

A Formal Model of Extended Finite State Machine

Michael Foster

Ramsay G. Taylor
John Derrick

Achim D. Brucker

July 26, 2019

Department of Computer Science

The University of Sheffield

Sheffield, UK

`{jmafooster1, a.brucker, r.g.taylor, j.derrick }@sheffield.ac.uk`

In this article, we formalize

Keywords:

Contents

1	Introduction	5
2	Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums	6
2.1	The datatype universe	6
2.2	Freeness: Distinctness of Constructors	8
2.3	Set Constructions	10
3	Bijections between natural numbers and other types	14
3.1	Type $\text{nat} \times \text{nat}$	14
3.2	Type $\text{nat} + \text{nat}$	15
3.3	Type int	16
3.4	Type nat list	17
3.5	Finite sets of naturals	17
4	Encoding (almost) everything into natural numbers	20
4.1	The class of countable types	20
4.2	Conversion functions	20
4.3	Finite types are countable	21
4.4	Automatically proving countability of old-style datatypes	21
4.5	Automatically proving countability of datatypes	23
4.6	More Countable types	23
4.7	The rationals are countably infinite	24
5	Type of finite sets defined as a subtype of sets	25
5.1	Definition of the type	25
5.2	Basic operations and type class instantiations	25
5.3	Other operations	28
5.4	Transferred lemmas from Set.thy	28
5.5	Additional lemmas	33
5.6	Choice in fsets	39
5.7	Induction and Cases rules for fsets	39
5.8	Setup for Lifting/Transfer	41
5.9	BNF setup	43
5.10	Size setup	44
5.11	Advanced relator customization	45
5.12	Quickcheck setup	46
6	Extended Finite State Machines	57
6.1	Arithmetic Expressions	57
6.2	Guard Expressions	67
6.3	Extended Finite State Machines	80
7	Infinite Streams	89
7.1	prepend list to stream	90
7.2	set of streams with elements in some fixed set	90
7.3	nth, take, drop for streams	91
7.4	unary predicates lifted to streams	93
7.5	recurring stream out of a list	94
7.6	iterated application of a function	95
7.7	stream repeating a single element	95
7.8	stream of natural numbers	96
7.9	flatten a stream of lists	96
7.10	merge a stream of streams	97
7.11	product of two streams	97
7.12	interleave two streams	98

7.13	zip	98
7.14	zip via function	99
8	List prefixes, suffixes, and homeomorphic embedding	99
8.1	Prefix order on lists	99
8.2	Basic properties of prefixes	100
8.3	Prefixes	103
8.4	Longest Common Prefix	104
8.5	Parallel lists	106
8.6	Suffix order on lists	107
8.7	Suffixes	111
8.8	Homeomorphic embedding on lists	112
8.9	Subsequences (special case of homeomorphic embedding)	114
8.10	Appending elements	116
8.11	Relation to standard list operations	117
8.12	Contiguous sublists	117
8.13	Parametricity	120
9	Infinite Sets and Related Concepts	122
9.1	The set of natural numbers is infinite	122
9.2	The set of integers is also infinite	123
9.3	Infinitely Many and Almost All	123
9.4	Enumeration of an Infinite Set	125
10	Countable sets	128
10.1	Predicate for countable sets	128
10.2	Enumerate a countable set	128
10.3	Closure properties of countability	131
10.4	Misc lemmas	133
10.5	Uncountable	134
11	Countable Complete Lattices	135
12	Continuity and iterations	139
12.1	Continuity for complete lattices	140
13	Extended natural numbers (i.e. with infinity)	147
13.1	Type definition	147
13.2	Constructors and numbers	148
13.3	Addition	149
13.4	Multiplication	150
13.5	Numerals	151
13.6	Subtraction	151
13.7	Ordering	152
13.8	Cancellation simprocs	155
13.9	Well-ordering	156
13.10	Complete Lattice	157
13.11	Traditional theorem names	158
14	Linear Temporal Logic on Streams	158
15	Preliminaries	158
16	Linear temporal logic	158

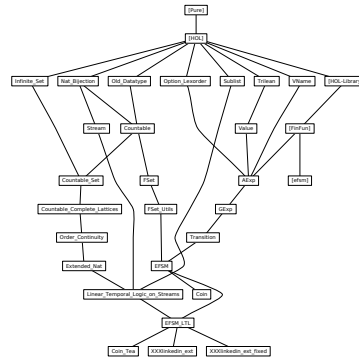


Figure 1: The Dependency Graph of the Isabelle Theories.

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.  $\exists f x k. p = (f :: \text{nat} \Rightarrow 'b + \text{nat}, x :: 'a + \text{nat}) \wedge f k = \text{Inr } 0\}$ "
```

```
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 o Inl"
```

```
definition Numb :: "nat => ('a, 'b) dtree"
where "Numb == Atom o 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 ==  $\bigcup \{z. \exists x. z = \text{Push\_Node } (\text{Inl } x) ' (f x)\}$ "
```

```
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 ==  $\bigcup x:A. \bigcup y:B. \{ \text{Scons } x y \}$ "
```

```

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.  $\exists x y. M = \text{Scons } x y \wedge u = c \ x \ y$ "

definition Case :: "[('a, 'b) dtree] => 'c, [('a, 'b) dtree] => 'c, ('a, 'b) dtree] => 'c"
  where "Case c d M == THE u.  $(\exists x. M = \text{In0}(x) \wedge u = c(x)) \vee (\exists y. M = \text{In1}(y) \wedge u = d(y))$ "

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')}"(\lambda k. \text{Inr } 0, a) \in \text{Node}"
by (simp add: Node_def)

lemma Node_Push_I: " $p \in \text{Node} \implies \text{apfst } (\text{Push } i) \ p \in \text{Node}$ "
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  $\neq$  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) o h1) = (ntrunc(k) o 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 $\text{nat} \times \text{nat}$

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

```

definition triangle :: "nat  $\Rightarrow$  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  $\times$  nat  $\Rightarrow$  nat"
  where "prod_encode = ( $\lambda(m, n).$  triangle (m + n) + m)"

```

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

```

fun prod_decode_aux :: "nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\times$  nat"
  where "prod_decode_aux k m =
    (if m  $\leq$  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  $\Rightarrow$  nat  $\times$  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  $\longleftrightarrow$  x = y"
  by (rule inj_prod_encode [THEN inj_eq])

lemma prod_decode_eq: "prod_decode x = prod_decode y  $\longleftrightarrow$  x = y"
  by (rule inj_prod_decode [THEN inj_eq])

  Ordering properties

lemma le_prod_encode_1: "a  $\leq$  prod_encode (a, b)"
  by (simp add: prod_encode_def)

lemma le_prod_encode_2: "b  $\leq$  prod_encode (a, b)"
  by (induct b) (simp_all add: prod_encode_def)

```

3.2 Type $\text{nat} + \text{nat}$

```

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

definition sum_decode :: "nat  $\Rightarrow$  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  $\longleftrightarrow$  x = y"
  by (rule inj_sum_encode [THEN inj_eq])

lemma sum_decode_eq: "sum_decode x = sum_decode y  $\longleftrightarrow$  x = y"
  by (rule inj_sum_decode [THEN inj_eq])

```

3.3 Type *int*

```

definition int_encode :: "int  $\Rightarrow$  nat"
  where "int_encode i = sum_encode (if 0  $\leq$  i then Inl (nat i) else Inr (nat (- i - 1)))"

definition int_decode :: "nat  $\Rightarrow$  int"
  where "int_decode n = (case sum_decode n of Inl a  $\Rightarrow$  int a | Inr b  $\Rightarrow$  - 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  $\longleftrightarrow$  x = y"
  by (rule inj_int_encode [THEN inj_eq])

lemma int_decode_eq: "int_decode x = int_decode y  $\longleftrightarrow$  x = y"
  by (rule inj_int_decode [THEN inj_eq])

```


3.4 Type `nat list`

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

function list_decode :: "nat  $\Rightarrow$  nat list"
  where
    "list_decode 0 = []"
  | "list_decode (Suc n) = (case prod_decode n of (x, y)  $\Rightarrow$  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  $\longleftrightarrow$  x = y"
  by (rule inj_list_encode [THEN inj_eq])

lemma list_decode_eq: "list_decode x = list_decode y  $\longleftrightarrow$  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)  $\longleftrightarrow$  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  $\longleftrightarrow$  even y"
  shows "x = y"
proof -
  from (even x  $\longleftrightarrow$  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  $\Rightarrow$  nat"
  where "set_encode = sum (( $\wedge$ ) 2)"

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

lemma set_encode_inf: " $\neg$  finite A  $\implies$  set_encode A = 0"
  by (simp add: set_encode_def)

lemma set_encode_insert [simp]: "finite A  $\implies$  n  $\notin$  A  $\implies$  set_encode (insert n A) = 2n + set_encode A"
  by (simp add: set_encode_def)

lemma even_set_encode_iff: "finite A  $\implies$  even (set_encode A)  $\longleftrightarrow$  0  $\notin$  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  $\Rightarrow$  nat set"
  where "set_decode x = {n. odd (x div 2 ^ n)}"

lemma set_decode_0 [simp]: "0  $\in$  set_decode x  $\longleftrightarrow$  odd x"
  by (simp add: set_decode_def)

lemma set_decode_Suc [simp]: "Suc n  $\in$  set_decode x  $\longleftrightarrow$  n  $\in$  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  $\notin$  set_decode z  $\implies$  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  $\in$  set_decode (2 ^ 0 + z)  $\longleftrightarrow$  q  $\in$  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  $\in$  set_decode (2 ^ Suc n + z)  $\longleftrightarrow$  q  $\in$  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  $\implies$  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  $\implies$  finite B  $\implies$  set_encode A = set_encode B  $\longleftrightarrow$  A = B"
  by (rule iffI) (simp_all add: inj_onD [OF inj_on_set_encode])

lemma subset_decode_imp_le:
  assumes "set_decode m  $\subseteq$  set_decode n"
  shows "m  $\leq$  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: " $\exists$  to_nat :: 'a  $\Rightarrow$  nat. inj to_nat"

lemma countable_classI:
  fixes f :: "'a  $\Rightarrow$  nat"
  assumes " $\bigwedge x y. f x = f y \implies x = y$ "
  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  $\Rightarrow$  nat" where
  "to_nat = (SOME f. inj f)"

definition from_nat :: "nat  $\Rightarrow$  'a::countable" where
  "from_nat = inv (to_nat :: 'a  $\Rightarrow$  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  $\longleftrightarrow$  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  $\Rightarrow$  '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 " $\exists$  to_nat :: 'a  $\Rightarrow$  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  $\Rightarrow$  bool" where
  undefined: "finite_item undefined"
| In0: "finite_item x  $\Longrightarrow$  finite_item (Old_Datatype.In0 x)"
| In1: "finite_item x  $\Longrightarrow$  finite_item (Old_Datatype.In1 x)"
| Leaf: "finite_item (Old_Datatype.Leaf a)"
| Scons: "[finite_item x; finite_item y]  $\Longrightarrow$  finite_item (Old_Datatype.Scons x y)"

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

lemma le_sum_encode_Inl: "x  $\leq$  y  $\Longrightarrow$  x  $\leq$  sum_encode (Inl y)"
unfolding sum_encode_def by simp

lemma le_sum_encode_Inr: "x  $\leq$  y  $\Longrightarrow$  x  $\leq$  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  $\implies \exists n. \text{nth\_item } n = x"$ 
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  $\Rightarrow$  ('a::countable) Old_Datatype.item"
  fixes Abs :: "('a::countable) Old_Datatype.item  $\Rightarrow$  'b"
  fixes rep_set :: "('a::countable) Old_Datatype.item  $\Rightarrow$  bool"
  assumes type: "type_definition Rep Abs (Collect rep_set)"
  assumes finite_item: " $\bigwedge x. \text{rep\_set } x \implies \text{finite\_item } x$ "
  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 " $\exists n. \text{nth\_item } n = \text{Rep } y$ "
      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 " $\exists f::'b \Rightarrow \text{nat}. \text{inj } f$ "
    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: " $\mathbb{Q}$  = 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:
  " $\mathbb{Q}$  = range (of_rat  $\circ$  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:
  " $r \in \mathbb{Q} \implies \exists n. r = \text{of\_rat } (\text{nat\_to\_rat\_surj } n)$ "
  by (simp add: Rats_eq_range_of_rat_o_nat_to_rat_surj image_def)

end

instance rat :: countable
proof
  show " $\exists \text{to\_nat} :: \text{rat} \Rightarrow \text{nat}. \text{inj to\_nat}$ "
  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: " $\exists f :: \text{nat} \Rightarrow \text{rat}. \text{surj } f$ "
  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  $\Rightarrow$  'a fset  $\Rightarrow$  bool" is subset_eq parametric subset_transfer
.

definition less_fset :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  bool" where "xs < ys  $\equiv$  xs  $\leq$  ys  $\wedge$  xs  $\neq$  (ys::'a fset)"

lemma less_fset_transfer[transfer_rule]:
  includes lifting_syntax
  assumes [transfer_rule]: "bi_unique A"
  shows "(pcr_fset A)  $\implies$  (pcr_fset A)  $\implies$  ( $=$ ) ( $\subset$ ) ( $<$ )"
  unfolding less_fset_def[abs_def] psubset_eq[abs_def] by transfer_prover

lift_definition sup_fset :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" is union parametric union_transfer
  by simp

lift_definition inf_fset :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" is inter parametric inter_transfer
  by simp

lift_definition minus_fset :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" is minus parametric Diff_transfer
  by simp

instance
  by (standard; transfer; auto)+

end

abbreviation fempty :: "'a fset" ("{|}|") where "{|}|  $\equiv$  bot"
abbreviation fsubset_eq :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  bool" (infix "| $\subseteq$ |" 50) where "xs | $\subseteq$ | ys  $\equiv$  xs  $\leq$  ys"
abbreviation fsubset :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  bool" (infix "| $\subset$ |" 50) where "xs | $\subset$ | ys  $\equiv$  xs < ys"
abbreviation funion :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" (infixl "| $\cup$ |" 65) where "xs | $\cup$ | ys  $\equiv$  sup xs ys"
abbreviation finter :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" (infixl "| $\cap$ |" 65) where "xs | $\cap$ | ys  $\equiv$  inf xs ys"
abbreviation fminus :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" (infixl "| $-$ |" 65) where "xs | $-$ | ys  $\equiv$  minus xs ys"

instantiation fset :: (equal) equal
begin
definition "HOL.equal A B  $\longleftrightarrow$  A | $\subseteq$ | B  $\wedge$  B | $\subseteq$ | 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)  $\implies$  rel_set A)
    ( $\lambda$ S. if finite ( $\bigcap$  S  $\cap$  Collect (Domainp A)) then  $\bigcap$  S  $\cap$  Collect (Domainp A) else {})
    ( $\lambda$ S. 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)  $\implies$  rel_set A) ( $\lambda$ A. if finite (Inf A) then Inf A else {})
    ( $\lambda$ A. if finite (Inf A) then Inf A else {})"
  by transfer_prover

```

```

lift_definition Inf_fset :: "'a fset set  $\Rightarrow$  'a fset" is " $\lambda A. \text{if finite (Inf A) then Inf A else \{ \}}$ "
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)  $\implies$  rel_set A) ( $\lambda A. \text{if finite (Sup A) then Sup A else \{ \}}$ )"
    ( $\lambda A. \text{if finite (Sup A) then Sup A else \{ \}}$ )" by transfer_prover

lift_definition Sup_fset :: "'a fset set  $\Rightarrow$  'a fset" is " $\lambda A. \text{if finite (Sup A) then Sup A else \{ \}}$ "
parametric Sup_fset_transfer by simp

lemma finite_Sup: " $\exists z. \text{finite } z \wedge (\forall a. a \in X \longrightarrow a \leq z) \implies \text{finite (Sup X)}$ "
by (auto intro: finite_subset)

lemma transfer_bdd_below[transfer_rule]: "(rel_set (pcr_fset (=))  $\implies$  (=)) bdd_below bdd_below"
by auto

end

instance
proof
  fix x z :: "'a fset"
  fix X :: "'a fset set"
  {
    assume "x  $\in$  X" "bdd_below X"
    then show "Inf X  $\sqsubseteq$  x" by transfer auto
  }
next
  assume "X  $\neq \{ \}$ " " $(\bigwedge x. x \in X \implies z \sqsubseteq x)$ "
  then show "z  $\sqsubseteq$  Inf X" by transfer (clarsimp, blast)
next
  assume "x  $\in$  X" "bdd_above X"
  then obtain z where "x  $\in$  X" " $(\bigwedge x. x \in X \implies x \sqsubseteq z)$ "
    by (auto simp: bdd_above_def)
  then show "x  $\sqsubseteq$  Sup X"
    by transfer (auto intro!: finite_Sup)
next
  assume "X  $\neq \{ \}$ " " $(\bigwedge x. x \in X \implies x \sqsubseteq z)$ "
  then show "Sup X  $\sqsubseteq$  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  $\Rightarrow$  '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  $\equiv$  top"
abbreviation fuminus :: "'a::finite fset  $\Rightarrow$  'a fset" ("|-|_" [81] 80) where "|-| x  $\equiv$  uminus x"

```

```

declare top_fset.rep_eq[simp]

```

5.3 Other operations

```

lift_definition finset :: "'a  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" is insert parametric Lifting_Set.insert_transfer
  by simp

```

```

syntax
  "_insert_fset"      :: "args  $\Rightarrow$  'a fset"  ("{|(_)|}")

```

```

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

```

```

lift_definition fmember :: "'a  $\Rightarrow$  'a fset  $\Rightarrow$  bool" (infix "| $\in$ |" 50) is Set.member
  parametric member_transfer .

```

```

abbreviation notin_fset :: "'a  $\Rightarrow$  'a fset  $\Rightarrow$  bool" (infix "| $\notin$ |" 50) where "x | $\notin$ | S  $\equiv$   $\neg$  (x | $\in$ | S)"

```

```

context includes lifting_syntax
begin

```

```

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

```

```

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

```

```

lift_definition fcard :: "'a fset  $\Rightarrow$  nat" is card parametric card_transfer .

```

```

lift_definition fimage :: "('a  $\Rightarrow$  'b)  $\Rightarrow$  'a fset  $\Rightarrow$  'b fset" (infixr "|'" 90) is image
  parametric image_transfer by simp

```

```

lift_definition fthe_elem :: "'a fset  $\Rightarrow$  'a" is the_elem .

```

```

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

```

```

lift_definition ffUnion :: "'a fset fset  $\Rightarrow$  'a fset" is Union parametric Union_transfer by simp

```

```

lift_definition fBall :: "'a fset  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool" is Ball parametric Ball_transfer .

```

```

lift_definition fBex :: "'a fset  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool" is Bex parametric Bex_transfer .

```

```

lift_definition ffold :: "('a  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  'a fset  $\Rightarrow$  'b" is Finite_Set.fold .

```

```

lift_definition fset_of_list :: "'a list  $\Rightarrow$  'a fset" is set by (rule finite_set)

```

```

lift_definition sorted_list_of_fset :: "'a::linorder fset  $\Rightarrow$  '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 = rev_subsetD[Transfer.transferred]
lemmas fset_mp = subsetD[Transfer.transferred]
lemmas fsubset_not_fsubset_eq[code] = subset_not_subset_eq[Transfer.transferred]
lemmas eq_fmemp_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 = equalsOI[Transfer.transferred]
lemmas equalsffemptyD = equalsOD[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_fm1 = if_split_mem1[Transfer.transferred]
lemmas if_split_fm2 = if_split_mem2[Transfer.transferred]
lemmas pfssubsetI[intro!,no_atp] = psubsetI[Transfer.transferred]
lemmas pfssubsetE[elim!,no_atp] = psubsetE[Transfer.transferred]
lemmas pfssubset_finsert_iff = psubset_insert_iff[Transfer.transferred]
lemmas pfssubset_eq = psubset_eq[Transfer.transferred]
lemmas pfssubset_imp_fsubset = psubset_imp_subset[Transfer.transferred]
lemmas pfssubset_trans = psubset_trans[Transfer.transferred]
lemmas pfssubsetD = psubsetD[Transfer.transferred]
lemmas pfssubset_fsubset_trans = psubset_subset_trans[Transfer.transferred]
lemmas fsubset_pfssubset_trans = subset_psubset_trans[Transfer.transferred]
lemmas pfssubset_imp_ex_fm1 = 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_pfssubset_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 fininsert_absorb2[simp] = insert_absorb2[Transfer.transferred]
lemmas fininsert_commute = insert_commute[Transfer.transferred]
lemmas fininsert_fsubset[simp] = insert_subset[Transfer.transferred]
lemmas fininsert_inter_finsert[simp] = insert_inter_insert[Transfer.transferred]
lemmas fininsert_disjoint[simp,no_atp] = insert_disjoint[Transfer.transferred]
lemmas disjoint_fininsert[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 fininsert_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_iffempty[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_iffempty[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_psubset_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 nonempty_fbind_const = nonempty_bind_const[Transfer.transferred]
lemmas fbind_const = bind_const[Transfer.transferred]
lemmas fmember_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*]: " $A = B \implies (\bigwedge x. x \in A \implies f\ x = g\ x) \implies fbind\ A\ f = fbind\ B\ g$ "
by *transfer force*

5.5.3 *fsingleton*

lemmas *fsingletonE* = *fsingletonD* [*elim_format*]

5.5.4 *fempty*

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

lemma *femptyE* [*elim!*]: " $a \in \{\} \implies P$ "
by *simp*

5.5.5 *fset*

lemmas *fset_simps*[simp] = *bot_fset.rep_eq* *finset.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* \cap *fset* *xs*"
by *transfer auto*

lemma *notin_fset*: " $x \notin S \longleftrightarrow x \notin fset\ S$ " 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* \subseteq *ffilter* *Q* *A* = $(\forall x. x \in A \longrightarrow P\ x \longrightarrow Q\ x)$ "
by *transfer auto*

lemma *eq_ffilter*:
"*ffilter* *P* *A* = *ffilter* *Q* *A* = $(\forall x. x \in A \longrightarrow P\ x = Q\ x)$ "
by *transfer auto*

lemma *pfssubset_ffilter*:
" $(\bigwedge x. x \in A \implies P\ x \implies Q\ x) \implies (x \in A \wedge \neg P\ x \wedge Q\ x) \implies$
ffilter *P* *A* \subset *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  $\subseteq$  set ys  $\implies$  fset_of_list xs  $\subseteq$  fset_of_list ys"
  by transfer simp

lemma fset_of_list_elem: "(x  $\in$  fset_of_list xs)  $\longleftrightarrow$  (x  $\in$  set xs)"
  by transfer simp

```

5.5.8 fininsert

