None r s)))) ∧ ¬
StateEq None (make_full_observation e None r (stl (v3_1 (λs. StateEq None (make_full_observation e None
r s)) (λs. StateEq None (make_full_observation e None r s))))) ∨ alw (λs. StateEq None (make_full_observation
e None r s)) t) = (¬ StateEq None (make_full_observation e None r t) ∨ StateEq None (make_full_observation
e None r (v3_1 (λs. StateEq None (make_full_observation e None r s)) (λs. StateEq None (make_full_observation
e None r s)))) ∧ ¬ alw (λs. StateEq None (make_full_observation e None r s)) (stl (v3_1 (λs. StateEq
None (make_full_observation e None r s)) (λs. StateEq None (make_full_observation e None r s)))) ∧ ¬
StateEq None (make_full_observation e None r (stl (v3_1 (λs. StateEq None (make_full_observation e None
r s)) (λs. StateEq None (make_full_observation e None r s))))) ∨ alw (λs. StateEq None (make_full_observation
e None r s)) t)"
  by (metis (full_types))
  have f2: "∀p s pa. (pa (s::(String.literal × value list) stream) ∧ (∀s. pa s ⟶ p s) ∧ (∀s. pa
s ∧ ¬ alw p (stl s) ⟶ pa (stl s)) ⟶ alw p s) = ((¬ pa s ∨ (∃s. pa s ∧ ¬ p s) ∨ (∃s. (pa s ∧
¬ alw p (stl s)) ∧ ¬ pa (stl s))) ∨ alw p s)"
    by blast
  obtain ss :: "((String.literal × value list) stream ⇒ bool) ⇒ ((String.literal × value list) stream
⇒ bool) ⇒ (String.literal × value list) stream" where
    f3: "∀x0 x2. (∃v3. (x2 v3 ∧ ¬ alw x0 (stl v3)) ∧ ¬ x2 (stl v3)) = ((x2 (ss x0 x2) ∧ ¬ alw x0
(stl (ss x0 x2))) ∧ ¬ x2 (stl (ss x0 x2)))"
    by moura
  obtain ssa :: "((String.literal × value list) stream ⇒ bool) ⇒ ((String.literal × value list) stream
⇒ bool) ⇒ (String.literal × value list) stream" where
    "∀x0 x2. (∃v3. x2 v3 ∧ ¬ x0 v3) = (x2 (ssa x0 x2) ∧ ¬ x0 (ssa x0 x2))"
    by moura
  then have f4: "∀p s pa. ¬ p s ∨ p (ssa pa p) ∧ ¬ pa (ssa pa p) ∨ p (ss pa p) ∧ ¬ alw pa (stl (ss
pa p)) ∧ ¬ p (stl (ss pa p)) ∨ alw pa s"
using f3 f2 by (simp add: alw_coinduct)
obtain ssb :: "((String.literal × value list) stream ⇒ state stream) ⇒ (String.literal × value list)
stream" where
  f5: "∀f p s. f (stl (ssb f)) ≠ stl (f (ssb f)) ∨ alw p (f s) = alw (λs. p (f s)) s"
  by (metis alw_inv)
  have f6: "StateEq None (make_full_observation e None r t)"
    using shd_state_is_none by auto
then have "alw (λs. StateEq None (make_full_observation e None r s)) t"
  using f6 f4 f1
  by (simp add: all_imp_alw shd_state_is_none)
  then show ?thesis
    using f5 by (metis (no_types) stream.sel(2) unfold_observe_none)
qed

lemma event_components: "(LabelEq l aand InputEq i) s = (event (shd s) = (String.implode l, i))"
  apply (simp add: LabelEq_def InputEq_def)
  by (metis fst_conv prod.collapse snd_conv)

lemma alw_not_some: "alw (λxs. statename (shd xs) ≠ Some s) (make_full_observation e None r t)"
  using once_none_always_none[of e r t]
  unfolding StateEq_def
  by (simp add: alw_mono)

lemma decompose_pair: "e ≠ (l, i) = (¬ (fst e =l ∧ snd e = i))"
  by (metis fst_conv prod.collapse sndI)

lemma check_binding_aand: "(alw x aand y) = (alw x) aand y"
  by simp

lemma check_binding_or: "(alw x or y) = (alw x) or y"
  by simp

lemma check_binding_impl: "(alw x impl y) = (alw x) impl y"
  by simp
```

```
end
theory Coin_Tea
  imports EFSM_LTL
begin

declare One_nat_def [simp del]
declare ValueLt_def [simp]
definition init :: transition where
"init ≡ (|
          Label = (STR ''init''),
          Arity = 0,
          Guard = [],
          Outputs = [],
          Updates = [(R 1, (L (Num 0)))]
        |)"

definition coin :: transition where
"coin ≡ (|
          Label = (STR ''coin''),
          Arity = 0,
          Guard = [],
          Outputs = [],
          Updates = [(R 1, (Plus (V (R 1)) (L (Num 1))))]
        |)"

definition vend :: transition where
"vend ≡ (|
          Label = (STR ''vend''),
          Arity = 0,
          Guard = [GExp.Gt (V (R 1)) (L (Num 0))],
          Outputs = [L (Str ''tea'')],
          Updates = []
        |)"