```

lemma set_fininsert:
  assumes "x  $\in$  A"
  obtains B where "A = fininsert x B" and "x  $\notin$  B"
using assms by transfer (metis Set.set_insert finite_insert)

lemma mk_disjoint_fininsert: "a  $\in$  A  $\implies \exists B. A = fininsert a B \wedge a \notin B"$ 
  by (rule exI [where x = "A -| {a}"]) blast

lemma fininsert_eq_iff:
  assumes "a  $\notin$  A" and "b  $\notin$  B"
  shows "(fininsert a A = fininsert b B) =
    (if a = b then A = B else  $\exists C. A = fininsert b C \wedge b \notin C \wedge B = fininsert a C \wedge a \notin C$ )"
  using assms by transfer (force simp: insert_eq_iff)

```

5.5.9 fimage

```

lemma subset_fimage_iff: "(B  $\subseteq$  f|'|A) = ( $\exists AA. AA \subseteq A \wedge B = f|'|AA$ )"
  by transfer (metis mem_Collect_eq rev_finite_subset subset_image_iff)

```

5.5.10 bounded quantification

```

lemma bex_simps [simp, no_atp]:
  " $\bigwedge A P Q. fBex A (\lambda x. P x \wedge Q) = (fBex A P \wedge Q)$ "
  " $\bigwedge A P Q. fBex A (\lambda x. P \wedge Q x) = (P \wedge fBex A Q)$ "
  " $\bigwedge P. fBex \{\} P = False$ "
  " $\bigwedge a B P. fBex (fininsert a B) P = (P a \vee fBex B P)$ "
  " $\bigwedge A P f. fBex (f |'| A) P = fBex A (\lambda x. P (f x))$ "
  " $\bigwedge A P. (\neg fBex A P) = fBall A (\lambda x. \neg P x)$ "
by auto

lemma ball_simps [simp, no_atp]:
  " $\bigwedge A P Q. fBall A (\lambda x. P x \vee Q) = (fBall A P \vee Q)$ "
  " $\bigwedge A P Q. fBall A (\lambda x. P \vee Q x) = (P \vee fBall A Q)$ "
  " $\bigwedge A P Q. fBall A (\lambda x. P \longrightarrow Q x) = (P \longrightarrow fBall A Q)$ "
  " $\bigwedge A P Q. fBall A (\lambda x. P x \longrightarrow Q) = (fBex A P \longrightarrow Q)$ "
  " $\bigwedge P. fBall \{\} P = True$ "
  " $\bigwedge a B P. fBall (fininsert a B) P = (P a \wedge fBall B P)$ "
  " $\bigwedge A P f. fBall (f |'| A) P = fBall A (\lambda x. P (f x))$ "
  " $\bigwedge A P. (\neg fBall A P) = fBex A (\lambda x. \neg P x)$ "
by auto

lemma atomize_fBall:
  " $(\bigwedge x. x \in A \implies P x) == Trueprop (fBall A (\lambda x. P x))$ "
apply (simp only: atomize_all atomize_imp)
apply (rule equal_intr_rule)
  by (transfer, simp)+

```

```

lemma fBall_mono[mono]: " $P \leq Q \implies fBall\ S\ P \leq fBall\ S\ Q$ "
by auto

lemma fBex_mono[mono]: " $P \leq Q \implies fBex\ S\ P \leq fBex\ S\ Q$ "
by auto

end

5.5.11 fcard

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

lemma fcard_finsert_disjoint:
  " $x \notin A \implies fcard\ (finsert\ x\ A) = Suc\ (fcard\ A)$ "
  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  $\longleftrightarrow$  A = {}"
  by transfer (rule card_0_eq)

lemma fcard_Suc_fminus1:
  " $x \in A \implies Suc\ (fcard\ (A - \{x\})) = fcard\ A$ "
  by transfer (rule card_Suc_Diff1)

lemma fcard_fminus_fsingleton:
  " $x \in A \implies fcard\ (A - \{x\}) = fcard\ A - 1$ "
  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  $\implies$  fcard A ≤ fcard B"
  by transfer (rule card_mono)

lemma fcard_seteq: "A ⊆ B  $\implies$  fcard B ≤ fcard A  $\implies$  A = B"
by transfer (rule card_seteq)

lemma psubset_fcard_mono: "A ⊂ B  $\implies$  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_psubset: "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_fininsert [simp]:
    assumes "x |∉| A"
    shows "ffold f z (fininsert 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_fininsert2:
    "x |∉| A ⇒ ffold f z (fininsert 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_fininsert_remove:
    "ffold f z (fininsert 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_fininsert_idem:
  "ffold f z (fininsert x A) = f x (ffold f z A)"
  by (transfer fixing: f) (rule fold_insert_idem)

declare ffold_fininsert [simp del] ffold_fininsert_idem [simp]

lemma ffold_fininsert_idem2:
  "ffold f z (fininsert 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 cong_simp[cong]:
  "⋈ A = B; ⋀x. x |∈| B =simp=> g x = h x ⋈ ⟹ F g A = F h B"
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  $\Rightarrow$  '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  $\notin$  A  $\implies$  A  $\neq$  {|}|  $\implies$  F (finsert x A) = x * F A"
  by transfer (rule set.insert_not_elem)

lemma in_idem: "x  $\in$  A  $\implies$  x * F A = F A"
  by transfer (rule set.in_idem)

lemma insert [simp]: "A  $\neq$  {|}|  $\implies$  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 ( $\leq$ ) ( $<$ )" 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 ( $\geq$ ) ( $>$ )"
    by standard

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

end

lemma mono_fMax_commute: "mono f  $\implies$  A  $\neq$  {|}|  $\implies$  f (fMax A) = fMax (f |' A)"
  by transfer (rule mono_Max_commute)

lemma mono_fMin_commute: "mono f  $\implies$  A  $\neq$  {|}|  $\implies$  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: " $S = \{\mid\} \implies P$ "
    and      finsert_case: " $\bigwedge x S'. S = \text{fininsert } x S' \implies P$ "
    shows "P"
    using assms by transfer blast

lemma fset_induct [case_names empty insert]:
  assumes fempty_case: " $P \{\mid\}$ "
  and      finsert_case: " $\bigwedge x S. P S \implies P (\text{fininsert } x S)$ "
  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: " $P \{\mid\}$ "
  and      insert_fset_case: " $\bigwedge x S. \llbracket x \notin S; P S \rrbracket \implies P (\text{fininsert } x S)$ "
  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: " $P \{\mid\}$ "
  and      card_fset_Suc_case: " $\bigwedge S T. \text{Suc } (\text{fcard } S) = (\text{fcard } T) \implies P S \implies P T$ "
  shows "P S"
proof (induct S)
  case empty
  show "P  $\{\mid\}$ " by (rule empty_fset_case)
next
  case (insert x S)
  have h: "P S" by fact
  have " $x \notin S$ " by fact
  then have " $\text{Suc } (\text{fcard } S) = \text{fcard } (\text{fininsert } x S)$ "
    by transfer auto
  then show "P (fininsert x S)"
    using h card_fset_Suc_case by simp
qed

lemma fset_strong_cases:
  obtains "xs =  $\{\mid\}$ "
    | ys x where " $x \notin \text{ys}$ " and " $\text{xs} = \text{fininsert } x \text{ys}$ "
  by transfer blast

lemma fset_induct2:
  " $P \{\mid\} \{\mid\} \implies$ "
  ( $\bigwedge x \text{xs}. x \notin \text{xs} \implies P (\text{fininsert } x \text{xs}) \{\mid\} \implies$ 
  ( $\bigwedge y \text{ys}. y \notin \text{ys} \implies P \{\mid\} (\text{fininsert } y \text{ys}) \implies$ 
  ( $\bigwedge x \text{xs } y \text{ys}. \llbracket P \text{xs ys}; x \notin \text{xs}; y \notin \text{ys} \rrbracket \implies P (\text{fininsert } x \text{xs}) (\text{fininsert } y \text{ys}) \implies$ 
  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 finset_transfer [transfer_rule]:
  "(A ==> rel_fset A ==> rel_fset A) finset finset"
  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  $\longleftrightarrow$  ( $\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u$ )  $\wedge$  ( $\forall t \in \text{fset } b. \exists u \in \text{fset } a. R \ u \ t$ )"
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:
  " $(\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u) \wedge (\forall u \in \text{fset } b. \exists t \in \text{fset } a. R \ t \ u) \longleftrightarrow$ 
  ( $(\text{BNF\_Def.Grp } \{a. \text{fset } a \subseteq \{(a, b). R \ a \ b\} \} \text{ (fimage fst)})^{-1-1} \ 00$ 
   $\text{BNF\_Def.Grp } \{a. \text{fset } a \subseteq \{(a, b). R \ a \ b\} \} \text{ (fimage snd)} \} \ a \ b$  (is "?L = ?R)")
proof
  assume ?L
  define R' where "R' =
    the_inv fset (Collect (case_prod R)  $\cap$  (fset a  $\times$  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 relcomp.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 conversesep.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 conversesep.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  $\chi$  (fset A1) (fset A2) = rel_fset  $\chi$  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  $\Rightarrow$  nat)  $\Rightarrow$  'a fset  $\Rightarrow$  nat" is " $\lambda f. \text{sum } (\text{Suc } \circ f)$ " .
end

instantiation fset :: (type) size begin
definition size_fset where
  size_fset_overloaded_def: "size_fset = FSet.size_fset ( $\lambda_. 0$ )"
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 " $\lambda_. 0$ ", unfolded add_0_left add_0_right,
    folded size_fset_overloaded_def]

lemma fset_size_o_map: "inj f  $\implies$  size_fset g  $\circ$  fimage f = size_fset (g  $\circ$  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  $\chi$   $\varphi$ ) A1 A2  $\longleftrightarrow$ 
   rel_set  $\chi$  (Inl -' A1) (Inl -' A2)  $\wedge$  rel_set  $\varphi$  (Inr -' A1) (Inr -' A2)"
  (is "?L  $\longleftrightarrow$  ?Rl  $\wedge$  ?Rr")
proof safe
  assume L: "?L"
  show ?Rl unfolding rel_set_def Bex_def vimage_eq proof safe
    fix l1 assume "Inl l1  $\in$  A1"
    then obtain a2 where a2: "a2  $\in$  A2" and "rel_sum  $\chi$   $\varphi$  (Inl l1) a2"
    using L unfolding rel_set_def by auto
    then obtain l2 where "a2 = Inl l2  $\wedge$   $\chi$  l1 l2" by (cases a2, auto)
    thus " $\exists$  l2. Inl l2  $\in$  A2  $\wedge$   $\chi$  l1 l2" using a2 by auto
  next
    fix l2 assume "Inl l2  $\in$  A2"
    then obtain a1 where a1: "a1  $\in$  A1" and "rel_sum  $\chi$   $\varphi$  a1 (Inl l2)"
    using L unfolding rel_set_def by auto
    then obtain l1 where "a1 = Inl l1  $\wedge$   $\chi$  l1 l2" by (cases a1, auto)
    thus " $\exists$  l1. Inl l1  $\in$  A1  $\wedge$   $\chi$  l1 l2" using a1 by auto
  qed
  show ?Rr unfolding rel_set_def Bex_def vimage_eq proof safe
    fix r1 assume "Inr r1  $\in$  A1"
    then obtain a2 where a2: "a2  $\in$  A2" and "rel_sum  $\chi$   $\varphi$  (Inr r1) a2"
    using L unfolding rel_set_def by auto
    then obtain r2 where "a2 = Inr r2  $\wedge$   $\varphi$  r1 r2" by (cases a2, auto)
    thus " $\exists$  r2. Inr r2  $\in$  A2  $\wedge$   $\varphi$  r1 r2" using a2 by auto
  next
    fix r2 assume "Inr r2  $\in$  A2"
    then obtain a1 where a1: "a1  $\in$  A1" and "rel_sum  $\chi$   $\varphi$  a1 (Inr r2)"
    using L unfolding rel_set_def by auto
    then obtain r1 where "a1 = Inr r1  $\wedge$   $\varphi$  r1 r2" by (cases a1, auto)
    thus " $\exists$  r1. Inr r1  $\in$  A1  $\wedge$   $\varphi$  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  $\in$  A1"
    show " $\exists$  a2. a2  $\in$  A2  $\wedge$  rel_sum  $\chi$   $\varphi$  a1 a2"
    proof(cases a1)
      case (Inl l1) then obtain l2 where "Inl l2  $\in$  A2  $\wedge$   $\chi$  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  $\in$  A2  $\wedge$   $\varphi$  r1 r2"
        using Rr a1 unfolding rel_set_def by blast
      thus ?thesis unfolding Inr by auto
    qed
  next
    fix a2 assume a2: "a2  $\in$  A2"
    show " $\exists$  a1. a1  $\in$  A1  $\wedge$  rel_sum  $\chi$   $\varphi$  a1 a2"
    proof(cases a2)
      case (Inl l2) then obtain l1 where "Inl l1  $\in$  A1  $\wedge$   $\chi$  l1 l2"

```

```

    using R1 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_compose)
  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 finset {·} (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_Exhaustive
    (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_finsert
  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_Natural.Suc i) j =
    Quickcheck_Random.collapse (Random.select_weight
      [(1, Pair valterm_femptyset),
      (Code_Natural.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

theory Trilean
imports Main
begin

datatype trilean = true | false | invalid

instantiation trilean :: semiring begin
fun times_trilean :: "trilean ⇒ trilean ⇒ trilean" where
  "times_trilean _ invalid = invalid" |
  "times_trilean invalid _ = invalid" |
  "times_trilean true true = true" |
  "times_trilean _ false = false" |

```

```

"times_trilean false _ = false"

fun plus_trilean :: "trilean  $\Rightarrow$  trilean  $\Rightarrow$  trilean" where
  "plus_trilean invalid _ = invalid" |
  "plus_trilean _ invalid = invalid" |
  "plus_trilean true _ = true" |
  "plus_trilean _ true = true" |
  "plus_trilean false false = false"

abbreviation maybe_and :: "trilean  $\Rightarrow$  trilean  $\Rightarrow$  trilean" (infixl " $\wedge$ ?" 70) where
  "maybe_and x y  $\equiv$  x * y"

abbreviation maybe_or :: "trilean  $\Rightarrow$  trilean  $\Rightarrow$  trilean" (infixl " $\vee$ ?" 65) where
  "maybe_or x y  $\equiv$  x + y"

lemma plus_trilean_assoc: "a  $\vee$ ? b  $\vee$ ? c = a  $\vee$ ? (b  $\vee$ ? c)"
proof(induct a b arbitrary: c rule: plus_trilean.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 plus_trilean.simps(2) plus_trilean.simps(4) trilean.exhaust)
next
case "3_2"
  then show ?case
  by (metis plus_trilean.simps(3) plus_trilean.simps(5) plus_trilean.simps(6) plus_trilean.simps(7)
trilean.exhaust)
next
case 4
  then show ?case
  by (metis plus_trilean.simps(2) plus_trilean.simps(3) plus_trilean.simps(4) plus_trilean.simps(5)
plus_trilean.simps(6) trilean.exhaust)
next
case 5
  then show ?case
  by (metis plus_trilean.simps(6) plus_trilean.simps(7) trilean.exhaust)
qed

lemma plus_trilean_commutative: "a  $\vee$ ? b = b  $\vee$ ? a"
proof(induct a b rule: plus_trilean.induct)
case (1 uu)
  then show ?case
  by (metis plus_trilean.simps(1) plus_trilean.simps(2) plus_trilean.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 times_trilean_commutative: "a  $\wedge$ ? b = b  $\wedge$ ? a"
  by (metis (mono_tags) times_trilean.simps trilean.distinct(5) trilean.exhaust)

lemma times_trilean_assoc: "a  $\wedge$ ? b  $\wedge$ ? c = a  $\wedge$ ? (b  $\wedge$ ? c)"
proof(induct a b arbitrary: c rule: plus_trilean.induct)
  case (1 uu)
  then show ?case
  by (metis (mono_tags, lifting) times_trilean.simps(1) times_trilean_commutative)
next
case "2_1"
  then show ?case
  by (metis (mono_tags, lifting) times_trilean.simps(1) times_trilean_commutative)
next
case "2_2"
  then show ?case
  by (metis (mono_tags, lifting) times_trilean.simps(1) times_trilean_commutative)
next
case "3_1"
  then show ?case
  by (metis times_trilean.simps(1) times_trilean.simps(4) times_trilean.simps(5) trilean.exhaust)
next
case "3_2"
  then show ?case
  by (metis times_trilean.simps(1) times_trilean.simps(5) times_trilean.simps(6) times_trilean.simps(7)
trilean.exhaust)
next
case 4
  then show ?case
  by (metis times_trilean.simps(1) times_trilean.simps(4) times_trilean.simps(5) times_trilean.simps(7)
trilean.exhaust)
next
case 5
  then show ?case
  by (metis (full_types) times_trilean.simps(1) times_trilean.simps(6) times_trilean.simps(7) trilean.exhaust)
qed

lemma trilean_distributivity_1: "(a  $\vee$ ? b)  $\wedge$ ? c = a  $\wedge$ ? c  $\vee$ ? b  $\wedge$ ? c"
proof(induct a b rule: times_trilean.induct)
case (1 uu)
  then show ?case
  by (metis (mono_tags, lifting) plus_trilean.simps(1) plus_trilean_commutative times_trilean.simps(1)
times_trilean_commutative)
next

```

```

    case "2_1"
    then show ?case
      by (metis (mono_tags, lifting) plus_trilean.simps(1) times_trilean.simps(1) times_trilean_commutative)
next
    case "2_2"
    then show ?case
      by (metis (mono_tags, lifting) plus_trilean.simps(1) times_trilean.simps(1) times_trilean_commutative)
next
    case 3
    then show ?case
      apply simp
      by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(4) plus_trilean.simps(7)
times_trilean.simps(1) times_trilean.simps(4) times_trilean.simps(5) trilean.exhaust)
next
    case "4_1"
    then show ?case
      apply simp
      by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(5) plus_trilean.simps(7)
times_trilean.simps(1) times_trilean.simps(4) times_trilean.simps(5) times_trilean.simps(6) times_trilean.simps(7)
trilean.exhaust)
next
    case "4_2"
    then show ?case
      apply simp
      by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(7) times_trilean.simps(1)
times_trilean.simps(6) times_trilean.simps(7) trilean.exhaust)
next
    case 5
    then show ?case
      apply simp
      by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(6) plus_trilean.simps(7)
times_trilean.simps(1) times_trilean.simps(4) times_trilean.simps(5) times_trilean.simps(6) times_trilean.simps(7)
trilean.exhaust)
qed

instance
  apply standard
    apply (simp add: plus_trilean_assoc)
    apply (simp add: plus_trilean_commutative)
    apply (simp add: times_trilean_assoc)
    apply (simp add: trilean_distributivity_1)
  using times_trilean_commutative trilean_distributivity_1 by auto
end

lemma maybe_or_idempotent: "a  $\vee$ ? a = a"
  apply (cases a)
  by auto

lemma maybe_and_idempotent: "a  $\wedge$ ? a = a"
  apply (cases a)
  by auto

instantiation trilean :: ord begin
definition less_eq_trilean :: "trilean  $\Rightarrow$  trilean  $\Rightarrow$  bool" where
  "less_eq_trilean a b = (a + b = b)"

definition less_trilean :: "trilean  $\Rightarrow$  trilean  $\Rightarrow$  bool" where
  "less_trilean a b = (a  $\leq$  b  $\wedge$  a  $\neq$  b)"

declare less_trilean_def less_eq_trilean_def [simp]

```

```

instance
  by standard
end

instantiation trilean :: uminus begin
  fun maybe_not :: "trilean  $\Rightarrow$  trilean" ("¬? _" [60] 60) where
    "¬? true = false" |
    "¬? false = true" |
    "¬? invalid = invalid"

instance
  by standard
end

lemma maybe_and_one: "true  $\wedge?$  x = x"
  apply (cases x)
  by auto

lemma maybe_or_zero: "false  $\vee?$  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_negate_false: "(¬? x = false) = (x = true)"
  apply (cases x)
  by auto

lemma maybe_and_true: "(x  $\wedge?$  y = true) = (x = true  $\wedge$  y = true)"
  using times_trilean.elims by blast

lemma maybe_and_not_true: "(x  $\wedge?$  y  $\neq$  true) = (x  $\neq$  true  $\vee$  y  $\neq$  true)"
  by (simp add: maybe_and_true)

lemma maybe_and_valid: "x  $\wedge?$  y  $\neq$  invalid  $\implies$  x  $\neq$  invalid  $\wedge$  y  $\neq$  invalid"
  using times_trilean.elims by blast

lemma maybe_or_valid: "x  $\vee?$  y  $\neq$  invalid  $\implies$  x  $\neq$  invalid  $\wedge$  y  $\neq$  invalid"
  using plus_trilean.elims by blast

lemma maybe_or_false: "(x  $\vee?$  y = false) = (x = false  $\wedge$  y = false)"
  using plus_trilean.elims by blast

lemma maybe_not_invalid: "(¬? x = invalid) = (x = invalid)"
  by (metis maybe_double_negation maybe_not.simps(3))

lemma maybe_or_invalid: "(x  $\vee?$  y = invalid) = (x = invalid  $\vee$  y = invalid)"
  using plus_trilean.elims by blast

lemma maybe_and_invalid: "(x  $\wedge?$  y = invalid) = (x = invalid  $\vee$  y = invalid)"
  using times_trilean.elims by blast

lemma invalid_maybe_and: "invalid  $\wedge?$  x = invalid"
  using maybe_and_valid by blast

```

```

lemma maybe_not_eq: "(¬? x = ¬? y) = (x = y)"
  by (metis maybe_double_negation)

lemma de_morgans_1: "¬? (a ∨? b) = (¬? a) ∧? (¬? b)"
  by (metis (no_types, hide_lams) add.commute invalid_maybe_and maybe_and_idempotent maybe_and_one maybe_not.elim
  maybe_not.simps(1) maybe_not.simps(3) maybe_not_invalid maybe_or_zero plus_trilean.simps(1) plus_trilean.simps(4)
  times_trilean.simps(1) times_trilean_commutative trilean.exhaust trilean.simps(6))

lemma de_morgans_2: "¬? (a ∧? b) = (¬? a) ∨? (¬? b)"
  by (metis de_morgans_1 maybe_double_negation)

end
theory Value
imports Trilean
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"

lemma MaybeBoolInt_lt: "MaybeBoolInt (λx y. y < x) (Some (Num n')) r = false ⟹ ∃n. r = Some (Num
n) ∧ n' ≤ n"
proof(induct n')
  case (nonneg n)
  then show ?case
    using MaybeBoolInt.elims by force
next
  case (neg n)
  then show ?case
    using MaybeBoolInt.elims by force
qed

lemma MaybeBoolInt_not_num_1: "∀n. r ≠ Some (Num n) ⟹ MaybeBoolInt f n r = invalid"
  apply (cases r)
  apply simp
  apply (case_tac a)
  by auto

lemma MaybeBoolInt_not_invalid: "MaybeBoolInt f a b ≠ invalid = (∃n n'. a = Some (Num n) ∧ b = Some
(Num n'))"
  apply (cases a)
  apply simp
  apply (cases b)
  apply simp
  apply (case_tac aa)
  apply (case_tac aaa)
  by auto