definition drinks :: "transition_matrix" where
"drinks ≡ {|
            ((0,1), init),
            ((1,1), coin),
            ((1,2), vend)
          |}"

lemma "(not (LabelEq ''vend'') until (LabelEq ''coin'')) (watch drinks t)"
  oops

lemma possible_steps_init: "possible_steps drinks 0 Map.empty STR ''init'' [] = {|(1, init)|}"
    apply (simp add: possible_steps_alt Abs_ffilter Set.filter_def drinks_def)
    apply safe
  by (simp_all add: init_def)

lemma possible_steps_not_init: "¬ (a = STR ''init'' ∧ b = []) ⟹ possible_steps drinks 0 Map.empty
a b = {||}"
    apply (simp add: possible_steps_def Abs_ffilter Set.filter_def drinks_def)
    apply clarify
    by (simp add: init_def)

lemma aux1: "¬ StateEq (Some 2)
        (make_full_observation drinks (fst (ltl_step drinks (Some 0) Map.empty (shd t)))
          (snd (snd (ltl_step drinks (Some 0) Map.empty (shd t)))) (stl t))"
proof-
  show ?thesis
    apply (case_tac "shd t")
```

```
    apply simp
    apply (case_tac "a = STR ''init'' ∧ b = []")
     apply (simp add: possible_steps_init StateEq_def)
    by (simp add: StateEq_def possible_steps_not_init)
qed

lemma make_full_obs_neq: "make_full_observation drinks (fst (ltl_step drinks (Some 0) Map.empty (shd
t))) (snd (snd (ltl_step drinks (Some 0) Map.empty (shd t))))
     (stl t) ≠
    make_full_observation drinks (Some 0) Map.empty t"
  apply (case_tac "ltl_step drinks (Some 0) Map.empty (shd t)")
  apply (case_tac "shd t")
    apply simp
    apply (case_tac "aa = STR ''init'' ∧ ba = []")
   apply (simp add: possible_steps_init init_def)
  apply (metis (no_types, lifting) make_full_observation.simps(1) option.inject state.ext_inject zero_neq_one)
  apply (simp add: possible_steps_not_init)
  by (metis make_full_observation.simps(1) option.simps(3) state.ext_inject)

lemma state_none: "((StateEq None) impl nxt (StateEq None)) (make_full_observation e s r t)"
  by (simp add: StateEq_def)

lemma shd_state_is_none: "(StateEq None) (make_full_observation e None r t)"
  by (simp add: StateEq_def)

lemma state_none_2: "(StateEq None) (make_full_observation e s r t) ⟹ (StateEq None) (make_full_observation
e s r (stl t))"
  by (simp add: StateEq_def)

lemma alw_ev: "alw f = not (ev (λs. ¬f s))"
  by simp

lemma StateEq_alt: "alw (StateEq s) s' = alw (λx. shd x = s) (smap (λx. statename x) s')"
  apply standard
  apply (simp add: StateEq_def alw_iff_sdrop)
  by (simp add: StateEq_def alw_mono alw_smap)

lemma test: "statename (shd (make_full_observation e None r t)) = None"
  by simp

lemma "alw (nxt (StateEq (Some 2)) impl (LabelEq ''vend'')) (watch drinks t)"
proof(coinduction)
  case alw
  then show ?case
    apply (case_tac "shd t")
    apply (case_tac "a = STR ''init'' ∧ b = []")
     defer
     apply (simp add: possible_steps_not_init)
    oops


lemma "alw (λs. StateEq None (stl s)) (make_full_observation drinks None Map.empty t)"
  by (metis alw_iff_sdrop once_none_always_none sdrop_simps(2))

lemma no_possible_steps: "possible_steps e s r (fst t) (snd t) = {||} ⟹ ltl_step e (Some s) r t =
(None, [], r)"
proof -
  assume "possible_steps e s r (fst t) (snd t) = {||}"
  then have "ltl_step e (Some s) r (fst t, snd t) = (None, [], r)"
    using ltl_step.simps(2) by presburger
  then show ?thesis
```

```
    by simp
qed

lemma no_possible_steps_not_init:"t ≠ (STR ''init'', []) ⟹ possible_steps drinks 0 r (fst t) (snd
t) = {||}"
  apply (simp add: possible_steps_def ffilter_def Set.filter_def drinks_def fset_both_sides Abs_fset_inverse)
  by (metis init_def length_0_conv less_numeral_extra(1) prod.collapse transition.ext_inject transition.surjectiv

lemma step_not_init: "t ≠ (STR ''init'', []) ⟹ ltl_step drinks (Some 0) r t = (None, [], r)"
  using no_possible_steps_not_init no_possible_steps
  by simp

lemma updates_init: "apply_updates (Updates init) (case_vname Map.empty Map.empty) Map.empty = <R 1
:= Num 0>"
  apply (rule ext)
  by (simp add: init_def)

lemma ltl_step_alt [simp]:  "ltl_step e (Some s) r t = (let possibilities = possible_steps e s r (fst
t) (snd t) in
                  if possibilities = {||} then (None, [], r)
                  else
                    let (s', t') = Eps (λx. x |∈| possibilities) in
                    (Some s', (apply_outputs (Transition.Outputs t') (join_ir (snd t) r)), (apply_updates
(Updates t') (join_ir (snd t) r) r))
                )"
  apply (case_tac t)
  by (simp add: Let_def)

lemma possible_steps_coin: "possible_steps drinks 1 r STR ''coin'' [] = {|(1, coin)|}"
  apply (simp add: possible_steps_alt ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
  by (simp_all add: vend_def coin_def)

lemma possible_steps_vend_insufficient: "n ≤ 0 ⟹ possible_steps drinks 1 <R 1 := Num n> STR ''vend''
[] = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
  by (simp_all add: vend_def coin_def gval.simps ValueGt_def)

lemma possible_steps_vend_sufficient: "n > 0 ⟹ possible_steps drinks 1 <R 1 := Num n> STR ''vend''
[] = {|(2, vend)|}"
  apply (simp add: possible_steps_alt ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
  by (simp_all add: vend_def coin_def gval.simps ValueGt_def)

lemma invalid_possible_steps_1: "shd t ≠ (STR ''coin'', []) ⟹
    shd t ≠ (STR ''vend'', []) ⟹ possible_steps drinks 1 r (fst (shd t)) (snd (shd t)) = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse drinks_def Set.filter_def)
  by (metis coin_def length_0_conv prod.collapse transition.ext_inject transition.surjective vend_def)

lemma updates_coin: "(apply_updates (Updates coin) (case_vname Map.empty (λna. if na = 1 then Some
(Num n) else None)) <R 1 := Num n>) = <R 1 := Num (n + 1)>"
  apply (rule ext)
  by (simp add: coin_def)

lemma updates_vend: "apply_updates (Updates vend) i r = r"
  apply (rule ext)
  by (simp add: vend_def)

lemma less_than_zero_not_nxt_2: "n ≤ 0 ⟹ ¬statename (shd (stl (make_full_observation drinks (Some
1) <R 1 := Num n> t))) = Some 2"
```

```
  apply (case_tac "shd t = (STR ''coin'', [])")
      apply (simp add: possible_steps_coin)
  apply (case_tac "shd t = (STR ''vend'', [])")
      apply (simp add: possible_steps_vend_insufficient ValueGt_def)
  by (simp add: invalid_possible_steps_1 StateEq_def)

lemma possible_steps_2: "possible_steps drinks 2 r (fst (shd t)) (snd (shd t)) = {||}"
  by (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)

lemma stop_at_2_old: "alw (λxs. statename (shd (stl xs)) = Some 2 ⟶ MaybeBoolInt (λx y. y < x) (datastate
(shd xs) (R 1)) (Some (Num 0)) = trilean.true)
     (make_full_observation drinks (Some 2) <R 1 := Num (n - 1)> t)"
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: possible_steps_2)
    using once_none_always_none
    by (simp add: StateEq_def alw_iff_sdrop)
qed

lemma reg_simp: "(λa. if a = R 1 then Some (Num n) else None) = <R 1 := Num n>"
  by (rule ext, simp)

lemma once_none_no_change_data: "alw (λxs. (datastate (shd xs)) = r) (make_full_observation e None
r t)"
proof(coinduction)
  case alw
  then show ?case
    apply simp
    by (simp add: all_imp_alw alw_inv)
qed

lemma shd_not_lt_zero: "0 ≤ n ⟹ (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0))
≠ trilean.true) (make_full_observation drinks None <R 1 := Num n> t)"
  by simp

lemma nxt_not_lt_zero: "0 ≤ n ⟹ nxt (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num
0)) ≠ trilean.true) (make_full_observation drinks None <R 1 := Num n> t)"
  by simp

lemma once_none_remains_not_lt_zero: "0 ≤ n ⟹ alw (λxs. MaybeBoolInt (<) (datastate (shd xs) (R
1)) (Some (Num 0)) ≠ trilean.true) (make_full_observation drinks None <R 1 := Num n> t)"
  using once_none_no_change_data
  by (simp add: alw_iff_sdrop)

lemma once_none_null_remains_not_lt_zero: "alw (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some
(Num 0)) ≠ trilean.true) (make_full_observation drinks None Map.empty t)"
  using once_none_no_change_data
  by (simp add: alw_iff_sdrop)

lemma stop_at_2: "0 ≤ n ⟹
     alw (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) ≠ trilean.true) (make_full_observatio
drinks (Some 2) <R 1 := Num n> t)"
proof(coinduction)
  case alw
  then show ?case
    by (simp add: possible_steps_2 once_none_remains_not_lt_zero)
qed

lemma next_not_lt_zero: "n ≥ 0 ⟹ nxt (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num
0)) ≠ trilean.true) (make_full_observation drinks (Some 1) <R 1 := Num n> t)"
```