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 " $x \leq x$ "
  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 " $x \leq y \implies y \leq z \implies x \leq z$ "
  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
  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
theory Option_Lexorder
imports Main
begin

instantiation option :: (linorder) linorder begin
fun less_option :: "'a option ⇒ 'a option ⇒ bool" where
  "None < None = False" |
  "None < _ = True" |
  "Some _ < None = False" |
  "Some a < Some b = (a < b)"

fun less_eq_option :: "'a option ⇒ 'a option ⇒ bool" where
  "less_eq_option a b = (a < b ∨ a = b)"

instance
  apply standard
  apply (metis less_eq_option.simps less_option.elims(2) dual_order.asym less_option.simps(3) option.inject)
  apply simp
  apply (case_tac x)
  apply (metis less_eq_option.simps less_option.elims(2) less_option.simps(2))
  apply simp
  apply (case_tac y)
  apply simp
  apply (case_tac z)
  apply simp
  apply auto[1]
  apply (metis less_eq_option.elims(2) less_option.elims(2) dual_order.asym option.discI option.inject
option.simps(3))
  by (metis less_option.elims(3) less_eq_option.elims(3) less_option.simps(2) neqE option.inject)
end
declare less_eq_option.simps [simp del]

```



```

lemma max_None_l: "max None x = x"
  by (metis less_option.elims(2) max.absorb2 not_le option.simps(3))

lemma max_None_r: "max x None = x"
  by (simp add: max commute max_None_l)

lemmas max_None = max_None_l max_None_r

lemma max_is_None: "(max x y = None) = (x = None ∧ y = None)"
  by (metis max.left_idem max_None_l max_None_r)

lemma max_Some_Some: "max (Some x) (Some y) = Some (max x y)"
  by (metis less_option.simps(4) max_def not_le)

lemma x_leq_None: "(x ≤ None) = (x = None)"
  by (meson eq_refl max_def max_is_None)

lemma None_leq_everything: "None ≤ x"
  by (metis linear x_leq_None)

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 FinFun.FinFun Option_Lexorder
begin

declare One_nat_def [simp del]
unbundle finfun_syntax

type_synonym registers = "nat ⇒ f value option"
type_synonym datastate = "vname ⇒ value option"
datatype aexp = L "value" | V vname | Plus aexp aexp | Minus aexp aexp

fun MaybeArithInt :: "(int ⇒ int ⇒ int) ⇒ value option ⇒ value option ⇒ value option" where
  "MaybeArithInt f (Some (Num x)) (Some (Num y)) = Some (Num (f x y))" |
  "MaybeArithInt _ _ _ = None"

lemma MaybeArithInt_not_None: "MaybeArithInt f a b ≠ None = (∃ n n'. a = Some (Num n) ∧ b = Some (Num n'))"
  apply (cases a)
  apply simp

```

```

    apply (cases b)
    apply simp
    apply (case_tac aa)
    apply (metis MaybeArithInt.elims MaybeArithInt.simps(1) option.distinct(1))
    apply (case_tac aaa)
    apply simp
  by simp

lemma MaybeArithInt_Some: "MaybeArithInt f a b = Some (Num x) = ( $\exists$  n n'. a = Some (Num n)  $\wedge$  b = Some (Num n')  $\wedge$  f n n' = x)"
  apply (cases a)
  apply simp
  apply (cases b)
  apply simp
  apply (case_tac aa)
  apply simp
  apply (metis MaybeArithInt.simps(1) MaybeArithInt_not_None option.inject option.simps(3) value.inject(1))
  apply (case_tac aaa)
  by auto

lemma MaybeArithInt_None: "(MaybeArithInt f a1 a2 = None) = ( $\nexists$  n n'. a1 = Some (Num n)  $\wedge$  a2 = Some (Num n'))"
  using MaybeArithInt_not_None by blast

lemma MaybeArithInt_Not_Num: "( $\forall$  n. MaybeArithInt f a1 a2  $\neq$  Some (Num n)) = (MaybeArithInt f a1 a2 = None)"
  by (metis MaybeArithInt.elims option.distinct(1))

definition "value_plus = MaybeArithInt (+)"

lemma plus_no_string [simp]: "MaybeArithInt f a b  $\neq$  Some (Str x)"
  using MaybeArithInt.elims by blast

lemma value_plus_symmetry: "value_plus x y = value_plus y x"
  apply (induct x y rule: MaybeArithInt.induct)
  by (simp_all add: value_plus_def)

definition "value_minus = MaybeArithInt (-)"

lemma minus_no_string [simp]: "value_minus a b  $\neq$  Some (Str x)"
  by (simp add: value_minus_def)

fun aval :: "aexp  $\Rightarrow$  datastate  $\Rightarrow$  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)

abbreviation null_state ("<>") where
  "null_state  $\equiv$  (K$ None)"

nonterminal maplets and maplet

syntax
  "_maplet"  :: "[ 'a, 'a ]  $\Rightarrow$  maplet"          ("_ /:=/_")
  "_maplets" :: "[ 'a, 'a ]  $\Rightarrow$  maplet"          ("_ /[:=]/ _")
  ""         :: "maplet  $\Rightarrow$  maplets"            ("_")
  "_Maplets" :: "[maplet, maplets]  $\Rightarrow$  maplets" ("_,/_")

```

```

_MapUpd"  :: "[a → b, maplets] ⇒ a → b" ("_/'(_)" [900, 0] 900)
_Map"     :: "maplets ⇒ a → b"          ("(1[_])")
_Map"     :: "maplets ⇒ a → b"          ("(1<_>)")

translations
_MapUpd m (_Maplets xy ms)  ⇒ "_MapUpd (_MapUpd m xy) ms"
_MapUpd m (_maplet x y)    ⇒ "m(x $:= CONST Some y)"
_Map ms                    ⇒ "_MapUpd (CONST empty) ms"
_Map (_Maplets ms1 ms2)    ← "_MapUpd (_Map ms1) ms2"
_Maplets ms1 (_Maplets ms2 ms3) ← "_Maplets (_Maplets ms1 ms2) ms3"

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

definition input2state :: "value list ⇒ registers" where
  "input2state n = fold (λ(k, v) f. f(k $:= Some v)) (enumerate 0 n) (K$ None)"

primrec input2state_prim :: "value list ⇒ nat ⇒ registers" where
  "input2state_prim [] _ = (K$ None)" |
  "input2state_prim (v#t) k = (input2state_prim t (k+1))(k $:= Some v)"

lemma input2state_append: "input2state (i @ [a]) = (input2state i)(length i $:= Some a)"
  apply (simp add: eq_funfun_All_ext finfun_All_def finfun_All_except_def)
  apply clarify
  by (simp add: input2state_def enumerate_eq_zip)

lemma fold_conv_foldr: "fold f xs = foldr f (rev xs)"
  by (simp add: foldr_conv_fold)

lemma length_map: "length i > 0 ⇒ [Suc 0..

```

```

lemma input2state_empty: "input2state [] $ x1 = None"
  by (simp add: input2state_out_of_bounds)

lemma input2state_nth: "i < length ia  $\implies$  input2state ia $ i = Some (ia ! i)"
proof(induct ia rule: rev_induct)
  case Nil
  then show ?case
    by simp
next
  case (snoc a ia)
  then show ?case
    apply (simp add: input2state_def enumerate_eq_zip)
    by (simp add: finfun_upd_apply nth_append)
qed

lemma input2state_not_None: "(input2state i $ x  $\neq$  None)  $\implies$  (x < length i)"
  using input2state_within_bounds by blast

lemma input2state_not_None_equiv: "(input2state i $ x  $\neq$  None) = (x < length i)"
  apply standard
  using input2state_within_bounds apply blast
  by (simp add: input2state_nth)

lemma input2state_Some: "(\exists v. input2state i $ x = Some v) = (x < length i)"
  apply standard
  using input2state_within_bounds apply blast
  by (simp add: input2state_nth)

lemma input2state_cons:
  "x1 > 0  $\implies$ 
  x1 < length ia  $\implies$ 
  input2state (a # ia) $ x1 = input2state ia $ (x1-1)"
  by (simp add: input2state_nth)

lemma input2state_cons_shift: "input2state i $ x1 = Some a  $\implies$  input2state (b # i) $ (Suc x1) = Some a"
proof(induct i rule: rev_induct)
  case Nil
  then show ?case
    by (simp add: input2state_def)
next
  case (snoc x xs)
  then show ?case
    using input2state_within_bounds[of xs x1 a]
    using input2state_cons[of "Suc x1" "xs @ [x]" b]
    apply simp
    apply (case_tac "x1 < length xs")
    apply simp
    by (metis finfun_upd_apply input2state_append input2state_nth length_Cons length_append_singleton
    lessI list.sel(3) nth_tl)
qed

lemma input2state_exists: "\exists i. input2state i $ x1 = Some a"
proof(induct x1)
  case 0
  then show ?case
    apply (rule_tac x="[a]" in exI)
    by (simp add: input2state_def)
next
  case (Suc x1)
  then show ?case

```

```

    apply clarify
    apply (rule_tac x="a#i" in exI)
    by (simp add: input2state_cons_shift)
qed

```

```

primrec repeat :: "nat  $\Rightarrow$  'a  $\Rightarrow$  'a list" where
  "repeat 0 _ = []" |
  "repeat (Suc m) a = a#(repeat m a)"

```

```

lemma length_repeat: "length (repeat n a) = n"
proof(induct n)
  case 0
  then show ?case
    by simp
next
  case (Suc a)
  then show ?case
    by simp
qed

```

```

lemma length_append_repeat: "length (i@(repeat a y))  $\geq$  length i"
  by simp

```

```

lemma length_input2state_repeat: "input2state i $ x = Some a  $\implies$  y < length (i @ repeat y a)"
  by (metis append.simps(1) append_eq_append_conv input2state_within_bounds length_append length_repeat
  list.size(3) neqE not_add_less2 zero_order(3))

```

```

lemma input2state_double_exists: " $\exists$ i. input2state i $ x = Some a  $\wedge$  input2state i $ y = Some a"
  apply (insert input2state_exists[of x a])
  apply clarify
  apply (case_tac "x  $\geq$  y")
  apply (rule_tac x="list_update i y a" in exI)
  apply (metis (no_types, lifting) input2state_within_bounds input2state_nth input2state_out_of_bounds
  le_trans length_list_update not_le_imp_less nth_list_update_eq nth_list_update_neq)
  apply (rule_tac x="list_update (i@(repeat y a)) y a" in exI)
  apply (simp add: not_le)
  by (metis length_input2state_repeat input2state_nth input2state_out_of_bounds le_trans length_append_repeat
  length_list_update not_le_imp_less nth_append nth_list_update_eq nth_list_update_neq option.distinct(1))

```

```

lemma input2state_double_exists_2: "x  $\neq$  y  $\implies$   $\exists$ i. input2state i $ x = Some a  $\wedge$  input2state i $ y
= Some a'"
  apply (insert input2state_exists[of x a])
  apply clarify
  apply (case_tac "x  $\geq$  y")
  apply (rule_tac x="list_update i y a'" in exI)
  apply (metis (no_types, lifting) input2state_within_bounds input2state_nth input2state_out_of_bounds
  le_trans length_list_update not_le_imp_less nth_list_update_eq nth_list_update_neq)
  apply (rule_tac x="list_update (i@(repeat y a')) y a'" in exI)
  apply (simp add: not_le)
  apply standard
  apply (metis input2state_nth input2state_within_bounds le_trans length_append_repeat length_list_update
  linorder_not_le nth_append nth_list_update_neq order_refl)
  by (metis input2state_nth length_append length_input2state_repeat length_list_update length_repeat
  nth_list_update_eq)

```

```

definition join_ir :: "value list  $\Rightarrow$  registers  $\Rightarrow$  datastate" where
  "join_ir i r  $\equiv$  ( $\lambda$ x. case x of
    R n  $\Rightarrow$  r $ n |
    I n  $\Rightarrow$  (input2state i) $ n
  )"

```

```

lemmas datastate = join_ir_def input2state_def

lemma join_ir_empty[simp]: "join_ir [] <> = (λx. None)"
  apply (rule ext)
  apply (simp add: join_ir_def)
  apply (case_tac x)
  apply (simp add: input2state_def)
  by simp

lemma join_ir_double_exists: "∃ i r. join_ir i r v = Some a ∧ join_ir i r v' = Some a"
proof(cases v)
  case (I x1)
  then show ?thesis
    apply (simp add: join_ir_def)
    apply (cases v')
    apply (simp add: input2state_double_exists input2state_exists)
    using input2state_exists by auto
next
  case (R x2)
  then show ?thesis
    apply (simp add: join_ir_def)
    apply (cases v')
    using input2state_exists apply force
    using input2state_double_exists by fastforce
qed

lemma join_ir_double_exists_2: "v ≠ v' ⇒ ∃ i r. join_ir i r v = Some a ∧ join_ir i r v' = Some a'"
proof(cases v)
  case (I x1)
  assume "v ≠ v'"
  then show ?thesis
    apply (simp add: join_ir_def)
    apply (cases v')
    apply (simp add: I input2state_double_exists_2)
    using I input2state_exists by auto
next
  case (R x2)
  assume "v ≠ v'"
  then show ?thesis
    apply (simp add: join_ir_def)
    apply (cases v')
    apply simp
    using R input2state_exists apply auto[1]
    apply (simp add: R)
    apply (rule_tac x="(K$ None)(x2 $:= Some a)(x2a $:= Some a')" in exI)
    by simp
qed

lemma aval_plus_aexp: "aval (a+b) s = aval (Plus a b) s"
  apply(induct a b rule: plus_aexp.induct)
  by (simp_all add: value_plus_def)

lemma aval_minus_aexp: "aval (a-b) s = aval (Minus a b) s"
  apply(induct a b rule: minus_aexp.induct)
  by (simp_all add: value_minus_def)

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"

fun enumerate_aexp_inputs :: "aexp ⇒ nat set" where
  "enumerate_aexp_inputs (L _) = {}" |
  "enumerate_aexp_inputs (V (I n)) = {n}" |
  "enumerate_aexp_inputs (V (R n)) = {}" |
  "enumerate_aexp_inputs (Plus v va) = enumerate_aexp_inputs v ∪ enumerate_aexp_inputs va" |
  "enumerate_aexp_inputs (Minus v va) = enumerate_aexp_inputs v ∪ enumerate_aexp_inputs va"

fun enumerate_aexp_inputs_list :: "aexp ⇒ nat list" where
  "enumerate_aexp_inputs_list (L _) = []" |
  "enumerate_aexp_inputs_list (V (I n)) = [n]" |
  "enumerate_aexp_inputs_list (V (R n)) = []" |
  "enumerate_aexp_inputs_list (Plus v va) = enumerate_aexp_inputs_list v @ enumerate_aexp_inputs_list va" |
  "enumerate_aexp_inputs_list (Minus v va) = enumerate_aexp_inputs_list v @ enumerate_aexp_inputs_list va"

fun enumerate_aexp_regs :: "aexp ⇒ nat set" where
  "enumerate_aexp_regs (L _) = {}" |
  "enumerate_aexp_regs (V (R n)) = {n}" |
  "enumerate_aexp_regs (V (I _)) = {}" |
  "enumerate_aexp_regs (Plus v va) = enumerate_aexp_regs v ∪ enumerate_aexp_regs va" |
  "enumerate_aexp_regs (Minus v va) = enumerate_aexp_regs v ∪ enumerate_aexp_regs va"

fun enumerate_aexp_regs_list :: "aexp ⇒ nat list" where
  "enumerate_aexp_regs_list (L _) = []" |
  "enumerate_aexp_regs_list (V (R n)) = [n]" |
  "enumerate_aexp_regs_list (V (I _)) = []" |
  "enumerate_aexp_regs_list (Plus v va) = enumerate_aexp_regs_list v @ enumerate_aexp_regs_list va" |
  "enumerate_aexp_regs_list (Minus v va) = enumerate_aexp_regs_list v @ enumerate_aexp_regs_list va"

lemma enumerate_aexp_inputs_list: "enumerate_aexp_inputs l = set (enumerate_aexp_inputs_list l)"
proof(induct l)
case (L x)
  then show ?case
  by simp
next
case (V x)
  then show ?case
  apply (cases x)
  by auto
next
case (Plus l1 l2)
  then show ?case
  by simp
next
case (Minus l1 l2)

```

```

    then show ?case
      by simp
qed

definition max_input :: "aexp  $\Rightarrow$  nat option" where
  "max_input g = (let inputs = (enumerate_aexp_inputs g) in if inputs = {} then None else Some (Max inputs))"

lemma enumerate_aexp_regs_list: "set (enumerate_aexp_regs_list l) = enumerate_aexp_regs l"
proof(induct l)
case (L x)
then show ?case
  by simp
next
case (V x)
then show ?case
  apply (cases x)
  by auto
next
case (Plus l1 l2)
then show ?case
  by simp
next
case (Minus l1 l2)
then show ?case
  by simp
qed

definition max_reg :: "aexp  $\Rightarrow$  nat option" where
  "max_reg g = (let regs = (enumerate_aexp_regs g) in if regs = {} then None else Some (Max regs))"

lemma max_reg_V_I: "max_reg (V (I n)) = None"
  by (simp add: max_reg_def)

lemma max_reg_V_R: "max_reg (V (R n)) = Some n"
  by (simp add: max_reg_def)

lemmas max_reg_V = max_reg_V_I max_reg_V_R

lemma max_reg_Plus: "max_reg (Plus a1 a2) = max (max_reg a1) (max_reg a2)"
  apply (simp add: max_reg_def Let_def max_None max_Some_Some)
  by (metis List.finite_set Max.union enumerate_aexp_regs_list)

lemma max_reg_Minus: "max_reg (Minus a1 a2) = max (max_reg a1) (max_reg a2)"
  apply (simp add: max_reg_def Let_def max_None max_Some_Some)
  by (metis List.finite_set Max.union enumerate_aexp_regs_list)

lemma no_reg_aval_swap_regs: "AExp.max_reg a = None  $\implies$  aval a (join_ir i r) = aval a (join_ir i r')"
proof(induct a)
case (L x)
then show ?case
  by simp
next
case (V x)
then show ?case
  apply (cases x)
  apply (simp add: join_ir_def)
  apply (simp add: join_ir_def)
  by (simp add: max_reg_def)
next
case (Plus a1 a2)
then show ?case

```



```

      by (simp add: max_reg_Plus max_is_None)
next
  case (Minus a1 a2)
  then show ?case
    by (simp add: max_reg_Minus max_is_None)
qed

```

```

lemma enumerate_aexp_regs_empty_reg_unconstrained:
  "enumerate_aexp_regs a = {}  $\implies \forall r. \neg \text{aexp\_constrains } a \text{ (V (R r))}"
proof(induct a)
case (L x)
  then show ?case
    by simp
next
  case (V x)
  then show ?case
    apply (cases x)
    apply simp
    by simp
next
  case (Plus a1 a2)
  then show ?case
    by simp
next
  case (Minus a1 a2)
  then show ?case
    by simp
qed$ 
```

```

lemma set_enumerate_aexp_inputs_list:
  "set (fold (@) (map enumerate_aexp_inputs_list l) []) = ( $\bigcup$  (set (map enumerate_aexp_inputs l)))"
proof(induct l)
case Nil
  then show ?case
    by simp
next
case (Cons a l)
  then show ?case
    using enumerate_aexp_inputs_list
    by (simp add: fold_append_concat_rev inf_sup_aci(5))
qed

```

```

lemma enumerate_aexp_inputs_empty_input_unconstrained:
  "enumerate_aexp_inputs a = {}  $\implies \forall r. \neg \text{aexp\_constrains } a \text{ (V (I r))}"
proof(induct a)
case (L x)
  then show ?case
    by simp
next
  case (V x)
  then show ?case
    apply (cases x)
    apply simp
    by simp
next
  case (Plus a1 a2)
  then show ?case
    by simp
next
  case (Minus a1 a2)$ 
```

```

    then show ?case
    by simp
qed

```

lemma input_unconstrained_aval_input_swap:

```

  "∀ i. ¬ aexp_constrains a (V (I i)) ⇒ aval a (join_ir i r) = aval a (join_ir i' r)"
proof(induct a)
case (L x)
  then show ?case
  by simp
next
  case (V x)
  then show ?case
  apply (cases x)
  apply simp
  by (simp add: join_ir_def)
next
  case (Plus a1 a2)
  then show ?case
  by simp
next
  case (Minus a1 a2)
  then show ?case
  by simp
qed

```

lemma input_unconstrained_aval_register_swap:

```

  "∀ i. ¬ aexp_constrains a (V (R i)) ⇒ aval a (join_ir i r) = aval a (join_ir i r'"
proof(induct a)
case (L x)
  then show ?case
  by simp
next
  case (V x)
  then show ?case
  apply (cases x)
  apply (simp add: join_ir_def)
  by simp
next
  case (Plus a1 a2)
  then show ?case
  by simp
next
  case (Minus a1 a2)
  then show ?case
  by simp
qed

```

lemma unconstrained_variable_swap_aval:

```

  "∀ i. ¬ aexp_constrains a (V (I i)) ⇒
  ∀ r. ¬ aexp_constrains a (V (R r)) ⇒
  aval a s = aval a s'"
proof(induct a)
case (L x)
  then show ?case
  by simp
next
  case (V x)
  then show ?case
  apply (cases x)
  by auto

```

```

next
  case (Plus a1 a2)
  then show ?case
  by simp
next
  case (Minus a1 a2)
  then show ?case
  by simp
qed

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 Trilean Option_Lexorder
begin

definition I :: "nat  $\Rightarrow$  vname" where
  "I n = vname.I (n-1)"
declare I_def [simp]
hide_const I
datatype gexp = Bc bool | Eq aexp aexp | Gt aexp aexp | Nor gexp gexp | Null aexp

syntax (xsymbols)
  Eq :: "aexp  $\Rightarrow$  aexp  $\Rightarrow$  gexp"
  Gt :: "aexp  $\Rightarrow$  aexp  $\Rightarrow$  gexp"

fun gval :: "gexp  $\Rightarrow$  datastate  $\Rightarrow$  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 =  $\neg$ ? ((gval a1 s)  $\vee$ ? (gval a2 s))" |
  "gval (Null v) s = ValueEq (aval v s) None"

abbreviation gNot :: "gexp  $\Rightarrow$  gexp" where
  "gNot g  $\equiv$  Nor g g"

abbreviation gOr :: "gexp  $\Rightarrow$  gexp  $\Rightarrow$  gexp" where
  "gOr v va  $\equiv$  Nor (Nor v va) (Nor v va)"

abbreviation gAnd :: "gexp  $\Rightarrow$  gexp  $\Rightarrow$  gexp" where
  "gAnd v va  $\equiv$  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'  $\wedge$  y = y')"
proof

```