```
    apply simp
    apply (case_tac "shd t = (STR ''vend'', [])")
    apply (case_tac "n = 0")
     apply (simp add: possible_steps_vend_insufficient)
    apply (simp add: possible_steps_vend_sufficient updates_vend)
    apply (case_tac "shd t = (STR ''coin'', [])")
    defer
    apply (simp add: invalid_possible_steps_1)
  by (simp add: possible_steps_coin updates_coin)

lemma StateEq_None_not_Some: "StateEq None s ⟹ ¬ StateEq (Some n) s"
  by (simp add: StateEq_def)

lemma not_initialised: "alw (λxs. StateEq (Some 1) xs ∧
              MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) = trilean.true ∧ LabelEq ''vend''
xs ∧ InputEq [] xs ⟶
              StateEq (Some 2) (stl xs))
     (make_full_observation drinks None Map.empty t)"
  using once_none_always_none StateEq_None_not_Some
  by (simp add: alw_iff_sdrop)

lemma implode_init: "String.implode ''init'' = STR ''init''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma not_init: "shd t ≠ (STR ''init'', []) ⟹
    LabelEq ''init'' (watch drinks t) ⟹ ¬ InputEq [] (watch drinks t)"
  apply (simp add: LabelEq_def InputEq_def implode_init watch_def)
  by (metis prod.collapse)

lemma implode_vend: "String.implode ''vend'' = STR ''vend''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_coin: "String.implode ''coin'' = STR ''coin''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma LTL_label_vend_not_2: "((LabelEq ''vend'') impl (not (ev (StateEq (Some 2))))) (watch drinks
t)"
  apply (simp only: watch_label implode_vend not_ev_iff)
  apply (simp add: watch_def)
  apply clarify
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: StateEq_def possible_steps_not_init)
    apply (rule disjI2)
    using once_none_always_none
    unfolding StateEq_def
    by (simp add: alw_iff_sdrop)
qed

lemma possible_steps_0: "possible_steps drinks 0 Map.empty l i = finsert x S' ⟹ finsert x S' = {|(1,
init)|}"
  apply (case_tac "l = STR ''init''")
   apply (case_tac "i = []")
    apply (simp add: possible_steps_init)
  using possible_steps_not_init
  by auto

lemma vend_insufficient: "possible_steps drinks 1 <R 1 := Num 0> STR ''vend'' i = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
```

```
   apply (simp add: coin_def)
  by (simp add: vend_def gval.simps ValueGt_def)

lemma LTL_aux2: "((nxt (LabelEq ''vend'')) impl not (ev (StateEq (Some 2)))) (watch drinks t)"
  apply (simp add: watch_def LabelEq_def implode_vend not_ev_iff)
  apply clarify
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: StateEq_def)
    apply (case_tac "shd t = (STR ''init'', [])")
     defer
    using possible_steps_not_init alw_not_some
     apply (simp add: no_possible_steps_not_init)
    apply (simp add: possible_steps_init updates_init)
    apply (rule disjI2)
  proof(coinduction)
    case alw
    then show ?case
      apply (simp add: vend_insufficient)
      apply (rule disjI2)
      using alw_not_some
      by simp
  qed
qed
lemma LTL_init_makes_r_1_zero:
  "((LabelEq ''init'' aand InputEq []) impl
    (nxt (checkInx rg 1 ValueEq (Some (Num 0)))))
   (watch drinks t)"

  apply (case_tac "shd t = (STR ''init'', [])")
  using watch_def
   apply (simp add: possible_steps_init updates_init ValueEq_def)
  apply clarify
  by (simp add: not_init)


lemma LTL_must_pay_wrong: "((not (LabelEq ''vend'' suntil LabelEq ''coin'')) suntil StateEq None) (watch
drinks t)"
  oops

lemma shd_not_init: "shd t ≠ (STR ''init'', []) ⟹ ¬ ev (λs. statename (shd s) = Some 2) (make_full_observati
drinks (Some 0) Map.empty t)"
  apply (simp add: not_ev_iff)
proof(coinduction)
  case alw
  then show ?case
    apply simp
    apply (case_tac "shd t")
    apply simp
    by (simp add: possible_steps_not_init alw_not_some)
qed

lemma vend_gets_stuck: "stl t = (STR ''vend'', []) ## x2 ⟹ ¬ev (λs. statename (shd s) = Some 2)
(make_full_observation drinks (Some 1) <R 1 := Num 0> ((STR ''vend'', []) ## x2))"
  apply (simp add: not_ev_iff)
proof(coinduction)
  case alw
  then show ?case
    by (simp add: vend_insufficient alw_not_some)
qed
```

```
lemma possible_steps_1_invalid: "x1 ≠ (STR ''coin'', []) ⟹
      x1 ≠ (STR ''vend'', []) ⟹
      possible_steps drinks 1 <R 1 := Num 0> (fst x1) (snd x1) = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse drinks_def Set.filter_def)
  apply safe
   apply (simp add: coin_def)
   apply (metis prod.collapse)
  by (simp add: vend_def gval.simps ValueGt_def)

lemma invalid_gets_stuck: "x1 ≠ (STR ''coin'', []) ⟹
                           x1 ≠ (STR ''vend'', []) ⟹
                           ¬ev (λs. statename (shd s) = Some 2) (make_full_observation drinks (Some
1) <R 1 := Num 0> (x1 ## x2))"
  apply (simp add: not_ev_iff)
proof(coinduction)
  case alw
  then show ?case
    by (simp add: possible_steps_1_invalid alw_not_some)
qed

lemma LTL_vend_no_coin: "((nxt (LabelEq ''vend'' aand InputEq [])) impl not (ev (StateEq (Some 2))))
(watch drinks t)"
  apply (simp add: not_ev_iff event_components implode_vend watch_def StateEq_def)
  apply clarify
proof(coinduction)
  case alw
  then show ?case
    apply simp
    apply (case_tac "shd t = (STR ''init'', [])")
     defer
    apply (simp add: decompose_pair)
     apply (simp add: possible_steps_not_init alw_not_some)
    apply (simp add: possible_steps_init updates_init)
    apply (rule disjI2)
  proof(coinduction)
    case alw
    then show ?case
      apply (simp add: vend_insufficient)
     by (simp add: possible_steps_not_init alw_not_some)
  qed
qed

lemma LTL_invalid_gets_stuck_2:
  "(((nxt (not (LabelEq ''coin'' aand InputEq []))) aand
   (nxt (not (LabelEq ''vend'' aand InputEq [])))) impl
   (not (ev (StateEq (Some 2))))) (watch drinks t)"
  apply (simp add: not_ev_iff event_components)
  unfolding watch_def StateEq_def LabelEq_def InputEq_def
  apply clarify
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: implode_coin implode_vend)
    apply (case_tac "shd t = (STR ''init'', [])")
     defer
     apply (simp only: decompose_pair)
    using possible_steps_not_init alw_not_some
     apply simp
    apply (simp add: possible_steps_init updates_init)
    apply (rule disjI2)
```

```
    using invalid_gets_stuck[of "shd (stl t)" "stl (stl t)"]
    by (simp add: alw_ev)
qed


lemma LTL_must_pay_correct_bracketed:
  "((ev (StateEq (Some 2))) impl
    ((not (LabelEq ''vend'')) suntil LabelEq ''coin''))
   (watch drinks t)"
  oops
lemma LTL_must_pay_correct:
  "((ev (StateEq (Some 2))) impl
    (not (LabelEq ''vend'') suntil LabelEq ''coin''))
   (watch drinks t)"
  apply clarify
  unfolding LabelEq_def StateEq_def implode_vend implode_coin
  apply (simp add: watch_def)
  apply (case_tac "shd t = (STR ''init'', [])")
   apply (rule suntil.step)
    apply simp
   apply (simp add: possible_steps_init updates_init)
   apply (case_tac "shd (stl t) = (STR ''coin'', [])")
    apply (simp add: suntil.base)
   apply (case_tac "shd (stl t) = (STR ''vend'', [])")
    apply (rule suntil.step)
  using watch_def LTL_vend_no_coin[of t]
     apply (simp add: event_components implode_vend StateEq_def ev_mono)
  using watch_def LTL_vend_no_coin[of t]
    apply (simp add: event_components implode_vend StateEq_def ev_mono)
  using StateEq_def watch_def LTL_invalid_gets_stuck_2[of t]
   apply (simp add: event_components implode_vend implode_coin ev_mono)
  by (simp add: shd_not_init)

end
theory Coin_Tea_Broken
  imports EFSM_LTL
begin

declare One_nat_def [simp del]
declare ValueLt_def [simp]
definition init :: transition where
"init ≡ (|
          Label = (STR ''init''),
          Arity = 0,
          Guard = [],
          Outputs = [],
          Updates = [(R 1, (L (Num 0)))]
        |)"

definition coin :: transition where
"coin ≡ (|
          Label = (STR ''coin''),
          Arity = 0,
          Guard = [],
          Outputs = [],
          Updates = [(R 1, (Plus (V (R 1)) (L (Num 1))))]
        |)"

definition vend :: transition where
"vend ≡ (|
          Label = (STR ''vend''),
```

```
          Arity = 0,
          Guard = [GExp.Ge (V (R 1)) (L (Num 0))],
          Outputs = [L (Str ''tea'')],
          Updates = []
        |)"

definition drinks :: "transition_matrix" where
"drinks ≡ {|
            ((0,1), init),
            ((1,1), coin),
            ((1,2), vend)
          |}"

lemma "(not (LabelEq ''vend'') until (LabelEq ''coin'')) (watch drinks t)"
  oops

lemma possible_steps_init: "possible_steps drinks 0 Map.empty STR ''init'' [] = {|(1, init)|}"
    apply (simp add: possible_steps_alt Abs_ffilter Set.filter_def drinks_def)
    apply safe
  by (simp_all add: init_def)

lemma possible_steps_not_init: "¬ (a = STR ''init'' ∧ b = []) ⟹ possible_steps drinks 0 Map.empty
a b = {||}"
    apply (simp add: possible_steps_def Abs_ffilter Set.filter_def drinks_def)
    apply clarify
    by (simp add: init_def)

lemma aux1: "¬ StateEq (Some 2)
        (make_full_observation drinks (fst (ltl_step drinks (Some 0) Map.empty (shd t)))
          (snd (snd (ltl_step drinks (Some 0) Map.empty (shd t)))) (stl t))"
proof-
  show ?thesis
    apply (case_tac "shd t")
    apply simp
    apply (case_tac "a = STR ''init'' ∧ b = []")
     apply (simp add: possible_steps_init StateEq_def)
    by (simp add: StateEq_def possible_steps_not_init)
qed