```

    show "gAnd x y = gAnd x' y'  $\implies$  x = x'  $\wedge$  y = y'"
    by (simp)
next
    show "x = x'  $\wedge$  y = y'  $\implies$  gAnd x y = gAnd x' y' "
    by simp
qed

abbreviation Lt :: "aexp  $\Rightarrow$  aexp  $\Rightarrow$  gexp" where
  "Lt a b  $\equiv$  Gt b a"

abbreviation Le :: "aexp  $\Rightarrow$  aexp  $\Rightarrow$  gexp" where
  "Le v va  $\equiv$  gNot (Gt v va)"

abbreviation Ge :: "aexp  $\Rightarrow$  aexp  $\Rightarrow$  gexp" where
  "Ge v va  $\equiv$  gNot (Lt v va)"

abbreviation Ne :: "aexp  $\Rightarrow$  aexp  $\Rightarrow$  gexp" where
  "Ne v va  $\equiv$  gNot (Eq v va)"

fun In :: "vname  $\Rightarrow$  value list  $\Rightarrow$  gexp" where
  "In a [] = Bc False" |
  "In a [v] = Eq (V a) (L v)" |
  "In a (v1#t) = gOr (Eq (V a) (L v1)) (In a t)"

lemma gval_gOr: "gval (gOr x y) r = (gval x r)  $\vee$ ? (gval y r)"
  by (simp add: maybe_double_negation maybe_or_idempotent)

lemma gval_gNot: "gval (gNot x) s =  $\neg$ ? (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 gval_gNot)

definition satisfiable :: "gexp  $\Rightarrow$  bool" where
  "satisfiable g  $\equiv$  ( $\exists$  i r. gval g (join_ir i r) = true)"

definition "satisfiable_list l = satisfiable (fold gAnd l (Bc True))"

lemma satisfiable_true: "satisfiable (Bc True)"
  by (simp add: satisfiable_def)

definition valid :: "gexp  $\Rightarrow$  bool" where
  "valid g  $\equiv$  ( $\forall$  i r. gval g (join_ir i r) = true)"

definition "valid_list l = valid (fold gAnd l (Bc True))"

definition gexp_equiv :: "gexp  $\Rightarrow$  gexp  $\Rightarrow$  bool" where
  "gexp_equiv a b  $\equiv$   $\forall$  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  $\implies$  gexp_equiv y x"
  by (simp add: gexp_equiv_def)

lemma gexp_equiv_transitive: "gexp_equiv x y  $\wedge$  gexp_equiv y z  $\implies$  gexp_equiv x z"
  by (simp add: gexp_equiv_def)

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

```

```

lemma gexp_equiv_satisfiable: "gexp_equiv x y  $\implies$  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: add.commute gexp_equiv_def)

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

lemma gval_gAnd: "gval (gAnd g1 g2) s = (gval g1 s)  $\wedge$ ? (gval g2 s)"
proof(induct "gval g1 s" "gval g2 s" rule: times_trilean.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)  $\wedge$  gval g2 s = true)"
  using gval_gAnd maybe_and_not_true by fastforce

declare gval.simps [simp del]

fun gexp_constrains :: "gexp  $\Rightarrow$  aexp  $\Rightarrow$  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  $\vee$  aexp_constrains a2 v)" |
  "gexp_constrains (gexp.Gt a1 a2) v = (aexp_constrains a1 v  $\vee$  aexp_constrains a2 v)" |
  "gexp_constrains (gexp.Nor g1 g2) v = (gexp_constrains g1 v  $\vee$  gexp_constrains g2 v)"

```

```

fun contains_bool :: "gexp  $\Rightarrow$  bool" where
  "contains_bool (Bc _) = True" |
  "contains_bool (Nor g1 g2) = (contains_bool g1  $\vee$  contains_bool g2)" |
  "contains_bool _ = False"

fun gexp_same_structure :: "gexp  $\Rightarrow$  gexp  $\Rightarrow$  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'  $\wedge$  aexp_same_structure a2 a2')" |
  "gexp_same_structure (gexp.Gt a1 a2) (gexp.Gt a1' a2') = (aexp_same_structure a1 a1'  $\wedge$  aexp_same_structure a2 a2')" |
  "gexp_same_structure (gexp.Nor g1 g2) (gexp.Nor g1' g2') = (gexp_same_structure g1 g1'  $\wedge$  gexp_same_structure g2 g2')" |
  "gexp_same_structure (gexp.Null a1) (gexp.Null a2) = aexp_same_structure a1 a2" |
  "gexp_same_structure _ _ = False"

lemma gval_foldr_true: "(gval (foldr gAnd G (Bc True)) s = true) = ( $\forall g \in \text{set } G. \text{gval } g \text{ } s = \text{true}$ )"
proof(induct G)
  case Nil
  then show ?case
    by (simp add: gval.simps)
next
  case (Cons a G)
  then show ?case
    apply (simp only: foldr.simps comp_def gval_gAnd maybe_and_true)
    by simp
qed

fun enumerate_gexp_inputs :: "gexp  $\Rightarrow$  nat set" where
  "enumerate_gexp_inputs (GExp.Bc _) = {}" |
  "enumerate_gexp_inputs (GExp.Null v) = enumerate_aexp_inputs v" |
  "enumerate_gexp_inputs (GExp.Eq v va) = enumerate_aexp_inputs v  $\cup$  enumerate_aexp_inputs va" |
  "enumerate_gexp_inputs (GExp.Lt v va) = enumerate_aexp_inputs v  $\cup$  enumerate_aexp_inputs va" |
  "enumerate_gexp_inputs (GExp.Nor v va) = enumerate_gexp_inputs v  $\cup$  enumerate_gexp_inputs va"

fun enumerate_gexp_inputs_list :: "gexp  $\Rightarrow$  nat list" where
  "enumerate_gexp_inputs_list (GExp.Bc _) = []" |
  "enumerate_gexp_inputs_list (GExp.Null v) = enumerate_aexp_inputs_list v" |
  "enumerate_gexp_inputs_list (GExp.Eq v va) = enumerate_aexp_inputs_list v @ enumerate_aexp_inputs_list va" |
  "enumerate_gexp_inputs_list (GExp.Lt v va) = enumerate_aexp_inputs_list v @ enumerate_aexp_inputs_list va" |
  "enumerate_gexp_inputs_list (GExp.Nor v va) = enumerate_gexp_inputs_list v @ enumerate_gexp_inputs_list va"

lemma enumerate_gexp_inputs_list: "enumerate_gexp_inputs l = set (enumerate_gexp_inputs_list l)"
proof(induct l)
  case (Bc x)
  then show ?case
    by simp
next
  case (Eq x1a x2)
  then show ?case
    by (simp add: enumerate_aexp_inputs_list)
next
  case (Gt x1a x2)
  then show ?case
    by (simp add: enumerate_aexp_inputs_list)
next
  case (Nor l1 l2)
  then show ?case

```

```

    by simp
next
  case (Null x)
  then show ?case
    by (simp add: enumerate_aexp_inputs_list)
qed

lemma set_enumerate_gexp_inputs_list:
  "set (fold (@) (map enumerate_gexp_inputs_list l) []) = (⋃ (set (map enumerate_gexp_inputs l)))"
proof(induct l)
case Nil
  then show ?case
    by simp
next
case (Cons a l)
  then show ?case
    using enumerate_gexp_inputs_list
    by (simp add: fold_append_concat_rev inf_sup_aci(5))
qed

definition max_input :: "gexp ⇒ nat option" where
  "max_input g = (let inputs = (enumerate_gexp_inputs g) in if inputs = {} then None else Some (Max inputs))"

fun enumerate_gexp_regs :: "gexp ⇒ nat set" where
  "enumerate_gexp_regs (GExp.Bc _) = {}" |
  "enumerate_gexp_regs (GExp.Null v) = enumerate_aexp_regs v" |
  "enumerate_gexp_regs (GExp.Eq v va) = enumerate_aexp_regs v ∪ enumerate_aexp_regs va" |
  "enumerate_gexp_regs (GExp.Lt v va) = enumerate_aexp_regs v ∪ enumerate_aexp_regs va" |
  "enumerate_gexp_regs (GExp.Nor v va) = enumerate_gexp_regs v ∪ enumerate_gexp_regs va"

fun enumerate_gexp_regs_list :: "gexp ⇒ nat list" where
  "enumerate_gexp_regs_list (GExp.Bc _) = []" |
  "enumerate_gexp_regs_list (GExp.Null v) = enumerate_aexp_regs_list v" |
  "enumerate_gexp_regs_list (GExp.Eq v va) = enumerate_aexp_regs_list v @ enumerate_aexp_regs_list va"
|
  "enumerate_gexp_regs_list (GExp.Lt v va) = enumerate_aexp_regs_list v @ enumerate_aexp_regs_list va"
|
  "enumerate_gexp_regs_list (GExp.Nor v va) = enumerate_gexp_regs_list v @ enumerate_gexp_regs_list va"

lemma enumerate_gexp_regs_list: "set (enumerate_gexp_regs_list l) = enumerate_gexp_regs l"
proof(induct l)
case (Bc x)
  then show ?case
    by simp
next
case (Eq x1a x2)
  then show ?case
    by (simp add: enumerate_aexp_regs_list)
next
case (Lt x1a x2)
  then show ?case
    by (simp add: enumerate_aexp_regs_list)
next
case (Nor l1 l2)
  then show ?case
    by simp
next
case (Null x)
  then show ?case
    by (simp add: enumerate_aexp_regs_list)
qed

```

```

definition max_reg :: "gexp  $\Rightarrow$  nat option" where
  "max_reg g = (let regs = (enumerate_gexp_regs g) in if regs = {} then None else Some (Max regs))"

lemma max_reg_gNot: "max_reg (gNot x) = max_reg x"
  by (simp add: max_reg_def)

lemma max_reg_Eq: "max_reg (Eq a b) = max (AExp.max_reg a) (AExp.max_reg b)"
  apply (simp add: max_reg_def AExp.max_reg_def Let_def max_None max_Some_Some)
  by (metis List.finite_set Max.union enumerate_aexp_regs_list)

lemma max_reg_Gt: "max_reg (Gt a b) = max (AExp.max_reg a) (AExp.max_reg b)"
  apply (simp add: max_reg_def AExp.max_reg_def Let_def max_None max_Some_Some)
  by (metis List.finite_set Max.union enumerate_aexp_regs_list max.commute)

lemma max_reg_Nor: "max_reg (Nor a b) = max (max_reg a) (max_reg b)"
  apply (simp add: max_reg_def Let_def max_None max_Some_Some)
  by (metis List.finite_set Max.union enumerate_gexp_regs_list)

lemma max_reg_Null: "max_reg (Null a) = AExp.max_reg a"
  by (simp add: AExp.max_reg_def GExp.max_reg_def)

lemma no_reg_gval_swap_regs: "GExp.max_reg g = None  $\implies$  gval g (join_ir i r) = gval g (join_ir i r')"
proof(induct g)
case (Bc x)
  then show ?case
  by (metis gval.simps(1) gval.simps(2))
next
case (Eq x1a x2)
  then show ?case
  by (metis gval.simps(4) max_is_None max_reg_Eq no_reg_aval_swap_regs)
next
case (Gt x1a x2)
  then show ?case
  by (metis gval.simps(3) max_is_None max_reg_Gt no_reg_aval_swap_regs)
next
case (Nor g1 g2)
  then show ?case
  by (simp add: gval.simps(5) max_is_None max_reg_Nor)
next
case (Null x)
  then show ?case
  by (metis gval.simps(6) max_reg_Null no_reg_aval_swap_regs)
qed

lemma enumerate_gexp_regs_empty_reg_unconstrained: "enumerate_gexp_regs g = {}  $\implies$   $\forall r. \neg$  gexp_constrains
g (V (R r))"
proof(induct g)
case (Bc x)
  then show ?case
  by simp
next
case (Eq x1a x2)
  then show ?case
  by (simp add: enumerate_aexp_regs_empty_reg_unconstrained)
next
case (Gt x1a x2)
  then show ?case
  by (simp add: enumerate_aexp_regs_empty_reg_unconstrained)
next
case (Nor g1 g2)

```



```

    then show ?case
    by simp
next
  case (Null x)
  then show ?case
  by (simp add: enumerate_aexp_regs_empty_reg_unconstrained)
qed

```

```

lemma enumerate_gexp_inputs_empty_input_unconstrained: "enumerate_gexp_inputs g = {}  $\implies \forall r. \neg \text{gexp\_constrains } g \ (V \ (vname.I \ r))"$ 

```

```

proof(induct g)
case (Bc x)
  then show ?case
  by simp
next
  case (Eq x1a x2)
  then show ?case
  by (simp add: enumerate_aexp_inputs_empty_input_unconstrained)
next
  case (Gt x1a x2)
  then show ?case
  by (simp add: enumerate_aexp_inputs_empty_input_unconstrained)
next
  case (Nor g1 g2)
  then show ?case
  by simp
next
  case (Null x)
  then show ?case
  by (simp add: enumerate_aexp_inputs_empty_input_unconstrained)
qed

```

```

lemma input_unconstrained_gval_input_swap: " $\forall r. \neg \text{gexp\_constrains } a \ (V \ (vname.I \ r)) \implies (\text{gval } a \ (\text{join\_ir } i \ r) = \text{gval } a \ (\text{join\_ir } i' \ r))"$ 

```

```

proof(induct a)
  case (Bc x)
  then show ?case
  apply (cases x)
  apply (simp add: gval.simps(1))
  by (simp add: gval.simps(2))
next
  case (Eq x1a x2)
  then show ?case
  apply (simp add: gval.simps ValueEq_def)
  by (metis input_unconstrained_aval_input_swap)
next
  case (Gt x1a x2)
  then show ?case
  apply (simp add: gval.simps ValueGt_def)
  by (metis input_unconstrained_aval_input_swap)
next
  case (Nor a1 a2)
  then show ?case
  by (simp add: gval.simps(5))
next
  case (Null x)
  then show ?case
  by (metis gexp_constrains.simps(2) gval.simps(6) input_unconstrained_aval_input_swap)
qed

```

```

lemma register_unconstrained_gval_register_swap: " $\forall r. \neg \text{gexp\_constrains } a \ (V \ (R \ r)) \implies (\text{gval } a \ (\text{join\_ir } i \ r) = \text{gval } a \ (\text{join\_ir } i' \ r))"$ 

```

```

i r) = gval a (join_ir i r'))"
proof(induct a)
  case (Bc x)
  then show ?case
    apply (cases x)
    apply (simp add: gval.simps(1))
    by (simp add: gval.simps(2))
next
  case (Eq x1a x2)
  then show ?case
    apply (simp add: gval.simps ValueEq_def)
    by (metis input_unconstrained_aval_register_swap)
next
  case (Gt x1a x2)
  then show ?case
    apply (simp add: gval.simps ValueGt_def)
    by (metis input_unconstrained_aval_register_swap)
next
  case (Nor a1 a2)
  then show ?case
    by (simp add: gval.simps(5))
next
  case (Null x)
  then show ?case
    by (metis gexp.distinct(13) gexp.distinct(17) gexp.distinct(19) gexp.distinct(7) gexp.inject(5)
    gexp_constrains.simps(2) gval.elims input_unconstrained_aval_register_swap)
qed

lemma unconstrained_variable_swap_gval:
  "∀r. ¬ gexp_constrains g (V (vname.I r)) ⇒
   ∀r. ¬ gexp_constrains g (V (R r)) ⇒
   gval g s = gval g s'"
proof(induct g)
  case (Bc x)
  then show ?case
    apply (cases x)
    apply (simp add: gval.simps(1))
    by (simp add: gval.simps(2))
next
  case (Eq x1a x2)
  then show ?case
    by (metis gexp_constrains.simps(3) gval.simps(4) unconstrained_variable_swap_aval)
next
  case (Gt x1a x2)
  then show ?case
    by (metis gexp_constrains.simps(4) gval.simps(3) unconstrained_variable_swap_aval)
next
  case (Nor g1 g2)
  then show ?case
    by (simp add: gval.simps(5))
next
  case (Null x)
  then show ?case
    by (metis gexp_constrains.simps(2) gval.simps(6) unconstrained_variable_swap_aval)
qed

lemma "gval (foldr gAnd G (Bc True)) s = foldr (∧?) (map (λg. gval g s) G) true"
proof(induct G)
  case Nil
  then show ?case
    by (simp add: gval.simps)

```

```

next
  case (Cons a G)
  then show ?case
    apply (simp only: foldr.simps comp_def gval_gAnd)
    by simp
qed

lemma possible_to_be_in: "s ≠ [] ⇒ satisfiable (In v s)"
proof(induct s)
case Nil
  then show ?case by simp
next
  case (Cons a s)
  then show ?case
    apply (case_tac s)
    apply (simp add: satisfiable_def gval.simps ValueEq_def)
    apply (case_tac v)
    apply (simp add: join_ir_def input2state_exists)
    apply (simp add: join_ir_def)
    using input2state_exists apply blast
    apply (simp add: satisfiable_def gval_gOr)
    by (metis ValueEq_def gval.simps(4) plus_trilean.simps(4) plus_trilean.simps(5) plus_trilean_commutative)
qed

lemma gAnd_commute: "gval (gAnd a b) s = gval (gAnd b a) s"
  using gval_gAnd times_trilean_commutative by auto

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 output_function = "aexp"
type_synonym update_function = "(nat × aexp)"
type_synonym updates = "update_function list"
record transition =
  Label :: label
  Arity :: nat
  Guard :: "gexp list"
  Outputs :: "aexp list"
  Updates :: updates

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))"
  by auto

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)))"

definition enumerate_inputs :: "transition ⇒ nat set" where
  "enumerate_inputs t = (⋃ (set (map enumerate_gexp_inputs (Guard t)))) ∪
    (⋃ (set (map enumerate_aexp_inputs (Outputs t)))) ∪
    (⋃ (set (map (λ(_, u). enumerate_aexp_inputs u) (Updates t))))"

definition enumerate_inputs_list :: "transition ⇒ nat list" where
  "enumerate_inputs_list t = (fold (@) (map enumerate_gexp_inputs_list (Guard t)) []) @
    (fold (@) (map enumerate_aexp_inputs_list (Outputs t)) []) @
    (fold (@) (map (λ(_, u). enumerate_aexp_inputs_list u) (Updates t)) [])"

lemma fold_enumerate_aexp_inputs_list_pairs: "set (fold (@) (map (λ(uu, y). enumerate_aexp_inputs_list
y) U) []) = (⋃ (uu, y) ∈ set U. enumerate_aexp_inputs y)"
  by (simp add: enumerate_aexp_inputs_list fold_append_concat_rev inf_sup_aci(5) split_def)

lemma fold_enumerate_aexp_inputs_list: "set (fold (@) (map enumerate_aexp_inputs_list P) []) = (⋃ x ∈ set
P. enumerate_aexp_inputs x)"
  by (simp add: enumerate_aexp_inputs_list fold_append_concat_rev inf_sup_aci(5) split_def)

lemma fold_enumerate_gexp_inputs_list: "set (fold (@) (map enumerate_gexp_inputs_list G) []) = (⋃ x ∈ set
G. enumerate_gexp_inputs x)"
  by (simp add: enumerate_gexp_inputs_list fold_append_concat_rev inf_sup_aci(5) split_def)

lemma set_enumerate_inputs_list: "enumerate_inputs t = set (enumerate_inputs_list t)"
  apply (simp add: enumerate_inputs_list_def enumerate_inputs_def)
  apply (simp add: fold_enumerate_aexp_inputs_list fold_enumerate_aexp_inputs_list_pairs)
  apply (simp add: fold_enumerate_gexp_inputs_list)
  by auto

lemma set_list_not_empty: "(set l ≠ {}) = (l ≠ [])"
  by simp

lemma max_set_nat_list: "(l :: nat list) ≠ [] ⇒ Max (set l) = foldr max l 0"
proof(induct l)
case Nil
  then show ?case
  by simp
next
case (Cons a l)
  then show ?case
  apply simp
  by (metis List.finite_set Max_insert Max_singleton fold.simps(1) fold_simps(1) foldr.simps(1) max_OR
set_empty)
qed

definition max_input :: "transition ⇒ nat option" where
  "max_input t = (if enumerate_inputs t = {} then None else Some (Max (enumerate_inputs t)))"

definition total_max_input :: "transition ⇒ nat" where
  "total_max_input t = (case max_input t of None ⇒ 0 | Some a ⇒ a)"

definition enumerate_registers :: "transition ⇒ nat set" where
  "enumerate_registers t = (⋃ (set (map enumerate_gexp_regs (Guard t)))) ∪
    (⋃ (set (map enumerate_aexp_regs (Outputs t)))) ∪
    (⋃ (set (map (λ(_, u). enumerate_aexp_regs u) (Updates t)))) ∪
    (⋃ (set (map (λ(r, _). enumerate_aexp_regs (V (R r))) (Updates t))))"

definition enumerate_registers_list :: "transition ⇒ nat list" where
  "enumerate_registers_list t = (fold (@) (map enumerate_gexp_regs_list (Guard t)) []) @
    (fold (@) (map enumerate_aexp_regs_list (Outputs t)) []) @
    (fold (@) (map (λ(_, u). enumerate_aexp_regs_list u) (Updates t)) []) @
    (fold (@) (map (λ(r, _). enumerate_aexp_regs_list (V (R r))) (Updates t)) [])"

```