lemma make_full_obs_neq: "make_full_observation drinks (fst (ltl_step drinks (Some 0) Map.empty (shd
t))) (snd (snd (ltl_step drinks (Some 0) Map.empty (shd t))))
     (stl t) ≠
    make_full_observation drinks (Some 0) Map.empty t"
  apply (case_tac "ltl_step drinks (Some 0) Map.empty (shd t)")
  apply (case_tac "shd t")
    apply simp
    apply (case_tac "aa = STR ''init'' ∧ ba = []")
   apply (simp add: possible_steps_init init_def)
  apply (metis (no_types, lifting) make_full_observation.simps(1) option.inject state.ext_inject stream.sel(1)
zero_neq_one)
  apply (simp add: possible_steps_not_init)
  by (metis make_full_observation.simps(1) option.simps(3) state.ext_inject stream.sel(1))

lemma state_none: "((StateEq None) impl nxt (StateEq None)) (make_full_observation e s r t)"
  by (simp add: StateEq_def)

lemma shd_state_is_none: "(StateEq None) (make_full_observation e None r t)"
  by (simp add: StateEq_def)

lemma state_none_2: "(StateEq None) (make_full_observation e s r t) ⟹ (StateEq None) (make_full_observation
e s r (stl t))"
```

```
  by (simp add: StateEq_def)

lemma alw_ev: "alw f = not (ev (λs. ¬f s))"
  by simp

lemma StateEq_alt: "alw (StateEq s) s' = alw (λx. shd x = s) (smap (λx. statename x) s')"
  apply standard
  apply (simp add: StateEq_def alw_iff_sdrop)
  by (simp add: StateEq_def alw_mono alw_smap)

lemma test: "statename (shd (make_full_observation e None r t)) = None"
  by simp

lemma "alw (nxt (StateEq (Some 2)) impl (LabelEq ''vend'')) (watch drinks t)"
proof(coinduction)
  case alw
  then show ?case
    apply (case_tac "shd t")
    apply (case_tac "a = STR ''init'' ∧ b = []")
     defer
     apply (simp add: possible_steps_not_init)
    oops


lemma "alw (λs. StateEq None (stl s)) (make_full_observation drinks None Map.empty t)"
  by (metis alw_iff_sdrop once_none_always_none sdrop_simps(2))

lemma no_possible_steps: "possible_steps e s r (fst t) (snd t) = {||} ⟹ ltl_step e (Some s) r t =
(None, [], r)"
proof -
  assume "possible_steps e s r (fst t) (snd t) = {||}"
  then have "ltl_step e (Some s) r (fst t, snd t) = (None, [], r)"
    using ltl_step.simps(2) by presburger
  then show ?thesis
    by simp
qed

lemma no_possible_steps_not_init:"t ≠ (STR ''init'', []) ⟹ possible_steps drinks 0 r (fst t) (snd
t) = {||}"
  apply (simp add: possible_steps_def ffilter_def Set.filter_def drinks_def fset_both_sides Abs_fset_inverse)
  by (metis init_def length_0_conv less_numeral_extra(1) prod.collapse transition.ext_inject transition.surjectiv

lemma step_not_init: "t ≠ (STR ''init'', []) ⟹ ltl_step drinks (Some 0) r t = (None, [], r)"
  using no_possible_steps_not_init no_possible_steps
  by simp

lemma updates_init: "apply_updates (Updates init) (case_vname Map.empty Map.empty) Map.empty = <R 1
:= Num 0>"
  apply (rule ext)
  by (simp add: init_def)

lemma ltl_step_alt [simp]:  "ltl_step e (Some s) r t = (let possibilities = possible_steps e s r (fst
t) (snd t) in
                  if possibilities = {||} then (None, [], r)
                  else
                    let (s', t') = Eps (λx. x |∈| possibilities) in
                    (Some s', (apply_outputs (Transition.Outputs t') (join_ir (snd t) r)), (apply_updates
(Updates t') (join_ir (snd t) r) r))
                 )"
  apply (case_tac t)
  by (simp add: Let_def)
```

```
lemma possible_steps_coin: "possible_steps drinks 1 r STR ''coin'' [] = {|(1, coin)|}"
  apply (simp add: possible_steps_alt ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
  by (simp_all add: vend_def coin_def)

lemma possible_steps_vend_insufficient: "n ≤ 0 ⟹ possible_steps drinks 1 <R 1 := Num n> STR ''vend''
[] = {||}"
  oops

lemma possible_steps_vend_sufficient: "n > 0 ⟹ possible_steps drinks 1 <R 1 := Num n> STR ''vend''
[] = {|(2, vend)|}"
  apply (simp add: possible_steps_alt ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
  by (simp_all add: vend_def coin_def gval.simps ValueGt_def)

lemma invalid_possible_steps_1: "shd t ≠ (STR ''coin'', []) ⟹
    shd t ≠ (STR ''vend'', []) ⟹ possible_steps drinks 1 r (fst (shd t)) (snd (shd t)) = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse drinks_def Set.filter_def)
  by (metis coin_def length_0_conv prod.collapse transition.ext_inject transition.surjective vend_def)

lemma updates_coin: "(apply_updates (Updates coin) (case_vname Map.empty (λna. if na = 1 then Some
(Num n) else None)) <R 1 := Num n>) = <R 1 := Num (n + 1)>"
  apply (rule ext)
  by (simp add: coin_def)

lemma updates_vend: "apply_updates (Updates vend) i r = r"
  apply (rule ext)
  by (simp add: vend_def)

lemma less_than_zero_not_nxt_2: "n ≤ 0 ⟹ ¬statename (shd (stl (make_full_observation drinks (Some
1) <R 1 := Num n> t))) = Some 2"
  apply (case_tac "shd t = (STR ''coin'', [])")
     apply (simp add: possible_steps_coin)
  apply (case_tac "shd t = (STR ''vend'', [])")
  oops

lemma possible_steps_2: "possible_steps drinks 2 r (fst (shd t)) (snd (shd t)) = {||}"
  by (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)

lemma stop_at_2_old: "alw (λxs. statename (shd (stl xs)) = Some 2 ⟶ MaybeBoolInt (λx y. y < x) (datastate
(shd xs) (R 1)) (Some (Num 0)) = trilean.true)
     (make_full_observation drinks (Some 2) <R 1 := Num (n - 1)> t)"
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: possible_steps_2)
    using once_none_always_none
    by (simp add: StateEq_def alw_iff_sdrop)
qed

lemma reg_simp: "(λa. if a = R 1 then Some (Num n) else None) = <R 1 := Num n>"
  by (rule ext, simp)

lemma once_none_no_change_data: "alw (λxs. (datastate (shd xs)) = r) (make_full_observation e None
r t)"
proof(coinduction)
  case alw
  then show ?case
    apply simp
    by (simp add: all_imp_alw alw_inv)
```

qed

lemma *shd_not_lt_zero: "0 ≤ n ⟹ (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) ≠ trilean.true) (make_full_observation drinks None <R 1 := Num n> t)"*
  by *simp*

lemma *nxt_not_lt_zero: "0 ≤ n ⟹ nxt (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) ≠ trilean.true) (make_full_observation drinks None <R 1 := Num n> t)"*
  by *simp*

lemma *once_none_remains_not_lt_zero: "0 ≤ n ⟹ alw (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) ≠ trilean.true) (make_full_observation drinks None <R 1 := Num n> t)"*
  using *once_none_no_change_data*
  by *(simp add: alw_iff_sdrop)*

lemma *once_none_null_remains_not_lt_zero: "alw (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) ≠ trilean.true) (make_full_observation drinks None Map.empty t)"*
  using *once_none_no_change_data*
  by *(simp add: alw_iff_sdrop)*

lemma *stop_at_2: "0 ≤ n ⟹*
      alw (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) ≠ trilean.true) (make_full_observatio drinks (Some 2) <R 1 := Num n> t)"
proof*(coinduction)*
  case *alw*
  then show *?case*
    by *(simp add: possible_steps_2 once_none_remains_not_lt_zero)*
qed

lemma *next_not_lt_zero: "n ≥ 0 ⟹ nxt (λxs. MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) ≠ trilean.true) (make_full_observation drinks (Some 1) <R 1 := Num n> t)"*
    apply *simp*
    apply *(case_tac "shd t = (STR ''vend'', [])")*
    apply *(case_tac "n = 0")*
  oops

lemma *StateEq_None_not_Some: "StateEq None s ⟹ ¬ StateEq (Some n) s"*
  by *(simp add: StateEq_def)*

lemma *not_initialised: "alw (λxs. StateEq (Some 1) xs ∧*
              MaybeBoolInt (<) (datastate (shd xs) (R 1)) (Some (Num 0)) = trilean.true ∧ LabelEq ''vend'' xs ∧ InputEq [] xs ⟶
              StateEq (Some 2) (stl xs))
     (make_full_observation drinks None Map.empty t)"
  using *once_none_always_none StateEq_None_not_Some*
  by *(simp add: alw_iff_sdrop)*

lemma *implode_init: "String.implode ''init'' = STR ''init''"*
  by *(metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)*

lemma *not_init: "shd t ≠ (STR ''init'', []) ⟹*
    LabelEq ''init'' (watch drinks t) ⟹ ¬ InputEq [] (watch drinks t)"
  apply *(simp add: LabelEq_def InputEq_def implode_init watch_def)*
  by *(metis prod.collapse)*

lemma *implode_vend: "String.implode ''vend'' = STR ''vend''"*
  by *(metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)*

lemma *implode_coin: "String.implode ''coin'' = STR ''coin''"*
  by *(metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)*

```
lemma LTL_label_vend_not_2: "((LabelEq ''vend'') impl (not (ev (StateEq (Some 2))))) (watch drinks
t)"
  apply (simp only: watch_label implode_vend not_ev_iff)
  apply (simp add: watch_def)
  apply clarify
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: StateEq_def possible_steps_not_init)
    apply (rule disjI2)
    using once_none_always_none
    unfolding StateEq_def
    by (simp add: alw_iff_sdrop)
qed

lemma possible_steps_0: "possible_steps drinks 0 Map.empty l i = finsert x S' ⟹ finsert x S' = {|(1,
init)|}"
  apply (case_tac "l = STR ''init''")
   apply (case_tac "i = []")
    apply (simp add: possible_steps_init)
  using possible_steps_not_init
  by auto

lemma vend_insufficient: "possible_steps drinks 1 <R 1 := Num 0> STR ''vend'' i = {||}"
  oops