```

      (fold (@) (map enumerate_aexp_regs_list (Outputs t)) []) @
      (fold (@) (map (λ(_, u). enumerate_aexp_regs_list u) (Updates t)) [])
@
      (fold (@) (map (λ(r, _). enumerate_aexp_regs_list (V (R r))) (Updates
t)) [])"

lemma fold_enumerate_aexp_regs_list_pairs: "set (fold (@) (map (λ(uu, y). enumerate_aexp_regs_list
y) U) []) = (⋃ (uu, y) ∈ set U. enumerate_aexp_regs y)"
  by (simp add: enumerate_aexp_regs_list fold_append_concat_rev inf_sup_aci(5) split_def)

lemma fold_enumerate_aexp_regs_list_pairs_2: "set (fold (@) (map (λ(r, uu). [r]) (Updates t)) []) =
(⋃ x ∈ set (Updates t). case x of (r, uu) ⇒ {r})"
  by (simp add: enumerate_aexp_regs_list fold_append_concat_rev inf_sup_aci(5) split_def)

lemma fold_enumerate_aexp_regs_list: "set (fold (@) (map enumerate_aexp_regs_list P) []) = (⋃ x ∈ set
P. enumerate_aexp_regs x)"
  by (simp add: enumerate_aexp_regs_list fold_append_concat_rev inf_sup_aci(5) split_def)

lemma fold_enumerate_gexp_regs_list: "set (fold (@) (map enumerate_gexp_regs_list G) []) = (⋃ x ∈ set
G. enumerate_gexp_regs x)"
  by (simp add: enumerate_gexp_regs_list fold_append_concat_rev inf_sup_aci(5) split_def)

lemma set_enumerate_registers_list: "enumerate_registers t = set (enumerate_registers_list t)"
  apply (simp add: enumerate_registers_list_def enumerate_registers_def)
  apply (simp add: fold_enumerate_aexp_regs_list fold_enumerate_aexp_regs_list_pairs fold_enumerate_aexp_regs_list)
  apply (simp add: fold_enumerate_gexp_regs_list)
  by auto

definition max_reg :: "transition ⇒ nat option" where
  "max_reg t = (if enumerate_registers t = {} then None else Some (Max (enumerate_registers t)))"

lemma max_reg_none_no_updates: "Transition.max_reg t = None ⇒ Updates t = []"
  apply (simp add: Transition.max_reg_def)
  apply (case_tac "enumerate_registers t = {}")
  apply (simp add: enumerate_registers_def)
  apply (case_tac "Updates t")
  by auto

definition total_max_reg :: "transition ⇒ nat" where
  "total_max_reg t = (case max_reg t of None ⇒ 0 | Some a ⇒ a)"

definition valid_transition :: "transition ⇒ bool" where
  "valid_transition t = (case max_input t of None ⇒ Arity t = 0 | Some x ⇒ x < Arity t)"

lemma not_leq_gt_set: "(∀ x ∈ (s :: ('a :: linorder) set). ¬ a ≤ x) = (∀ x ∈ s. a > x)"
  by auto

lemma fold_append_Union: "(⋃ x ∈ set G. set (enumerate_gexp_inputs_list x)) = set (fold (@) (map enumerate_gexp_i
G) [])"
proof(induct G)
  case Nil
  then show ?case
    by simp
next
  case (Cons a G)
  then show ?case
    by (simp add: fold_append_concat_rev inf_sup_aci(5))
qed

lemma map_enumerate_gexp_inputs:
  "(Max (⋃ (set (map enumerate_gexp_inputs G))) =

```

```

    (fold (max) (fold (@) (map enumerate_gexp_inputs_list G) []) 0)) ∨ ((⋃ (set (map enumerate_gexp_inputs
G)) = {}))"
proof(induct G)
  case Nil
  then show ?case
  by simp
next
  case (Cons a G)
  then show ?case
  apply simp
  apply (simp add: enumerate_gexp_inputs_list)
  apply (simp add: fold_append_Union)
  by (metis (no_types, lifting) List.finite_set Max.set_eq_fold Max.insert fold_append_concat_rev
inf_sup_aci(5) list.simps(15) max_OL set_append set_empty sup.left_idem sup_bot.right_neutral)
qed

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

definition "fSum ≡ fsum (λx. x)"

syntax (ASCII)
  "_fBall"      :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3ALL (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"       ("(3EX (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex1"      :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"       ("(3EX! (_/:_)./ _)" [0, 0, 10] 10)

syntax (input)
  "_fBall"      :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3! (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"       ("(3? (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex1"      :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"       ("(3?! (_/:_)./ _)" [0, 0, 10] 10)

syntax
  "_fBall"      :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3∀ (_/|∈|_)./ _)" [0, 0, 10] 10)
  "_fBex"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"       ("(3∃ (_/|∈|_)./ _)" [0, 0, 10] 10)
  "_fBex1"      :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"       ("(3∃! (_/|∈|_)./ _)" [0, 0, 10] 10)

translations
  "∀ x |∈| A. P" ⇒ "CONST fBall A (λx. P)"
  "∃ x |∈| A. P" ⇒ "CONST fBex A (λx. P)"
  "∃ !x |∈| A. P" ⇒ "∃ !x. x |∈| A ∧ P"

lemma fis_singleton_code[code]: "fis_singleton s = (size s = 1)"
  apply (simp add: fis_singleton_def is_singleton_def)
  by (simp add: card_Suc_eq)

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 Abs_fimage: "(fimage f s = s') = (Set.image f (fset s) = (fset s'))"
  by (simp add: fimage_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 finset_equiv: "(finset e f = f') = (insert e (fset f) = (fset f'))"
  by (simp add: finset_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 (|∪|) 1 {} = 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

lemma fsubset_strict: "x2 |⊂| x1 ⟹ ∃ e. e |∈| x1 ∧ e |∉| x2"
  by auto

lemma fsubset: "x2 |⊂| x1 ⟹ ∄ e. e |∈| x2 ∧ e |∉| x1"
  by auto

lemma size_fsubset_elem:
  "∃ e. e |∈| x1 ∧ e |∉| x2 ⟹
  ∄ e. e |∈| x2 ∧ e |∉| x1 ⟹
  size x2 < size x1"
  apply (simp add: fmember_def)
  by (metis card_seteq finite_fset linorder_not_le subsetI)

lemma size_fsubset: "x2 |⊂| x1 ⟹ size x2 < size x1"
  using fsubset fsubset_strict size_fsubset_elem
  by metis

definition fremove :: "'a ⇒ 'a fset ⇒ 'a fset"
  where [code_abbrev]: "fremove x A = A - {|x|}"

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 deviation 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 EFMS
  imports "~/src/HOL/Library/FSet" Transition FSet_Utils
begin

declare One_nat_def [simp del]

type_synonym cfstate = nat
type_synonym inputs = "value list"
type_synonym outputs = "value option list"

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

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

lemma str_not_num: "Str s ≠ Num x1"
  by (simp add: Str_def)

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

lemma "S e = (fst ∘ fst) |' e |∪| (snd ∘ fst) |' e"
  apply (simp add: comp_def S_def)
  by force

definition apply_outputs :: "aexp list ⇒ datastate ⇒ value option list" where
  "apply_outputs p s = map (λp. aval p s) p"

lemmas apply_outputs = datastate apply_outputs_def

lemma apply_outputs_empty [simp]: "apply_outputs [] s = []"
  by (simp add: apply_outputs_def)

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

definition apply_guards :: "gexp list ⇒ datastate ⇒ bool" where
  "apply_guards G s = (∀ g ∈ set (map (λg. gval g s) G). g = true)"

lemma apply_guards_rearrange: "x ∈ set G ⟹ apply_guards G s = apply_guards (x#G) s"
  apply (simp add: apply_guards_def)
  by auto

lemma apply_guards_double_cons: "apply_guards (y # x # G) s = (gval (gAnd y x) s = true ∧ apply_guards G s)"
  by (simp add: apply_guards_def gval_gAnd_True)

lemmas apply_guards = datastate apply_guards_def gval.simps ValueEq_def ValueGt_def

lemma apply_guards_singleton: "(apply_guards [g] s) = (gval g s = true)"
  by (simp add: apply_guards_def)

lemma apply_guards_empty [simp]: "apply_guards [] s"
  by (simp add: apply_guards_def)

lemma apply_guards_cons: "apply_guards (a # G) c = (gval a c = true ∧ apply_guards G c)"

```

```

by (simp add: apply_guards_def)

lemma apply_guards_append: "apply_guards (a@a') s = (apply_guards a s ∧ apply_guards a' s)"
  apply (simp add: apply_guards_def)
  by auto

lemma apply_guards_foldr: "apply_guards G s = (gval (foldr gAnd G (Bc True)) s = true)"
proof(induct G)
  case Nil
  then show ?case
    by (simp add: apply_guards_def gval.simps)
next
  case (Cons a G)
  then show ?case
    apply (simp only: foldr.simps comp_def)
    by (simp add: apply_guards_cons gval_gAnd maybe_and_true)
qed

lemma apply_guards_rev: "apply_guards G s = apply_guards (rev G) s"
  by (simp add: apply_guards_def)

lemma apply_guards_fold: "apply_guards G s = (gval (fold gAnd G (Bc True)) s = true)"
  using apply_guards_rev
  by (simp add: foldr_conv_fold apply_guards_foldr)

lemma fold_apply_guards: "(gval (fold gAnd G (Bc True)) s = true) = apply_guards G s"
  by (simp add: apply_guards_fold)

lemma foldr_apply_guards: "(gval (foldr gAnd G (Bc True)) s = true) = apply_guards G s"
  by (simp add: apply_guards_foldr)

lemma apply_guards_subset: "set g' ⊆ set g ⇒ apply_guards g c ⟶ apply_guards g' c"
proof(induct g)
  case Nil
  then show ?case
    by simp
next
  case (Cons a g)
  then show ?case
    apply (simp add: apply_guards_def)
    by auto
qed

definition choice :: "transition ⇒ transition ⇒ bool" where
  "choice t t' = (∃ i r. apply_guards (Guard t) (join_ir i r) ∧ apply_guards (Guard t') (join_ir i r))"

definition choice_alt :: "transition ⇒ transition ⇒ bool" where
  "choice_alt t t' = (∃ i r. apply_guards (Guard t@Guard t') (join_ir i r))"

lemma choice_alt: "choice t t' = choice_alt t t'"
  by (simp add: choice_def choice_alt_def apply_guards_append)

lemma choice_symmetry: "choice x y = choice y x"
  using choice_def by auto

primrec apply_updates :: "updates ⇒ datastate ⇒ registers ⇒ registers" where
  "apply_updates [] _ new = new" |
  "apply_updates (h#t) old new = (apply_updates t old new)(fst h $:= aval (snd h) old)"

lemma apply_updates_foldr: "apply_updates u old new = foldr (λh r. r(fst h $:= aval (snd h) old)) u
  new"

```

```

proof(induct u)
  case Nil
  then show ?case
    by simp
next
  case (Cons a u)
  then show ?case
    apply (simp add: eq_finfun_All_ext finfun_All_def finfun_All_except_def)
    by (simp add: Cons.hyps)
qed

lemma r_not_updated_stays_the_same: "r ∉ fst ` set U ⇒
  apply_updates U c d $ r = d $ r"
proof(induct U)
  case Nil
  then show ?case
    by simp
next
  case (Cons a U)
  then show ?case
    by simp
qed

definition possible_steps :: "transition_matrix ⇒ nat ⇒ registers ⇒ 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 in_possible_steps: "(a, bb) |∈| possible_steps b 0 <> ab ba ⇒ ∃ s. ((s, a), bb) |∈| b"
  apply (simp add: possible_steps_def fimage_def ffilter_def fmember_def Abs_fset_inverse)
  by auto

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) (join_ir i r)) 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 vname.I
n ⇒ input2state i $ n | R n ⇒ 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 join_ir_def)
    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 $ n) (λn. 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) (join_ir i r))
    e =
    {/(s, d), t/})"
  apply standard
  apply (simp add: possible_steps_def possible_steps_alt_aux join_ir_def)
  by (simp add: possible_steps_def join_ir_def)

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

lemmas ffilter = ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def
lemmas possible_steps_singleton = ffilter possible_steps_alt
lemmas possible_steps_empty = ffilter possible_steps_def

definition step :: "transition_matrix ⇒ cfstate ⇒ registers ⇒ label ⇒ inputs ⇒ (transition × cfstate
× outputs × registers) 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_1: "possible_steps e s r l i = {/}/ ⇒ step e s r l i = None"
  by (simp add: step_def)

lemma no_possible_steps_2: "step e s r l i = None ⇒ possible_steps e s r l i = {/}/"
  apply (simp add: step_def Let_def)
  apply (case_tac "possible_steps e s r l i = {/}/")
  apply simp
  apply (case_tac "SOME x. x |∈| possible_steps e s r l i")
  by simp

lemma no_possible_steps_3: "(possible_steps e s r l i = {/}/) = (step e s r l i = None)"
  using no_possible_steps_1 no_possible_steps_2 by blast

lemma no_possible_steps_4: "(step e s r l i = None) = (possible_steps e s r l i = {/}/)"
  using no_possible_steps_1 no_possible_steps_2 by blast

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)"
  by (simp add: step_def)

lemma step_empty[simp]: "step {/}/ s r l i = None"
  by (simp add: step_def possible_steps_def ffilter_empty)

primrec observe_all :: "transition_matrix ⇒ nat ⇒ registers ⇒ trace ⇒ (transition × nat × outputs

```

```

× registers) 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 ⇒ registers ⇒ 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_empty [simp]: "observe_trace e s r [] = []"
  by (simp add: observe_trace_def)

lemma observe_trace_step: "
  step e s r (fst h) (snd h) = Some (t, s', p, r') ⇒
  observe_trace e s' r' es = obs ⇒
  observe_trace e s r (h#es) = p#obs"
  by (simp add: observe_trace_def)

lemma observe_trace_possible_step: "possible_steps e s r (fst h) (snd h) = {|(s', t)|} ⇒
  apply_outputs (Outputs t) (join_ir (snd h) r) = p ⇒
  apply_updates (Updates t) (join_ir (snd h) r) r = r' ⇒
  observe_trace e s' r' es = obs ⇒
  observe_trace e s r (h#es) = p#obs"
  using observe_trace_step one_possible_step
  by simp

lemma observe_trace_no_possible_step: "possible_steps e s r (fst h) (snd h) = {|}| ⇒
  observe_trace e s r (h#es) = []"
  by (simp add: observe_trace_def step_def)

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

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_possible_step:
  "possible_steps e1 s1 r1 (fst h) (snd h) = {|(s1', t1)|} ⇒
  possible_steps e2 s2 r2 (fst h) (snd h) = {|(s2', t2)|} ⇒
  apply_outputs (Outputs t1) (join_ir (snd h) r1) = apply_outputs (Outputs t2) (join_ir (snd h) r2)
⇒
  apply_updates (Updates t1) (join_ir (snd h) r1) r1 = r1' ⇒
  apply_updates (Updates t2) (join_ir (snd h) r2) r2 = r2' ⇒
  observe_trace e1 s1' r1' t = observe_trace e2 s2' r2' t ⇒
  observe_trace e1 s1 r1 (h#t) = observe_trace e2 s2 r2 (h#t)"
  by (simp add: observe_trace_possible_step)

lemma efsm_equiv_no_possible_step:
  "possible_steps e1 s1 r1 (fst h) (snd h) = {|}| ⇒
  possible_steps e2 s2 r2 (fst h) (snd h) = {|}| ⇒
  observe_trace e1 s1 r1 (h#t) = observe_trace e2 s2 r2 (h#t)"
  by (simp add: observe_trace_no_possible_step)

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

```

```

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

lemma efsm_equiv_transitive: "efsm_equiv e1 e2 t  $\wedge$  efsm_equiv e2 e3 t  $\longrightarrow$  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: " $\bigwedge s\ r.$  length (observe_all e s r t)  $\leq$  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)  $\neq$  None"
    with Cons show ?thesis by(auto)
  qed
qed

inductive accepts :: "transition_matrix  $\Rightarrow$  nat  $\Rightarrow$  registers  $\Rightarrow$  trace  $\Rightarrow$  bool" where
  base: "accepts e s d []" |
  step: " $\exists (s', T) \mid \mid$  possible_steps e s d (fst h) (snd h). accepts e s' (apply_updates (Updates T)
(join_ir (snd h) d) d) t  $\Longrightarrow$ 
accepts e s d (h#t)"

lemma accepts_cons: "accepts e s d (h#t) = ( $\exists (s', T) \mid \mid$  possible_steps e s d (fst h) (snd h). accepts
e s' (apply_updates (Updates T) (join_ir (snd h) d) d) t)"
  apply standard
  apply (metis (mono_tags) accepts.simps list.distinct(1) list.inject)
  by (simp add: accepts.step)

lemma accepts_cons_step: "accepts e s r (a # t)  $\Longrightarrow$  step e s r (fst a) (snd a)  $\neq$  None"
  using accepts_cons no_possible_steps_4 by auto

lemma step_some_updated: "step e s r l i = Some (transition, s', outputs, updated)  $\Longrightarrow$ 
updated = (apply_updates (Updates transition) (join_ir i r) r)"
  apply (simp add: step_def Let_def)
  apply (case_tac "possible_steps e s r l i = {||}")
  apply simp
  apply simp
  apply (case_tac "SOME x. x  $\mid \mid$  possible_steps e s r l i")
  by auto

lemma step_in_possible_steps:
  "step e s r l i = Some (transition, s', outputs, updated)  $\Longrightarrow$ 
(s', transition)  $\mid \mid$  possible_steps e s r l i"
  apply (simp add: step_def Let_def)
  apply (case_tac "possible_steps e s r l i = {||}")
  apply simp
  apply simp
  apply (case_tac "SOME x. x  $\mid \mid$  possible_steps e s r l i")

```

```

    apply simp
    apply clarify
    by (metis equalsffemptyI verit_sko_ex')

lemma accepts_step: "step e s r l i = Some (transition, s', outputs, updated)  $\implies$ 
    accepts e s' updated t  $\implies$ 
    accepts e s r ((l, i)#t)"
    apply (rule accepts.step)
    apply (rule_tac x="(s', transition)" in FSet.fBexI)
    using step_some_updated apply simp
    by (simp add: step_in_possible_steps)

abbreviation accepts_trace :: "transition_matrix  $\Rightarrow$  trace  $\Rightarrow$  bool" where
    "accepts_trace e t  $\equiv$  accepts e 0 <> t"

lemma no_step_none: "step e s r aa ba = None  $\implies$   $\neg$  accepts e s r ((aa, ba) # p)"
    apply safe
    apply (rule accepts.cases)
    apply simp
    apply simp
    apply clarify
    apply (simp add: step_def Let_def)
    apply (case_tac "possible_steps e s r aa ba")
    apply simp
    apply (case_tac "SOME xa. xa = x  $\vee$  xa  $\in$  S'")
    by simp

lemma step_none_inaccepts: "((step e s d (fst h) (snd h)) = None)  $\implies$   $\neg$  (accepts e s d (h#t))"
    using no_step_none surjective_pairing by fastforce

lemma accepts_head: "accepts e s d (h#t)  $\implies$  accepts e s d [h]"
    apply (rule accepts.cases)
    apply simp
    apply simp
    apply (rule accepts.step)
    apply (simp add: accepts.base)
    by auto

lemma accepts_single_event: "step e s d a b  $\neq$  None  $\implies$  accepts e s d [(a, b)]"
    apply (rule accepts.step)
    apply (simp add: step_def Let_def)
    apply (case_tac "possible_steps e s d a b = {||}")
    apply simp
    apply (case_tac "SOME x. x  $\in$  possible_steps e s d a b")
    apply (simp add: accepts.base)
    by auto

lemma inaccepts_single_event: " $\neg$  accepts e s d [(a, b)]  $\implies$  step e s d a b = None"
    using accepts_single_event by blast

lemma single_event_reject: " $\neg$  accepts e s d [(a, b)] = (step e s d a b = None)"
    using accepts_single_event no_step_none by blast

lemma trace_reject: "(possible_steps e s d a b = {||}  $\vee$  ( $\forall$  (s', T)  $\in$  possible_steps e s d a b.  $\neg$  accepts
e s' (apply_updates (Updates T) (join_ir b d) d) t)) = ( $\neg$  accepts e s d ((a, b)#t))"
    apply safe
    apply (simp add: no_possible_steps_4 no_step_none)
    apply (rule accepts.cases)
    apply simp
    apply simp
    apply clarify

```

```

    apply simp
    apply auto[1]
    using accepts.intros(2) by fastforce

lemma trace_reject_2: "(¬ accepts e s d ((a, b)#t)) = (possible_steps e s d a b = {||} ∨ (∀ (s', T)
|∈| possible_steps e s d a b. ¬ accepts e s' (apply_updates (Updates T) (join_ir b d) d) t))"
  using trace_reject
  by simp

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 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 (rule accepts.cases)
    apply simp
    apply simp
    using accepts.step by blast
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
    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

inductive gets_us_to :: "nat ⇒ transition_matrix ⇒ nat ⇒ registers ⇒ trace ⇒ bool" where
  base: "s = target ⇒ gets_us_to target _ s _ []" |
  step_some: "∃ (s', T) |∈| possible_steps e s d (fst h) (snd h). gets_us_to target e s' (apply_updates
(Updates T) (join_ir i r) 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  $\Rightarrow$  nat  $\Rightarrow$  bool" where
  "incoming_transition_to t s = ((ffilter ( $\lambda$ ((from, to), t). to = s) t)  $\neq$  {})"
```

lemma incoming_transition_alt_def: "incoming_transition_to e n = (\exists t from. ((from, n), t) \in 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

primrec accepting_sequence :: "transition_matrix  $\Rightarrow$  cfstate  $\Rightarrow$  registers  $\Rightarrow$  trace  $\Rightarrow$  (transition  $\times$ 
cfstate  $\times$  outputs  $\times$  registers) list  $\Rightarrow$  (transition  $\times$  cfstate  $\times$  outputs  $\times$  registers) list option"
where
  "accepting_sequence _ _ r [] obs = Some (rev obs)" |
  "accepting_sequence e s r (a#t) obs = (let
    poss = possible_steps e s r (fst a) (snd a);
    accepting = ffilter ( $\lambda$ (s', T). accepts e s' (apply_updates (Updates T) (join_ir (snd a) r) r) t)
  in
    if accepting = {} then
      None
    else let
      (s', T) = Eps ( $\lambda$ x. x  $\in$  accepting);
      r' = (apply_updates (Updates T) (join_ir (snd a) r) r) in
      accepting_sequence e s' r' t ((T, s', (apply_outputs (Outputs T) (join_ir (snd a) r)), r')#obs)
    )"

end
```

7 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  $\Rightarrow$  'a stream  $\Rightarrow$  bool" where
  [code_abbrev]: "smember x s  $\longleftrightarrow$  x  $\in$  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  $\in$  sset s" and " $\bigwedge$ s. P (shd s) s" and " $\bigwedge$ s y.  $\llbracket$ y  $\in$  sset (stl s); P y (stl s) $\rrbracket \Longrightarrow$  P y
```

```

s"
  shows "P y s"
using assms by induct (metis stream.sel(1), auto)

lemma smap_ctr: "smap f s = x ## s'  $\longleftrightarrow$  f (shd s) = x  $\wedge$  smap f (stl s) = s'"
  by (cases s) simp

```

7.1 prepend list to stream

```

primrec shift :: "'a list  $\Rightarrow$  'a stream  $\Rightarrow$  '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  $\cup$  sset s"
  by (induct xs) auto

lemma shift_left_inj[simp]: "xs @- s1 = xs @- s2  $\longleftrightarrow$  s1 = s2"
  by (induct xs) auto

```

7.2 set of streams with elements in some fixed set

```

context
  notes [[inductive_internals]]
begin

coinductive_set
  streams :: "'a set  $\Rightarrow$  'a stream set"
  for A :: "'a set"
where
  Stream[intro!, simp, no_atp]: "[a  $\in$  A; s  $\in$  streams A]  $\Longrightarrow$  a ## s  $\in$  streams A"

end

lemma in_streams: "stl s  $\in$  streams S  $\Longrightarrow$  shd s  $\in$  S  $\Longrightarrow$  s  $\in$  streams S"
  by (cases s) auto

lemma streamsE: "s  $\in$  streams A  $\Longrightarrow$  (shd s  $\in$  A  $\Longrightarrow$  stl s  $\in$  streams A  $\Longrightarrow$  P)  $\Longrightarrow$  P"
  by (erule streams.cases) simp_all

lemma Stream_image: "x ## y  $\in$  ((##) x') ' Y  $\longleftrightarrow$  x = x'  $\wedge$  y  $\in$  Y"
  by auto

lemma shift_streams: "[w  $\in$  lists A; s  $\in$  streams A]  $\Longrightarrow$  w @- s  $\in$  streams A"
  by (induct w) auto

lemma streams_Stream: "x ## s  $\in$  streams A  $\longleftrightarrow$  x  $\in$  A  $\wedge$  s  $\in$  streams A"
  by (auto elim: streams.cases)

lemma streams_stl: "s  $\in$  streams A  $\Longrightarrow$  stl s  $\in$  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)

```

7.3 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  $\longleftrightarrow$  ( $\forall n. s !! n \in X$ )"
  by (force simp: streams_iff_sset sset_range)

lemma snth_in: "s ∈ streams X  $\implies s !! n \in X$ "
  by (simp add: streams_iff_snth)

primrec stake :: "nat  $\Rightarrow$  'a stream  $\Rightarrow$  '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  $\Rightarrow$  'a stream  $\Rightarrow$  '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'  $\longleftrightarrow$  ( $\forall n. f (s !! n) = s' !! n$ )" (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

```

7.4 unary predicates lifted to streams

```

definition "stream_all P s = (∀p. P (s !! p))"

```

```

lemma stream_all_iff[iff]: "stream_all P s  $\longleftrightarrow$  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  $\wedge$  stream_all P s)"
  unfolding stream_all_iff list_all_iff by auto

lemma stream_all_Stream: "stream_all P (x ## X)  $\longleftrightarrow$  P x  $\wedge$  stream_all P X"
  by simp

```

7.5 recurring stream out of a list

```

primcorec cycle :: "'a list  $\Rightarrow$  'a stream" where
  "shd (cycle xs) = hd xs"
| "stl (cycle xs) = cycle (tl xs @ [hd xs])"

lemma cycle_decomp: "u  $\neq$  []  $\implies$  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  $\neq$  []; cycle u = v @- s]  $\implies$  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  $\neq$  []" "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  $\neq$  []  $\implies$  stake (length u) (cycle u) = u"
  by (subst cycle_decomp) (auto simp: stake_shift)

lemma sdrop_cycle_eq[simp]: "u  $\neq$  []  $\implies$  sdrop (length u) (cycle u) = cycle u"
  by (subst cycle_decomp) (auto simp: sdrop_shift)

lemma stake_cycle_eq_mod_0[simp]: "[u  $\neq$  []; n mod length u = 0]  $\implies$ 
  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  $\neq$  []; n mod length u = 0]  $\implies$ 
  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  $\neq$  []  $\implies$ 
  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  $\neq$  []  $\implies$  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)

```

7.6 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

```

7.7 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)
```

7.8 stream of natural numbers

abbreviation "fromN \equiv siterate Suc"

abbreviation "nats \equiv 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
```

7.9 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

```

7.10 merge a stream of streams

```

definition smerge :: "'a 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) = ⋃ (sset ' (sset ss))"
proof safe
  fix x assume "x ∈ sset (smerge ss)"
  thus "x ∈ ⋃ (sset ' (sset ss))"
    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

```

7.11 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)

```

7.12 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  $\implies$  sinterleave s1 s2 !! n = s1 !! (n div 2)"
  "odd n  $\implies$  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  $\cup$  sset s2"
proof (intro equalityI subsetI)
  fix x assume "x  $\in$  sset (sinterleave s1 s2)"
  then obtain n where "x = sinterleave s1 s2 !! n" unfolding sset_range by blast
  thus "x  $\in$  sset s1  $\cup$  sset s2" by (cases "even n") auto
next
  fix x assume "x  $\in$  sset s1  $\cup$  sset s2"
  thus "x  $\in$  sset (sinterleave s1 s2)"
  proof
    assume "x  $\in$  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  $\in$  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
```

7.13 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 ( $\lambda x. f$  (fst x)) (szip s1 s2) = smap f s1"
  by (coinduction arbitrary: s1 s2) auto

lemma smap_szip snd:
  "smap ( $\lambda x. g$  (snd x)) (szip s1 s2) = smap g s2"
```

```
by (coinduction arbitrary: s1 s2) auto
```

7.14 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
```

8 List prefixes, suffixes, and homeomorphic embedding

```
theory Sublist
imports Main
begin
```

8.1 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  $\implies$  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  $\neq$  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  $\implies$  xs  $\neq$  ys  $\implies$  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  $\neq$  ys"
  using assms unfolding strict_prefix_def by blast



## 8.2 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])  $\longleftrightarrow$  xs = ys @ [y]  $\vee$  prefix xs ys"
proof
  assume "prefix xs (ys @ [y])"
  then obtain zs where zs: "ys @ [y] = xs @ zs" ..
  show "xs = ys @ [y]  $\vee$  prefix xs ys"
    by (metis append_Nil2 butlast_append butlast_snoc prefixI zs)
next
  assume "xs = ys @ [y]  $\vee$  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  $\wedge$  prefix xs ys)"
  by (auto simp add: prefix_def)

lemma prefix_code [code]:
  "prefix [] xs  $\longleftrightarrow$  True"
  "prefix (x # xs) []  $\longleftrightarrow$  False"
  "prefix (x # xs) (y # ys)  $\longleftrightarrow$  x = y  $\wedge$  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  $\implies$  prefix xs (ys @ zs)"
  unfolding prefix_def by fastforce

lemma append_prefixD: "prefix (xs @ ys) zs  $\implies$  prefix xs zs"
  by (auto simp add: prefix_def)

theorem prefix_Cons: "prefix xs (y # ys) = (xs = []  $\vee$  ( $\exists$ zs. xs = y # zs  $\wedge$  prefix zs ys))"
  by (cases xs) (auto simp add: prefix_def)

theorem prefix_append:
  "prefix xs (ys @ zs) = (prefix xs ys  $\vee$  ( $\exists$ us. xs = ys @ us  $\wedge$  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  $\implies$  length xs < length ys  $\implies$  prefix (xs @ [ys ! length xs]) ys"
  proof (unfold prefix_def)
    assume a1: " $\exists$ zs. ys = xs @ zs"
    then obtain sk :: "'a list" where sk: "ys = xs @ sk" by fastforce
    assume a2: "length xs < length ys"
    have f1: " $\bigwedge$ v. ([::'a list) @ v = v" using append_Nil2 by simp
    have "[ ]  $\neq$  sk" using a1 a2 sk less_not_refl by force
    hence " $\exists$ v. xs @ hd sk # v = ys" using sk by (metis hd_Cons_tl)
    thus " $\exists$ zs. ys = (xs @ [ys ! length xs]) @ zs" using f1 by fastforce
  qed

theorem prefix_length_le: "prefix xs ys  $\implies$  length xs  $\leq$  length ys"
  by (auto simp add: prefix_def)

lemma prefix_same_cases:
  "prefix (xs1::'a list) ys  $\implies$  prefix xs2 ys  $\implies$  prefix xs1 xs2  $\vee$  prefix xs2 xs1"
  unfolding prefix_def by (force simp: append_eq_append_conv2)

lemma prefix_length_prefix:
  "prefix ps xs  $\implies$  prefix qs xs  $\implies$  length ps  $\leq$  length qs  $\implies$  prefix ps qs"
  by (auto simp: prefix_def) (metis append_Nil2 append_eq_append_conv_if)

lemma set_mono_prefix: "prefix xs ys  $\implies$  set xs  $\subseteq$  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  $\implies$  prefix (map f xs) (map f ys)"
  by (auto simp: prefix_def)

lemma filter_mono_prefix: "prefix xs ys  $\implies$  prefix (filter P xs) (filter P ys)"
  by (auto simp: prefix_def)

lemma sorted_antimono_prefix: "prefix xs ys  $\implies$  sorted ys  $\implies$  sorted xs"
  by (metis sorted_append prefix_def)

```

```

lemma prefix_length_less: "strict_prefix xs ys  $\implies$  length xs < length ys"
  by (auto simp: strict_prefix_def prefix_def)

lemma prefix_snocD: "prefix (xs@[x]) ys  $\implies$  strict_prefix xs ys"
  by (simp add: strict_prefixI' prefix_order.dual_order.strict_trans1)

lemma strict_prefix_simps [simp, code]:
  "strict_prefix xs []  $\longleftrightarrow$  False"
  "strict_prefix [] (x # xs)  $\longleftrightarrow$  True"
  "strict_prefix (x # xs) (y # ys)  $\longleftrightarrow$  x = y  $\wedge$  strict_prefix xs ys"
  by (simp_all add: strict_prefix_def cong: conj_cong)

lemma take_strict_prefix: "strict_prefix xs ys  $\implies$  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: " $\neg$  prefix ps ls"
  obtains
    (c1) "ps  $\neq$  []" and "ls = []"
  | (c2) a as x xs where "ps = a#as" and "ls = x#xs" and "x = a" and " $\neg$  prefix as xs"
  | (c3) a as x xs where "ps = a#as" and "ls = x#xs" and "x  $\neq$  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 " $\neg$  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: " $\neg$  prefix ps ls"
  and base: " $\bigwedge$  x xs. P (x#xs) []"
  and r1: " $\bigwedge$  x xs y ys. x  $\neq$  y  $\implies$  P (x#xs) (y#ys)"
  and r2: " $\bigwedge$  x xs y ys. [ $x = y$ ;  $\neg$  prefix xs ys; P xs ys]  $\implies$  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

```

8.3 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

```

```

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)  $\cup$  {xs @ ys' | ys'. ys'  $\in$  set (prefixes ys)}"
  by (subst prefixes_append, cases ys) auto

```

8.4 Longest Common Prefix

```

definition Longest_common_prefix :: "'a list set  $\Rightarrow$  'a list" where
  "Longest_common_prefix L = (ARG_MAX length ps.  $\forall$  xs  $\in$  L. prefix ps xs)"

lemma Longest_common_prefix_ex: "L  $\neq$  {}  $\implies$ 
   $\exists$  ps. ( $\forall$  xs  $\in$  L. prefix ps xs)  $\wedge$  ( $\forall$  qs. ( $\forall$  xs  $\in$  L. prefix qs xs)  $\longrightarrow$  size qs  $\leq$  size ps)"
  (is "_  $\implies \exists$  ps. ?P L ps")
proof(induction "LEAST n.  $\exists$  xs  $\in$  L. n = length xs" arbitrary: L)
  case 0
  have "[]  $\in$  L" using "0.hyps" LeastI[of " $\lambda$ n.  $\exists$  xs  $\in$  L. n = length xs"]  $\langle$  L  $\neq$  {} $\rangle$ 
  by auto
  hence "?P L []" by(auto)
  thus ?case ..
next
  case (Suc n)
  let ?EX = " $\lambda$ n.  $\exists$  xs  $\in$  L. n = length xs"
  obtain x xs where xxs: "x#xs  $\in$  L" "size xs = n" using Suc.prem1 Suc.hyps(2)
  by (metis LeastI_ex[of ?EX] Suc_length_conv ex_in_conv)
  hence "[]  $\notin$  L" using Suc.hyps(2) by auto
  show ?case
  proof (cases " $\forall$  xs  $\in$  L.  $\exists$  ys. xs = x#ys")
    case True
    let ?L = "{ys. x#ys  $\in$  L}"
    have 1: "(LEAST n.  $\exists$  xs  $\in$  ?L. n = length xs) = n"
      using xxs Suc.prem1 Suc.hyps(2) LeastI[of "?EX"]
      by - (rule Least_equality, fastforce+)
    have 2: "?L  $\neq$  {}" using <x # xs  $\in$  L> by auto
    from Suc.hyps(1)[OF 1[symmetric] 2] obtain ps where IH: "?P ?L ps" ..
    { fix qs
      assume " $\forall$  qs. ( $\forall$  xa. x # xa  $\in$  L  $\longrightarrow$  prefix qs xa)  $\longrightarrow$  length qs  $\leq$  length ps"
      and " $\forall$  xs  $\in$  L. prefix qs xs"
      hence "length (tl qs)  $\leq$  length ps"
        by (metis Cons_prefix_Cons hd_Cons_tl list.sel(2) Nil_prefix)
      hence "length qs  $\leq$  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 $\neq$ y" "y#ys  $\in$  L" using <[]  $\notin$  L>
  by (auto) (metis list.exhaust)
  have " $\forall$  qs. ( $\forall$  xs  $\in$  L. prefix qs xs)  $\longrightarrow$  qs = []" using yys <x#xs  $\in$  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 equalsOI 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 someI_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_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 ]]
 $\implies$  prefix ps (longest_common_prefix xs ys)"
by(induction xs ys arbitrary: ps rule: longest_common_prefix.induct)
(auto simp: prefix_Cons)

```

8.5 Parallel lists

```

definition parallel :: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool" (infixl "||" 50)
where "(xs || ys) = ( $\neg$  prefix xs ys  $\wedge$   $\neg$  prefix ys xs)"

lemma parallelI [intro]: " $\neg$  prefix xs ys  $\implies$   $\neg$  prefix ys xs  $\implies$  xs || ys"
unfolding parallel_def by blast

lemma parallelE [elim]:
assumes "xs || ys"
obtains " $\neg$  prefix xs ys  $\wedge$   $\neg$  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  $\implies$   $\exists$  as b bs c cs. b  $\neq$  c  $\wedge$  xs = as @ b # bs  $\wedge$  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.premys ys)
next
fix c cs assume ys': "ys' = c # cs"
have "x  $\neq$  c" using snoc.premys ys ys' by fastforce
thus " $\exists$  as b bs c cs. b  $\neq$  c  $\wedge$  xs @ [x] = as @ b # bs  $\wedge$  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)  $\neq$  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  $\implies$  a @ c || b @ d"
  apply (rule parallelI)
  apply (erule parallelE, erule conjE,
    induct rule: not_prefix_induct, simp+)+
  done

lemma parallel_appendI: "xs || ys  $\implies$  x = xs @ xs'  $\implies$  y = ys @ ys'  $\implies$  x || y"
  by (simp add: parallel_append)

lemma parallel_commute: "a || b  $\longleftrightarrow$  b || a"
  unfolding parallel_def by auto

```

8.6 Suffix order on lists

```

definition suffix :: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool"
  where "suffix xs ys = ( $\exists$  zs. ys = zs @ xs)"

definition strict_suffix :: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool"
  where "strict_suffix xs ys  $\longleftrightarrow$  suffix xs ys  $\wedge$  xs  $\neq$  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  $\implies$  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  $\neq$  []  $\implies$  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  $\implies$  suffix xs (y # ys)"
  by (auto simp add: suffix_def)

lemma suffix_ConsD: "suffix (x # xs) ys  $\implies$  suffix xs ys"
  by (auto simp add: suffix_def)

lemma suffix_appendI: "suffix xs ys  $\implies$  suffix xs (zs @ ys)"
  by (auto simp add: suffix_def)

```

```

lemma suffix_appendD: "suffix (zs @ xs) ys  $\implies$  suffix xs ys"
  by (auto simp add: suffix_def)

lemma strict_suffix_set_subset: "strict_suffix xs ys  $\implies$  set xs  $\subseteq$  set ys"
  by (auto simp: strict_suffix_def suffix_def)

lemma set_mono_suffix: "suffix xs ys  $\implies$  set xs  $\subseteq$  set ys"
  by (auto simp: suffix_def)

lemma sorted_antimono_suffix: "suffix xs ys  $\implies$  sorted ys  $\implies$  sorted xs"
  by (metis sorted_append suffix_def)

lemma suffix_ConsD2: "suffix (x # xs) (y # ys)  $\implies$  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  $\longleftrightarrow$  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  $\longleftrightarrow$  strict_prefix (rev xs) (rev ys)"
  by (auto simp: suffix_to_prefix strict_suffix_def strict_prefix_def)

lemma distinct_suffix: "distinct ys  $\implies$  suffix xs ys  $\implies$  distinct xs"
  by (clarsimp elim!: suffixE)

lemma map_mono_suffix: "suffix xs ys  $\implies$  suffix (map f xs) (map f ys)"
  by (auto elim!: suffixE intro: suffixI)

lemma filter_mono_suffix: "suffix xs ys  $\implies$  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  $\implies$  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  $\implies$  ys  $\in$  lists A  $\implies$  xs  $\in$  lists A"
  unfolding suffix_def by auto

lemma suffix_snoc [simp]: "suffix xs (ys @ [y])  $\longleftrightarrow$  xs = []  $\vee$  ( $\exists$  zs. xs = zs @ [y]  $\wedge$  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)  $\longleftrightarrow$  xs = y # ys  $\vee$  suffix xs ys"
  unfolding suffix_def by (auto simp: Cons_eq_append_conv)

theorem suffix_append:
  "suffix xs (ys @ zs)  $\longleftrightarrow$  suffix xs zs  $\vee$  ( $\exists$  xs'. xs = xs' @ zs  $\wedge$  suffix xs' ys)"
  by (auto simp: suffix_def append_eq_append_conv2)

theorem suffix_length_le: "suffix xs ys  $\implies$  length xs  $\leq$  length ys"
  by (auto simp add: suffix_def)

lemma suffix_same_cases:
  "suffix (xs1::'a list) ys  $\implies$  suffix xs2 ys  $\implies$  suffix xs1 xs2  $\vee$  suffix xs2 xs1"
  unfolding suffix_def by (force simp: append_eq_append_conv2)

lemma suffix_length_suffix:
  "suffix ps xs  $\implies$  suffix qs xs  $\implies$  length ps  $\leq$  length qs  $\implies$  suffix ps qs"
  by (auto simp: suffix_to_prefix intro: prefix_length_prefix)

lemma suffix_length_less: "strict_suffix xs ys  $\implies$  length xs < length ys"
  by (auto simp: strict_suffix_def suffix_def)

lemma suffix_ConsD': "suffix (x#xs) ys  $\implies$  strict_suffix xs ys"
  by (auto simp: strict_suffix_def suffix_def)

lemma drop_strict_suffix: "strict_suffix xs ys  $\implies$  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: " $\neg$  suffix ps ls"
  obtains
    (c1) "ps  $\neq$  []" and "ls = []"
  | (c2) a as x xs where "ps = as@[a]" and "ls = xs@[x]" and "x = a" and " $\neg$  suffix as xs"
  | (c3) a as x xs where "ps = as@[a]" and "ls = xs@[x]" and "x  $\neq$  a"
proof (cases ps rule: rev_cases)
  case Nil
  then show ?thesis using pfx by simp
next
  case (snoc as a)
  note c =  $\langle$ ps = as@[a] $\rangle$ 
  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

```

8.7 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)  $\cup$  {xs' @ ys | xs'. xs'  $\in$  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

```

8.8 Homeomorphic embedding on lists

```

inductive list_emb :: "('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool"
  for P :: "('a  $\Rightarrow$  'a  $\Rightarrow$  bool)"
where
  list_emb_Nil [intro, simp]: "list_emb P [] ys"
| list_emb_Cons [intro] : "list_emb P xs ys  $\implies$  list_emb P xs (y#ys)"
| list_emb_Cons2 [intro]: "P x y  $\implies$  list_emb P xs ys  $\implies$  list_emb P (x#xs) (y#ys)"

lemma list_emb_mono:
  assumes " $\bigwedge x y. P x y \longrightarrow Q x y$ "
  shows "list_emb P xs ys  $\longrightarrow$  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 " $\bigwedge x. x \in \text{set } xs \implies P x x$ "
  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  $\implies$  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 " $\exists$ us v vs. ys = us @ v # vs  $\wedge$  P x v  $\wedge$  list_emb P xs vs"
  using assms
  proof (induct x  $\equiv$  "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 " $\exists$ us vs. zs = us @ vs  $\wedge$  list_emb P xs us  $\wedge$  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  $\Rightarrow$  'a list  $\Rightarrow$  'a list" and sk1 :: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list" where
      sk: " $\forall$ x0 x1.  $\neg$  list_emb P (xs @ x0) x1  $\vee$  sk0 x0 x1 @ sk1 x0 x1 = x1  $\wedge$  list_emb P xs (sk0 x0 x1)"
       $\wedge$  list_emb P x0 (sk1 x0 x1)"
      using Cons(1) by (metis (no_types))
    hence " $\forall$ 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  $\implies$  length xs  $\leq$  length ys"
  by (induct rule: list_emb.induct) auto

lemma list_emb_trans:
  assumes " $\bigwedge$ x y z.  $\llbracket$ x  $\in$  set xs; y  $\in$  set ys; z  $\in$  set zs; P x y; P y z $\rrbracket \implies$  P x z"
  shows " $\llbracket$ list_emb P xs ys; list_emb P ys zs $\rrbracket \implies$  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.prem1 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

```

8.9 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  $\implies$  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  $\neq$  y  $\implies$  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)  $\longleftrightarrow$  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  $\longleftrightarrow$  (subseq xs ys  $\wedge$   $\neg$ 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  $\in$  set (subseqs ys)  $\longleftrightarrow$  subseq xs ys"
proof
  assume "xs  $\in$  set (subseqs ys)"
  thus "subseq xs ys"
  by (induction ys arbitrary: xs) (auto simp: Let_def)
next
  have [simp]: "[ ]  $\in$  set (subseqs ys)" for ys :: "'a list"
  by (induction ys) (auto simp: Let_def)
  assume "subseq xs ys"
  thus "xs  $\in$  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  $\longleftrightarrow$  xs = [ ]"
  by (auto dest: list_emb_length)

```

```
lemma subseq_singleton_left: "subseq [x] ys  $\longleftrightarrow$  x  $\in$  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' ]  $\implies$  list_emb P (xs@ys) (xs'@ys' )"
  by (induct rule: list_emb.induct) auto
```

```
lemma prefix_imp_subseq [intro]: "prefix xs ys  $\implies$  subseq xs ys"
  by (auto simp: prefix_def)
```

```
lemma suffix_imp_subseq [intro]: "suffix xs ys  $\implies$  subseq xs ys"
  by (auto simp: suffix_def)
```

8.10 Appending elements

```
lemma subseq_append [simp]:
  "subseq (xs @ zs) (ys @ zs)  $\longleftrightarrow$  subseq xs ys" (is "?l = ?r")
proof
  { fix xs' ys' xs ys zs :: "'a list" assume "subseq xs' ys'"
    then have "xs' = xs @ zs  $\wedge$  ys' = ys @ zs  $\longrightarrow$  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  $\langle (=) x y \rangle$  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)  $\longleftrightarrow$  ( $\exists$  xs1 xs2. xs = xs1 @ xs2  $\wedge$  subseq xs1 ys  $\wedge$  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  $\implies$  subseq xs (zs @ ys)"
  by (induct zs) auto

lemma subseq_rev_drop_many: "subseq xs ys  $\implies$  subseq xs (ys @ zs)"
  by (metis append_Nil2 list_emb_Nil list_emb_append_mono)