lemma LTL_aux2: "((nxt (LabelEq ''vend'')) impl not (ev (StateEq (Some 2)))) (watch drinks t)"
  apply (simp add: watch_def LabelEq_def implode_vend not_ev_iff)
  apply clarify
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: StateEq_def)
    apply (case_tac "shd t = (STR ''init'', [])")
     defer
    using possible_steps_not_init alw_not_some
     apply (simp add: no_possible_steps_not_init)
    apply (simp add: possible_steps_init updates_init)
    apply (rule disjI2)
  proof(coinduction)
    case alw
    then show ?case
      oops

lemma LTL_init_makes_r_1_zero: "((LabelEq ''init'' aand InputEq []) impl nxt (checkInx rg 1 ValueEq
(Some (Num 0)))) (watch drinks t)"
  apply (case_tac "shd t = (STR ''init'', [])")
   apply (simp add: possible_steps_init updates_init ValueEq_def watch_def)
  apply clarify
  using not_init
  by simp

lemma LTL_must_pay_wrong: "((not (LabelEq ''vend'' suntil LabelEq ''coin'')) suntil StateEq None) (watch
drinks t)"
  oops

lemma shd_not_init: "shd t ≠ (STR ''init'', []) ⟹ ¬ ev (λs. statename (shd s) = Some 2) (make_full_observati
drinks (Some 0) Map.empty t)"
  apply (simp add: not_ev_iff)
proof(coinduction)
  case alw
```

```
  then show ?case
    apply simp
    apply (case_tac "shd t")
    apply simp
    by (simp add: possible_steps_not_init alw_not_some)
qed

lemma vend_gets_stuck: "stl t = (STR ''vend'', []) ## x2 ⟹ ¬ev (λs. statename (shd s) = Some 2)
(make_full_observation drinks (Some 1) <R 1 := Num 0> ((STR ''vend'', []) ## x2))"
  apply (simp add: not_ev_iff)
proof(coinduction)
  case alw
  then show ?case
    oops

lemma possible_steps_1_invalid: "x1 ≠ (STR ''coin'', []) ⟹
      x1 ≠ (STR ''vend'', []) ⟹
      possible_steps drinks 1 <R 1 := Num 0> (fst x1) (snd x1) = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse drinks_def Set.filter_def)
  apply safe
   apply (simp add: coin_def)
   apply (metis prod.collapse)
  oops

lemma invalid_gets_stuck: "x1 ≠ (STR ''coin'', []) ⟹
                           x1 ≠ (STR ''vend'', []) ⟹
                           ¬ev (λs. statename (shd s) = Some 2) (make_full_observation drinks (Some
1) <R 1 := Num 0> (x1 ## x2))"
  apply (simp add: not_ev_iff)
proof(coinduction)
  case alw
  then show ?case
    oops

lemma LTL_vend_no_coin: "((nxt (LabelEq ''vend'' aand InputEq [])) impl not (ev (StateEq (Some 2))))
(watch drinks t)"
  apply (simp add: not_ev_iff event_components implode_vend watch_def StateEq_def)
  apply clarify
proof(coinduction)
  case alw
  then show ?case
    apply simp
    apply (case_tac "shd t = (STR ''init'', [])")
     defer
    apply (simp add: decompose_pair)
     apply (simp add: possible_steps_not_init alw_not_some)
    apply (simp add: possible_steps_init updates_init)
    apply (rule disjI2)
  proof(coinduction)
    case alw
    then show ?case
      oops

lemma LTL_invalid_gets_stuck_2:
  "(((nxt (not (LabelEq ''coin'' aand InputEq []))) aand
   (nxt (not (LabelEq ''vend'' aand InputEq [])))) impl
   (not (ev (StateEq (Some 2))))) (watch drinks t)"
  apply (simp add: not_ev_iff event_components)
  unfolding watch_def StateEq_def LabelEq_def InputEq_def
  apply clarify
proof(coinduction)
```

```
  case alw
  then show ?case
    apply (simp add: implode_coin implode_vend)
    apply (case_tac "shd t = (STR ''init'', [])")
     defer
     apply (simp only: decompose_pair)
    using possible_steps_not_init alw_not_some
     apply simp
    apply (simp add: possible_steps_init updates_init)
    apply (rule disjI2)
    oops


lemma LTL_must_pay_correct: "((ev (StateEq (Some 2))) impl (not (LabelEq ''vend'') suntil LabelEq ''coin''))
(watch drinks t)"
  apply clarify
  unfolding LabelEq_def StateEq_def implode_vend implode_coin
  apply (simp add: watch_def)
  apply (case_tac "shd t = (STR ''init'', [])")
   defer
   apply (simp add: shd_not_init)
  apply (rule suntil.step)
   apply simp
  apply (simp add: possible_steps_init updates_init)
  apply (case_tac "shd (stl t) = (STR ''coin'', [])")
   apply (simp add: suntil.base)
  apply (case_tac "shd (stl t) = (STR ''vend'', [])")
   apply (rule suntil.step)
  oops




end
```

References

[1] M. Foster, R. G. Taylor, A. D. Brucker, and J. Derrick. Formalising extended finite state machine transition merging. In J. S. Dong and J. Sun, editors, *ICFEM*, LNCS. Springer, 2018. URL `http://www.brucker.ch/bibliography/abstract/foster.ea-efsm-2018`.