```

8.11 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_nth:
  "subseq xs ys  $\longleftrightarrow$  ( $\exists N. xs = nth\ ys\ N$ )" (is "?L = ?R")
proof
  assume ?L
  then show ?R
  proof (induct)
    case list_emb_Nil show ?case by (metis nth_empty)
  next
    case (list_emb_Cons xs ys x)
    then obtain N where "xs = nth\ ys\ N" by blast
    then have "xs = nth\ (x#ys)\ (Suc ' N)"
      by (clarsimp simp add: nth_Cons inj_image_mem_iff)
    then show ?case by blast
  next
    case (list_emb_Cons2 x y xs ys)
    then obtain N where "xs = nth\ ys\ N" by blast
    then have "x#xs = nth\ (x#ys)\ (insert 0 (Suc ' N))"
      by (clarsimp simp add: nth_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 = nth\ ys\ N" ..
  moreover have "subseq (nth\ ys\ N) ys"
  proof (induct ys arbitrary: N)
    case Nil show ?case by simp
  next
    case Cons then show ?case by (auto simp: nth_Cons)
  qed
  ultimately show ?L by simp
qed

```

8.12 Contiguous sublists

```

definition sublist :: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool" where

```

```

"sublist xs ys = ( $\exists$  ps ss. ys = ps @ xs @ ss)"

definition strict_sublist :: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool" where
  "strict_sublist xs ys  $\longleftrightarrow$  sublist xs ys  $\wedge$  xs  $\neq$  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  $\longleftrightarrow$  (sublist xs ys  $\wedge$   $\neg$  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]: " $\neg$  sublist (x#xs) []"
  by (auto simp: sublist_def)

lemma sublist_Nil_right [simp]: "sublist xs []  $\longleftrightarrow$  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  $\longleftrightarrow$  ( $\exists$  ys'. prefix ys' ys  $\wedge$  suffix xs ys)"
proof safe
  assume "sublist xs ys"
  then obtain ps ss where "ys = ps @ xs @ ss" by (auto simp: sublist_def)
  thus " $\exists$  ys'. prefix ys' ys  $\wedge$  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  $\longleftrightarrow$  ( $\exists$  ys'. suffix ys' ys  $\wedge$  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])  $\longleftrightarrow$ 
    (x = y  $\wedge$  suffix xs ys  $\vee$  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])  $\longleftrightarrow$  suffix xs (ys @ [y])  $\vee$  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  $\implies$  subseq xs ys"
  by (auto simp: sublist_def)

```

8.13 Parametricity

```

context includes lifting_syntax
begin

```

```

private lemma prefix_primrec:
  "prefix = rec_list ( $\lambda$ xs. True) ( $\lambda$ x xs xsa ys.
    case ys of []  $\Rightarrow$  False | y # ys  $\Rightarrow$  x = y  $\wedge$  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 = ( $\lambda$ xs ys. rec_list ( $\lambda$ xs. xs = []) ( $\lambda$ y ys ysa xs. prefix xs (y # ys)  $\vee$  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 = ( $\lambda$ uu uua uuaa. rec_list ( $\lambda$ P xs. List.null xs) ( $\lambda$ y ys ysa P xs. case xs of []  $\Rightarrow$  True
    | x # xs  $\Rightarrow$  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

```

9 Infinite Sets and Related Concepts

```

theory Infinite_Set
  imports Main
begin

```

9.1 The set of natural numbers is infinite

```

lemma infinite_nat_iff_unbounded_le: "infinite S  $\longleftrightarrow$  ( $\forall m. \exists n \geq m. n \in S$ )"
  for S :: "nat set"
  using frequently_cofinite[of " $\lambda x. x \in S$ "]
  by (simp add: cofinite_eq_sequentially frequently_def eventually_sequentially)

lemma infinite_nat_iff_unbounded: "infinite S  $\longleftrightarrow$  ( $\forall m. \exists n > m. n \in S$ )"
  for S :: "nat set"
  using frequently_cofinite[of " $\lambda x. x \in S$ "]
  by (simp add: cofinite_eq_sequentially frequently_def eventually_at_top_dense)

lemma finite_nat_iff_bounded: "finite S  $\longleftrightarrow$  ( $\exists k. S \subseteq \{..<k\}$ )"
  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  $\longleftrightarrow$  ( $\exists k. S \subseteq \{.. k\}$ )"
  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  $\implies \exists k. S \subseteq \{..<k\}$ "
  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: " $\forall m > k. \exists n > m. n \in S \implies$  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)  $\implies$  R"
  by simp

lemma range_inj_infinite:
  fixes f :: "nat  $\Rightarrow$  '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

9.2 The set of integers is also infinite

```

lemma infinite_int_iff_infinite_nat_abs: "infinite S  $\longleftrightarrow$  infinite ((nat  $\circ$  abs) ' S)"
  for S :: "int set"
proof (unfold Not_eq_iff, rule iffI)
  assume "finite ((nat  $\circ$  abs) ' S)"
  then have "finite (nat ' (abs ' S))"
    by (simp add: image_image cong: image_cong)
  moreover have "inj_on nat (abs ' S)"
    by (rule inj_onI) auto
  ultimately have "finite (abs ' S)"
    by (rule finite_imageD)
  then show "finite S"
    by (rule finite_image_absD)
qed simp

proposition infinite_int_iff_unbounded_le: "infinite S  $\longleftrightarrow$  ( $\forall m. \exists n. |n| \geq m \wedge n \in S$ )"
  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  $\longleftrightarrow$  ( $\forall m. \exists n. |n| > m \wedge n \in S$ )"
  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  $\longleftrightarrow$  ( $\exists k. \text{abs ' } S \subseteq \{..<k\}$ )"
  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  $\longleftrightarrow$  ( $\exists k. \text{abs ' } S \subseteq \{.. k\}$ )"
  for S :: "int set"
  using infinite_int_iff_unbounded[of S] by (simp add: subset_eq) (metis not_le)

```

9.3 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]: " $\neg$  (INFM x. P x)  $\longleftrightarrow$  (MOST x.  $\neg$  P x)"
  by (rule not_frequently)

lemma not_MOST [simp]: " $\neg$  (MOST x. P x)  $\longleftrightarrow$  (INFM x.  $\neg$  P x)"
  by (rule not_eventually)

lemma INFM_const [simp]: "(INFM x::'a. P)  $\longleftrightarrow$  P  $\wedge$  infinite (UNIV::'a set)"
  by (simp add: frequently_const_iff)

lemma MOST_const [simp]: "(MOST x::'a. P)  $\longleftrightarrow$  P  $\vee$  finite (UNIV::'a set)"
  by (simp add: eventually_const_iff)

lemma INFM_imp_distrib: "(INFM x. P x  $\longrightarrow$  Q x)  $\longleftrightarrow$  ((MOST x. P x)  $\longrightarrow$  (INFM x. Q x))"
  by (rule frequently_imp_iff)

lemma MOST_imp_iff: "MOST x. P x  $\implies$  (MOST x. P x  $\longrightarrow$  Q x)  $\longleftrightarrow$  (MOST x. Q x)"
  by (auto intro: eventually_rev_mp eventually_mono)

lemma INFM_conjI: "INFM x. P x  $\implies$  MOST x. Q x  $\implies$  INFM x. P x  $\wedge$  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)  $\implies$  inj f  $\implies$  INFM x. P x"
  using finite_vimageI[of "{x. P x}" f] by (auto simp: frequently_cofinite)
```

```
lemma MOST_inj: "MOST x. P x  $\implies$  inj f  $\implies$  MOST x. P (f x)"
  using finite_vimageI[of "{x.  $\neg$  P x}" f] by (auto simp: eventually_cofinite)
```

Properties of quantifiers with singletons.

```
lemma not_INFM_eq [simp]:
  " $\neg$  (INFM x. x = a)"
  " $\neg$  (INFM x. a = x)"
  unfolding frequently_cofinite by simp_all
```

```
lemma MOST_neq [simp]:
  "MOST x. x  $\neq$  a"
  "MOST x. a  $\neq$  x"
  unfolding eventually_cofinite by simp_all
```

```
lemma INFM_neq [simp]:
  "(INFM x::'a. x  $\neq$  a)  $\longleftrightarrow$  infinite (UNIV::'a set)"
  "(INFM x::'a. a  $\neq$  x)  $\longleftrightarrow$  infinite (UNIV::'a set)"
  unfolding frequently_cofinite by simp_all
```

```
lemma MOST_eq [simp]:
  "(MOST x::'a. x = a)  $\longleftrightarrow$  finite (UNIV::'a set)"
  "(MOST x::'a. a = x)  $\longleftrightarrow$  finite (UNIV::'a set)"
  unfolding eventually_cofinite by simp_all
```

```
lemma MOST_eq_imp:
  "MOST x. x = a  $\longrightarrow$  P x"
  "MOST x. a = x  $\longrightarrow$  P x"
  unfolding eventually_cofinite by simp_all
```

Properties of quantifiers over the naturals.

```
lemma MOST_nat: "( $\forall_{\infty} n. P n$ )  $\longleftrightarrow$  ( $\exists m. \forall n > m. P n$ )"
  for P :: "nat  $\Rightarrow$  bool"
  by (auto simp add: eventually_cofinite finite_nat_iff_bounded_le subset_eq simp flip: not_le)
```

```
lemma MOST_nat_le: "( $\forall_{\infty} n. P n$ )  $\longleftrightarrow$  ( $\exists m. \forall n \geq m. P n$ )"
  for P :: "nat  $\Rightarrow$  bool"
  by (auto simp add: eventually_cofinite finite_nat_iff_bounded subset_eq simp flip: not_le)
```

```
lemma INFM_nat: "( $\exists_{\infty} n. P n$ )  $\longleftrightarrow$  ( $\forall m. \exists n > m. P n$ )"
  for P :: "nat  $\Rightarrow$  bool"
  by (simp add: frequently_cofinite infinite_nat_iff_unbounded)
```

```
lemma INFM_nat_le: "( $\exists_{\infty} n. P n$ )  $\longleftrightarrow$  ( $\forall m. \exists n \geq m. P n$ )"
  for P :: "nat  $\Rightarrow$  bool"
  by (simp add: frequently_cofinite infinite_nat_iff_unbounded_le)
```

```
lemma MOST_INFM: "infinite (UNIV::'a set)  $\implies$  MOST x::'a. P x  $\implies$  INFM x::'a. P x"
  by (simp add: eventually_frequently)
```

```
lemma MOST_Suc_iff: "(MOST n. P (Suc n))  $\longleftrightarrow$  (MOST n. P n)"
  by (simp add: cofinite_eq_sequentially)
```

```
lemma MOST_SucI: "MOST n. P n  $\implies$  MOST n. P (Suc n)"
  and MOST_SucD: "MOST n. P (Suc n)  $\implies$  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  $\longleftrightarrow$  infinite {x. P x}" by (fact frequently_cofinite)
lemma Alm_all_def: "Alm_all P  $\longleftrightarrow$   $\neg$  (INFM x.  $\neg$  P x)" by simp
lemma INFM_iff_infinite: "(INFM x. P x)  $\longleftrightarrow$  infinite {x. P x}" by (fact frequently_cofinite)
lemma MOST_iff_cofinite: "(MOST x. P x)  $\longleftrightarrow$  finite {x.  $\neg$  P x}" by (fact eventually_cofinite)
lemma INFM_EX: " $(\exists_{\infty} x. P x) \implies (\exists x. P x)$ " by (fact frequently_ex)
lemma ALL_MOST: " $\forall x. P x \implies \forall_{\infty} x. P x$ " by (fact always_eventually)
lemma INFM_mono: " $\exists_{\infty} x. P x \implies (\bigwedge x. P x \implies Q x) \implies \exists_{\infty} x. Q x$ " by (fact frequently_elim1)
lemma MOST_mono: " $\forall_{\infty} x. P x \implies (\bigwedge x. P x \implies Q x) \implies \forall_{\infty} x. Q x$ " by (fact eventually_mono)
lemma INFM_disj_distrib: " $(\exists_{\infty} x. P x \vee Q x) \longleftrightarrow (\exists_{\infty} x. P x) \vee (\exists_{\infty} x. Q x)$ " by (fact frequently_disj_iff)
lemma MOST_rev_mp: " $\forall_{\infty} x. P x \implies \forall_{\infty} x. P x \longrightarrow Q x \implies \forall_{\infty} x. Q x$ " by (fact eventually_rev_mp)
lemma MOST_conj_distrib: " $(\forall_{\infty} x. P x \wedge Q x) \longleftrightarrow (\forall_{\infty} x. P x) \wedge (\forall_{\infty} x. Q x)$ " by (fact eventually_conj_iff)
lemma MOST_conjI: "MOST x. P x  $\implies$  MOST x. Q x  $\implies$  MOST x. P x  $\wedge$  Q x" by (fact eventually_conj)
lemma INFM_finite_Bex_distrib: "finite A  $\implies$  (INFM y.  $\exists x \in A. P x y$ )  $\longleftrightarrow$  ( $\exists x \in A. \text{INFM } y. P x y$ )" by
(fact frequently_bex_finite_distrib)
lemma MOST_finite_Ball_distrib: "finite A  $\implies$  (MOST y.  $\forall x \in A. P x y$ )  $\longleftrightarrow$  ( $\forall x \in A. \text{MOST } y. P x y$ )" by
(fact eventually_ball_finite_distrib)
lemma INFM_E: "INFM x. P x  $\implies$  ( $\bigwedge x. P x \implies$  thesis)  $\implies$  thesis" by (fact frequentlyE)
lemma MOST_I: " $(\bigwedge x. P x) \implies \text{MOST } x. P x$ " by (rule eventuallyI)
lemmas MOST_iff_finiteNeg = MOST_iff_cofinite

```

9.4 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 \in s \wedge finite {s \in S. s < t} \wedge card {s \in S. s < t} = n).`

```

primrec (in wellorder) enumerate :: "'a set  $\Rightarrow$  nat  $\Rightarrow$  'a"
  where
    enumerate_0: "enumerate S 0 = (LEAST n. n  $\in$  S)"
    | enumerate_Suc: "enumerate S (Suc n) = enumerate (S - {LEAST n. n  $\in$  S}) n"

lemma enumerate_Suc': "enumerate S (Suc n) = enumerate (S - {enumerate S 0}) n"
  by simp

lemma enumerate_in_set: "infinite S  $\implies$  enumerate S n  $\in$  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  $\implies$  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  $\in$  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  $\implies$  infinite S  $\implies$  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  $\leq$  enumerate S n"
  using S
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have "n  $\leq$  enumerate S n" by simp
  also note enumerate_mono[of n "Suc n", OF _ (infinite S)]
  finally show ?case by simp
qed
```

```
lemma infinite_enumerate:
  assumes fS: "infinite S"
  shows " $\exists r::\text{nat} \Rightarrow \text{nat}. \text{strict\_mono } r \wedge (\forall n. r\ n \in S)$ "
  unfolding strict_mono_def
  using enumerate_in_set[OF fS] enumerate_mono[of _ _ S] fS by auto
```

```
lemma enumerate_Suc'':
  fixes S :: "'a::wellorder set"
  assumes "infinite S"
  shows "enumerate S (Suc n) = (LEAST s. s  $\in$  S  $\wedge$  enumerate S n < s)"
  using assms
proof (induct n arbitrary: S)
  case 0
  then have " $\forall s \in S. \text{enumerate } S\ 0 \leq s$ "
    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  $\in$  S"
  shows " $\exists n. \text{enumerate } S\ n = s$ "
  using s
proof (induct s rule: less_induct)
  case (less s)
  show ?case
  proof (cases " $\exists y \in S. y < s$ ")
    case True
    let ?y = "Max {s'  $\in$  S. s' < s}"
    from True have y: " $\bigwedge x. ?y < x \iff (\forall s' \in S. s' < s \implies s' < x)$ "
```

```

    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

```

10 Countable sets

```

theory Countable_Set
imports Countable Infinite_Set
begin

```

10.1 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

```

10.2 Enumerate a countable set

```

lemma countableE_infinite:
  assumes "countable S" "infinite S"

```



```

    obtains e :: "'a  $\Rightarrow$  nat" where "bij_betw e S UNIV"
proof -
  obtain f :: "'a  $\Rightarrow$  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))  $\circ$  f) S UNIV"
    by (rule bij_betw_trans)
  then show thesis ..
qed

lemma countable_infiniteE':
  assumes "countable A" "infinite A"
  obtains g where "bij_betw g (UNIV :: nat set) A"
  using bij_betw_inv[OF countableE_infinite[OF assms]] that by blast

lemma countable_enum_cases:
  assumes "countable S"
  obtains (finite) f :: "'a  $\Rightarrow$  nat" where "finite S" "bij_betw f S {..\Rightarrow 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  $\Rightarrow$  'a  $\Rightarrow$  nat" where
  "to_nat_on S = (SOME f. if finite S then bij_betw f S {..\Rightarrow nat  $\Rightarrow$  'a" where
  "from_nat_into S n = (if n  $\in$  to_nat_on S ' S then inv_into S (to_nat_on S) n else SOME s. s  $\in$  S)"

lemma to_nat_on_finite: "finite S  $\implies$  bij_betw (to_nat_on S) S {..\implies infinite S  $\implies$  bij_betw (to_nat_on S) S UNIV"
  using countableE_infinite unfolding to_nat_on_def
  by (intro someI2_ex[where Q=" $\lambda$ f. bij_betw f S UNIV"]) auto

lemma bij_betw_from_nat_into_finite: "finite S  $\implies$  bij_betw (from_nat_into S) {..\implies infinite S  $\implies$  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  $\Rightarrow$  '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  $\implies$  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  $\implies$  a  $\in$  A  $\implies$  b  $\in$  A  $\implies$  to_nat_on A a = to_nat_on A b  $\longleftrightarrow$  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  $\implies$  a  $\in$  A  $\implies$  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  $\implies$  A  $\subseteq$  range (from_nat_into A)"
  by (auto intro: from_nat_into_to_nat_on[symmetric])

lemma from_nat_into: "A  $\neq$  {}  $\implies$  from_nat_into A n  $\in$  A"
  unfolding from_nat_into_def by (metis equalsOI inv_into_into someI_ex)

lemma range_from_nat_into_subset: "A  $\neq$  {}  $\implies$  range (from_nat_into A)  $\subseteq$  A"
  using from_nat_into[of A] by auto

lemma range_from_nat_into[simp]: "A  $\neq$  {}  $\implies$  countable A  $\implies$  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  $\implies$  infinite A  $\implies$  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  $\implies$  infinite A  $\implies$   $\exists$  a  $\in$  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  $\in$  to_nat_on A ' A  $\implies$  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  $\implies$  infinite A  $\implies$  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  $\implies$  m  $\in$  to_nat_on A ' A  $\implies$  n  $\in$  to_nat_on A ' A  $\implies$ 
    from_nat_into A m = from_nat_into A n  $\longleftrightarrow$  m = n"
  by (subst to_nat_on_inj[symmetric, of A]) auto

lemma from_nat_into_inj_infinite[simp]:
  "countable A  $\implies$  infinite A  $\implies$  from_nat_into A m = from_nat_into A n  $\longleftrightarrow$  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  $\implies$  x  $\in$  A  $\implies$  i  $\in$  to_nat_on A ' A  $\implies$  x = from_nat_into A i  $\longleftrightarrow$  i = to_nat_on A x"
  by auto

lemma from_nat_into_surj: "countable A  $\implies$  a  $\in$  A  $\implies$   $\exists$  n. from_nat_into A n = a"
  by (rule exI[of _ "to_nat_on A a"]) simp

lemma from_nat_into_inject[simp]:
  "A  $\neq$  {}  $\implies$  countable A  $\implies$  B  $\neq$  {}  $\implies$  countable B  $\implies$  from_nat_into A = from_nat_into B  $\longleftrightarrow$  A = B"
  by (metis range_from_nat_into)

lemma inj_on_from_nat_into: "inj_on from_nat_into ({A. A  $\neq$  {}  $\wedge$  countable A})"
  unfolding inj_on_def by auto

```

10.3 Closure properties of countability

```

lemma countable_SIGMA[intro, simp]:
  "countable I  $\implies$  ( $\bigwedge i. i \in I \implies \text{countable } (A \ i)$ )  $\implies \text{countable } (\text{SIGMA } i : I. A \ i)$ "
  by (intro countableI' [of " $\lambda(i, a). (\text{to\_nat\_on } I \ i, \text{to\_nat\_on } (A \ i) \ a)$ "]) (auto simp: inj_on_def)

lemma countable_image[intro, simp]:
  assumes "countable A"
  shows "countable (f ` A)"
proof -
  obtain g :: "'a  $\Rightarrow$  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  $\subseteq$  A"
    by (auto intro: inj_on_inv_into inv_into_inv)
  ultimately show ?thesis
    by (blast dest: comp_inj_on subset_inj_on intro: countableI)
qed

lemma countable_image_inj_on: "countable (f ` A)  $\implies$  inj_on f A  $\implies$  countable A"
  by (metis countable_image the_inv_into_onto)

lemma countable_image_inj_Int_vimage:
  "[inj_on f S; countable A]  $\implies$  countable (S  $\cap$  f ` A)"
  by (meson countable_image_inj_on countable_subset image_subset_iff_subset_vimage inf_le2 inj_on_Int)

lemma countable_image_inj_gen:
  "[inj_on f S; countable A]  $\implies$  countable {x  $\in$  S. f x  $\in$  A}"
  using countable_image_inj_Int_vimage
  by (auto simp: vimage_def Collect_conj_eq)

lemma countable_image_inj_eq:
  "inj_on f S  $\implies$  countable(f ` S)  $\longleftrightarrow$  countable S"
  using countable_image_inj_on by blast

lemma countable_image_inj:
  "[countable A; inj f]  $\implies$  countable {x. f x  $\in$  A}"
  by (metis (mono_tags, lifting) countable_image_inj_eq countable_subset image_Collect_subsetI inj_on_inverseI
  the_inv_f_f)

lemma countable_UN[intro, simp]:
  fixes I :: "'i set" and A :: "'i  $\Rightarrow$  'a set"
  assumes I: "countable I"
  assumes A: " $\bigwedge i. i \in I \implies \text{countable } (A \ i)$ "
  shows "countable ( $\bigcup i \in I. A \ i$ )"
proof -
  have "( $\bigcup i \in I. A \ i$ ) = 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  $\implies$  countable B  $\implies$  countable (A  $\cup$  B)"
  by (rule countable_UN [of "{True, False}" " $\lambda \text{True} \Rightarrow A \mid \text{False} \Rightarrow B$ ", simplified])
  (simp split: bool.split)

lemma countable_Un_iff[simp]: "countable (A  $\cup$  B)  $\longleftrightarrow$  countable A  $\wedge$  countable B"
  by (metis countable_Un countable_subset inf_sup_ord(3,4))

lemma countable_Plus[intro, simp]:
  "countable A  $\implies$  countable B  $\implies$  countable (A  $\lt+>$  B)"
  by (simp add: Plus_def)

lemma countable_empty[intro, simp]: "countable {}"

```

```

by (blast intro: countable_finite)

lemma countable_insert[intro, simp]: "countable A  $\implies$  countable (insert a A)"
  using countable_Un[of "{a}" A] by (auto simp: countable_finite)

lemma countable_Int1[intro, simp]: "countable A  $\implies$  countable (A  $\cap$  B)"
  by (force intro: countable_subset)

lemma countable_Int2[intro, simp]: "countable B  $\implies$  countable (A  $\cap$  B)"
  by (blast intro: countable_subset)

lemma countable_INT[intro, simp]: " $i \in I \implies$  countable (A i)  $\implies$  countable ( $\bigcap_{i \in I} A i$ )"
  by (blast intro: countable_subset)

lemma countable_Diff[intro, simp]: "countable A  $\implies$  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  $\subseteq$  range f  $\implies$  countable (f  $^{-1}$  B)  $\implies$  countable B"
  by (metis Int_absorb2 countable_image image_vimage_eq)

lemma surj_countable_vimage: "surj f  $\implies$  countable (f  $^{-1}$  B)  $\implies$  countable B"
  by (metis countable_vimage top_greatest)

lemma countable_Collect[simp]: "countable A  $\implies$  countable {a  $\in$  A.  $\varphi$  a}"
  by (metis Collect_conj_eq Int_absorb Int_commute Int_def countable_Int1)

lemma countable_Image:
  assumes " $\bigwedge y. y \in Y \implies$  countable (X  $^{-1}$  {y})"
  assumes "countable Y"
  shows "countable (X  $^{-1}$  Y)"
proof -
  have "countable (X  $^{-1}$  ( $\bigcup_{y \in Y} \{y\}$ ))"
    unfolding Image_UN by (intro countable_UN assms)
  then show ?thesis by simp
qed

lemma countable_relpow:
  fixes X :: "'a rel"
  assumes Image_X: " $\bigwedge Y. \text{countable } Y \implies \text{countable } (X^{-1} Y)$ "
  assumes Y: "countable Y"
  shows "countable ((X  $^{**}$  i)  $^{-1}$  Y)"
  using Y by (induct i arbitrary: Y) (auto simp: relcomp_Image Image_X)

lemma countable_funpow:
  fixes f :: "'a set  $\Rightarrow$  'a set"
  assumes " $\bigwedge A. \text{countable } A \implies \text{countable } (f A)$ "
  and "countable A"
  shows "countable ((f  $^{**}$  n) A)"
by (induction n) (simp_all add: assms)

lemma countable_rtranc1:
  " $(\bigwedge Y. \text{countable } Y \implies \text{countable } (X^{-1} Y)) \implies \text{countable } Y \implies \text{countable } (X^{*}^{-1} Y)$ "
  unfolding rtranc1_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  $\mathbb{Z}$ "
  unfolding Ints_def by auto

```

```

lemma countable_rat: "countable  $\mathbb{Q}$ "
  unfolding Rats_def by auto

```

```

lemma Collect_finite_subset_eq_lists: "{A. finite A  $\wedge$  A  $\subseteq$  T} = set ' lists T"
  using finite_list by (auto simp: lists_eq_set)

```

```

lemma countable_Collect_finite_subset:
  "countable T  $\implies$  countable {A. finite A  $\wedge$  A  $\subseteq$  T}"
  unfolding Collect_finite_subset_eq_lists by auto

```

```

lemma countable_set_option [simp]: "countable (set_option x)"
  by (cases x) auto

```

10.4 Misc lemmas

```

lemma countable_subset_image:
  "countable B  $\wedge$  B  $\subseteq$  (f ' A)  $\longleftrightarrow$  ( $\exists$  A'. countable A'  $\wedge$  A'  $\subseteq$  A  $\wedge$  (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 ex_subset_image_inj:
  "( $\exists$  T. T  $\subseteq$  f ' S  $\wedge$  P T)  $\longleftrightarrow$  ( $\exists$  T. T  $\subseteq$  S  $\wedge$  inj_on f T  $\wedge$  P (f ' T))"
  by (auto simp: subset_image_inj)

```

```

lemma all_subset_image_inj:
  "( $\forall$  T. T  $\subseteq$  f ' S  $\longrightarrow$  P T)  $\longleftrightarrow$  ( $\forall$  T. T  $\subseteq$  S  $\wedge$  inj_on f T  $\longrightarrow$  P (f ' T))"
  by (metis subset_image_inj)

```

```

lemma ex_countable_subset_image_inj:
  "( $\exists$  T. countable T  $\wedge$  T  $\subseteq$  f ' S  $\wedge$  P T)  $\longleftrightarrow$ 
  ( $\exists$  T. countable T  $\wedge$  T  $\subseteq$  S  $\wedge$  inj_on f T  $\wedge$  P (f ' T))"
  by (metis countable_image_inj_eq subset_image_inj)

```

```

lemma all_countable_subset_image_inj:
  "( $\forall$  T. countable T  $\wedge$  T  $\subseteq$  f ' S  $\longrightarrow$  P T)  $\longleftrightarrow$  ( $\forall$  T. countable T  $\wedge$  T  $\subseteq$  S  $\wedge$  inj_on f T  $\longrightarrow$  P (f ' T))"
  by (metis countable_image_inj_eq subset_image_inj)

```

```

lemma ex_countable_subset_image:
  "( $\exists$  T. countable T  $\wedge$  T  $\subseteq$  f ' S  $\wedge$  P T)  $\longleftrightarrow$  ( $\exists$  T. countable T  $\wedge$  T  $\subseteq$  S  $\wedge$  P (f ' T))"

```

```

by (metis countable_subset_image)

lemma all_countable_subset_image:
  "( $\forall T. \text{countable } T \wedge T \subseteq f \text{ ' } S \longrightarrow P \ T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \longrightarrow P(f \text{ ' } T))"$ 
  by (metis countable_subset_image)

lemma countable_image_eq:
  " $\text{countable}(f \text{ ' } S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ' } S = f \text{ ' } T)"$ 
  by (metis countable_image countable_image_inj_eq order_refl subset_image_inj)

lemma countable_image_eq_inj:
  " $\text{countable}(f \text{ ' } S) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ' } S = f \text{ ' } T \wedge \text{inj\_on } f \ T)"$ 
  by (metis countable_image_inj_eq order_refl subset_image_inj)

lemma infinite_countable_subset':
  assumes X: " $\text{infinite } X$ " shows " $\exists C \subseteq X. \text{countable } C \wedge \text{infinite } C$ "
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: " $\text{countable } S$ "
  shows " $(\forall s \in S. P \ s) \longleftrightarrow (\forall n::\text{nat}. \text{from\_nat\_into } S \ n \in S \longrightarrow P \ (\text{from\_nat\_into } S \ n))"$ 
  using S[THEN subset_range_from_nat_into] by auto

lemma finite_sequence_to_countable_set:
  assumes "countable X"
  obtains F where " $\bigwedge i. F \ i \subseteq X$ " " $\bigwedge i. F \ i \subseteq F \ (\text{Suc } i)$ " " $\bigwedge i. \text{finite } (F \ i)$ " " $\bigcup i. F \ i = X$ "
proof -
  show thesis
    apply (rule that[of " $\lambda i. \text{if } X = \{\} \text{ then } \{\} \text{ else from\_nat\_into } X \text{ ' } \{..i\}$ "])
    apply (auto simp add: image_iff intro: from_nat_into split: if_splits)
    using asms from_nat_into_surj by (fastforce cong: image_cong)
qed

lemma transfer_countable[transfer_rule]:
  " $\text{bi\_unique } R \implies \text{rel\_fun } (\text{rel\_set } R) \ (=) \text{countable countable}$ "
  by (rule rel_funI, erule (1) bi_unique_rel_set_lemma)
  (auto dest: countable_image_inj_on)

```

10.5 Uncountable

abbreviation uncountable where

"uncountable A $\equiv \neg \text{countable } A$ "

```

lemma uncountable_def: " $\text{uncountable } A \longleftrightarrow A \neq \{\} \wedge \neg (\exists f::(\text{nat} \Rightarrow 'a). \text{range } f = A)"$ 
  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: " $\text{bij\_betw } f \ A \ B \implies \text{uncountable } B \implies \text{uncountable } A$ "
  unfolding bij_betw_def by (metis countable_image)

```

```

lemma uncountable_infinite: " $\text{uncountable } A \implies \text{infinite } A$ "
  by (metis countable_finite)

```

```

lemma uncountable_minus_countable:
  " $\text{uncountable } A \implies \text{countable } B \implies \text{uncountable } (A - B)"$ 
  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

```

11 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  $\implies x \in A \implies \text{Inf } A \leq x$ "
  assumes ccInf_greatest: "countable A  $\implies (\bigwedge x. x \in A \implies z \leq x) \implies z \leq \text{Inf } A$ "
  assumes ccSup_upper: "countable A  $\implies x \in A \implies x \leq \text{Sup } A$ "
  assumes ccSup_least: "countable A  $\implies (\bigwedge x. x \in A \implies x \leq z) \implies \text{Sup } A \leq z$ "
  assumes ccInf_empty [simp]: "Inf {} = top"
  assumes ccSup_empty [simp]: "Sup {} = bot"
begin

subclass bounded_lattice
proof
  fix a
  show "bot  $\leq a$ " by (auto intro: ccSup_least simp only: ccSup_empty [symmetric])
  show "a  $\leq$  top" by (auto intro: ccInf_greatest simp only: ccInf_empty [symmetric])
qed

lemma ccINF_lower: "countable A  $\implies i \in A \implies (\text{INF } i \in A. f\ i) \leq f\ i$ "
  using ccInf_lower [of "f ' A"] by simp

lemma ccINF_greatest: "countable A  $\implies (\bigwedge i. i \in A \implies u \leq f\ i) \implies u \leq (\text{INF } i \in A. f\ i)$ "
  using ccInf_greatest [of "f ' A"] by auto

lemma ccSUP_upper: "countable A  $\implies i \in A \implies f\ i \leq (\text{SUP } i \in A. f\ i)$ "
  using ccSup_upper [of "f ' A"] by simp

lemma ccSUP_least: "countable A  $\implies (\bigwedge i. i \in A \implies f\ i \leq u) \implies (\text{SUP } i \in A. f\ i) \leq u$ "
  using ccSup_least [of "f ' A"] by auto

lemma ccInf_lower2: "countable A  $\implies u \in A \implies u \leq v \implies \text{Inf } A \leq v$ "
  using ccInf_lower [of A u] by auto

lemma ccINF_lower2: "countable A  $\implies i \in A \implies f\ i \leq u \implies (\text{INF } i \in A. f\ i) \leq u$ "
  using ccInf_lower [of A i f] by auto

lemma ccSup_upper2: "countable A  $\implies u \in A \implies v \leq u \implies v \leq \text{Sup } A$ "
  using ccSup_upper [of A u] by auto

lemma ccSUP_upper2: "countable A  $\implies i \in A \implies u \leq f\ i \implies u \leq (\text{SUP } i \in A. f\ i)$ "
  using ccSup_upper [of A i f] by auto

lemma le_ccInf_iff: "countable A  $\implies b \leq \text{Inf } A \iff (\forall a \in A. b \leq a)$ "
  by (auto intro: ccInf_greatest dest: ccInf_lower)

lemma le_ccINF_iff: "countable A  $\implies u \leq (\text{INF } i \in A. f\ i) \iff (\forall i \in A. u \leq f\ i)$ "
  using le_ccInf_iff [of "f ' A"] by simp

```

```

lemma ccSup_le_iff: "countable A  $\implies$  Sup A  $\leq$  b  $\longleftrightarrow$  ( $\forall a \in A. a \leq b$ )"
  by (auto intro: ccSup_least dest: ccSup_upper)

lemma ccSUP_le_iff: "countable A  $\implies$  (SUP i  $\in$  A. f i)  $\leq$  u  $\longleftrightarrow$  ( $\forall i \in A. f i \leq u$ )"
  using ccSup_le_iff [of "f ' A"] by simp

lemma ccInf_insert [simp]: "countable A  $\implies$  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  $\implies$  (INF x  $\in$  insert a A. f x) = inf (f a) (Inf (f ' A))"
  unfolding image_insert by simp

lemma ccSup_insert [simp]: "countable A  $\implies$  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  $\implies$  (SUP x  $\in$  insert a A. f x) = sup (f a) (Sup (f ' A))"
  unfolding image_insert by simp

lemma ccINF_empty [simp]: "(INF x  $\in$  {}. f x) = top"
  unfolding image_empty by simp

lemma ccSUP_empty [simp]: "(SUP x  $\in$  {}. f x) = bot"
  unfolding image_empty by simp

lemma ccInf_superset_mono: "countable A  $\implies$  B  $\subseteq$  A  $\implies$  Inf A  $\leq$  Inf B"
  by (auto intro: ccInf_greatest ccInf_lower countable_subset)

lemma ccSup_subset_mono: "countable B  $\implies$  A  $\subseteq$  B  $\implies$  Sup A  $\leq$  Sup B"
  by (auto intro: ccSup_least ccSup_upper countable_subset)

lemma ccInf_mono:
  assumes [intro]: "countable B" "countable A"
  assumes " $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$ "
  shows "Inf A  $\leq$  Inf B"
proof (rule ccInf_greatest)
  fix b assume "b  $\in$  B"
  with assms obtain a where "a  $\in$  A" and "a  $\leq$  b" by blast
  from  $\langle a \in A \rangle$  have "Inf A  $\leq$  a" by (rule ccInf_lower[rotated]) auto
  with  $\langle a \leq b \rangle$  show "Inf A  $\leq$  b" by auto
qed auto

lemma ccINF_mono:
  "countable A  $\implies$  countable B  $\implies$  ( $\bigwedge m. m \in B \implies \exists n \in A. f n \leq g m$ )  $\implies$  (INF n  $\in$  A. f n)  $\leq$  (INF n  $\in$  B. g n)"
  using ccInf_mono [of "g ' B" "f ' A"] by auto

lemma ccSup_mono:
  assumes [intro]: "countable B" "countable A"
  assumes " $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$ "
  shows "Sup A  $\leq$  Sup B"
proof (rule ccSup_least)
  fix a assume "a  $\in$  A"
  with assms obtain b where "b  $\in$  B" and "a  $\leq$  b" by blast
  from  $\langle b \in B \rangle$  have "b  $\leq$  Sup B" by (rule ccSup_upper[rotated]) auto
  with  $\langle a \leq b \rangle$  show "a  $\leq$  Sup B" by auto
qed auto

lemma ccSUP_mono:
  "countable A  $\implies$  countable B  $\implies$  ( $\bigwedge n. n \in A \implies \exists m \in B. f n \leq g m$ )  $\implies$  (SUP n  $\in$  A. f n)  $\leq$  (SUP n  $\in$  B. g n)"
  using ccSup_mono [of "g ' B" "f ' A"] by auto

```



```

lemma ccINF_superset_mono:
  "countable A  $\implies$  B  $\subseteq$  A  $\implies$  ( $\bigwedge x. x \in B \implies f\ x \leq g\ x$ )  $\implies$  (INF x $\in$ A. f x)  $\leq$  (INF x $\in$ B. g x)"
  by (blast intro: ccINF_mono countable_subset dest: subsetD)

lemma ccSUP_subset_mono:
  "countable B  $\implies$  A  $\subseteq$  B  $\implies$  ( $\bigwedge x. x \in A \implies f\ x \leq g\ x$ )  $\implies$  (SUP x $\in$ A. f x)  $\leq$  (SUP x $\in$ B. g x)"
  by (blast intro: ccSUP_mono countable_subset dest: subsetD)

lemma less_eq_ccInf_inter: "countable A  $\implies$  countable B  $\implies$  sup (Inf A) (Inf B)  $\leq$  Inf (A  $\cap$  B)"
  by (auto intro: ccInf_greatest ccInf_lower)

lemma ccSup_inter_less_eq: "countable A  $\implies$  countable B  $\implies$  Sup (A  $\cap$  B)  $\leq$  inf (Sup A) (Sup B)"
  by (auto intro: ccSup_least ccSup_upper)

lemma ccInf_union_distrib: "countable A  $\implies$  countable B  $\implies$  Inf (A  $\cup$  B) = inf (Inf A) (Inf B)"
  by (rule antisym) (auto intro: ccInf_greatest ccInf_lower le_infI1 le_infI2)

lemma ccINF_union:
  "countable A  $\implies$  countable B  $\implies$  (INF i $\in$ A  $\cup$  B. M i) = inf (INF i $\in$ A. M i) (INF i $\in$ B. M i)"
  by (auto intro!: antisym ccINF_mono intro: le_infI1 le_infI2 ccINF_greatest ccINF_lower)

lemma ccSup_union_distrib: "countable A  $\implies$  countable B  $\implies$  Sup (A  $\cup$  B) = sup (Sup A) (Sup B)"
  by (rule antisym) (auto intro: ccSup_least ccSup_upper le_supI1 le_supI2)

lemma ccSUP_union:
  "countable A  $\implies$  countable B  $\implies$  (SUP i $\in$ A  $\cup$  B. M i) = sup (SUP i $\in$ A. M i) (SUP i $\in$ B. M i)"
  by (auto intro!: antisym ccSUP_mono intro: le_supI1 le_supI2 ccSUP_least ccSUP_upper)

lemma ccINF_inf_distrib: "countable A  $\implies$  inf (INF a $\in$ A. f a) (INF a $\in$ A. g a) = (INF a $\in$ 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  $\implies$  sup (SUP a $\in$ A. f a) (SUP a $\in$ A. g a) = (SUP a $\in$ 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  $\neq$  {}  $\implies$  (INF i  $\in$  A. f) = f"
  unfolding image_constant_conv by auto

lemma ccSUP_const [simp]: "A  $\neq$  {}  $\implies$  (SUP i  $\in$  A. f) = f"
  unfolding image_constant_conv by auto

lemma ccINF_top [simp]: "(INF x $\in$ A. top) = top"
  by (cases "A = {}") simp_all

lemma ccSUP_bot [simp]: "(SUP x $\in$ A. bot) = bot"
  by (cases "A = {}") simp_all

lemma ccINF_commute: "countable A  $\implies$  countable B  $\implies$  (INF i $\in$ A. INF j $\in$ B. f i j) = (INF j $\in$ B. INF i $\in$ A. f i j)"
  by (iprover intro: ccINF_lower ccINF_greatest order_trans antisym)

lemma ccSUP_commute: "countable A  $\implies$  countable B  $\implies$  (SUP i $\in$ A. SUP j $\in$ B. f i j) = (SUP j $\in$ B. SUP i $\in$ 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  $\implies$  a < Sup S  $\longleftrightarrow$  ( $\exists x \in S. a < x$ )"
  unfolding not_le [symmetric] by (subst ccSup_le_iff) auto

lemma less_ccSUP_iff: "countable A  $\implies$  a < (SUP i  $\in$  A. f i)  $\longleftrightarrow$  ( $\exists x \in A. a < f x$ )"
  using less_ccSup_iff [of "f ' A"] by simp

lemma ccInf_less_iff: "countable S  $\implies$  Inf S < a  $\longleftrightarrow$  ( $\exists x \in S. x < a$ )"
  unfolding not_le [symmetric] by (subst le_ccInf_iff) auto

lemma ccINF_less_iff: "countable A  $\implies$  (INF i  $\in$  A. f i) < a  $\longleftrightarrow$  ( $\exists x \in A. f x < a$ )"
  using ccInf_less_iff [of "f ' A"] by simp

end

class countable_complete_distrib_lattice = countable_complete_lattice +
  assumes sup_ccInf: "countable B  $\implies$  sup a (Inf B) = (INF b  $\in$  B. sup a b)"
  assumes inf_ccSup: "countable B  $\implies$  inf a (Sup B) = (SUP b  $\in$  B. inf a b)"
begin

lemma sup_ccINF:
  "countable B  $\implies$  sup a (INF b  $\in$  B. f b) = (INF b  $\in$  B. sup a (f b))"
  by (simp only: sup_ccInf image_image countable_image)

lemma inf_ccSUP:
  "countable B  $\implies$  inf a (SUP b  $\in$  B. f b) = (SUP b  $\in$  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  $\in$  {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  $\implies$  sup (Inf B) a = (INF b  $\in$  B. sup b a)"
  by (simp add: sup_ccInf sup_commute)

lemma ccSup_inf:
  "countable B  $\implies$  inf (Sup B) a = (SUP b  $\in$  B. inf b a)"
  by (simp add: inf_ccSup inf_commute)

lemma ccINF_sup:
  "countable B  $\implies$  sup (INF b  $\in$  B. f b) a = (INF b  $\in$  B. sup (f b) a)"
  by (simp add: sup_ccINF sup_commute)

lemma ccSUP_inf:
  "countable B  $\implies$  inf (SUP b  $\in$  B. f b) a = (SUP b  $\in$  B. inf (f b) a)"
  by (simp add: inf_ccSUP inf_commute)

lemma ccINF_sup_distrib2:
  "countable A  $\implies$  countable B  $\implies$  sup (INF a  $\in$  A. f a) (INF b  $\in$  B. g b) = (INF a  $\in$  A. INF b  $\in$  B. sup (f a) (g b))"
  by (subst ccINF_commute) (simp_all add: sup_ccINF ccINF_sup)

lemma ccSUP_inf_distrib2:

```

```

"countable A  $\implies$  countable B  $\implies$  inf (SUP a $\in$ A. f a) (SUP b $\in$ B. g b) = (SUP a $\in$ A. SUP b $\in$ B. inf (f a)
(g b))"
by (subst ccSUP_commute) (simp_all add: inf_ccSUP ccSUP_inf)

context
  fixes f :: "'a  $\Rightarrow$  'b::countable_complete_lattice"
  assumes "mono f"
begin

lemma mono_ccInf:
  "countable A  $\implies$  f (Inf A)  $\leq$  (INF x $\in$ 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  $\implies$  (SUP x $\in$ A. f x)  $\leq$  f (Sup A)"
  using (mono f) by (auto intro: countable_complete_lattice_class.ccSUP_least ccSup_upper dest: monoD)

lemma mono_ccINF:
  "countable I  $\implies$  f (INF i  $\in$  I. A i)  $\leq$  (INF x  $\in$  I. f (A x))"
  by (intro countable_complete_lattice_class.ccINF_greatest monoD[OF (mono f)] ccINF_lower)

lemma mono_ccSUP:
  "countable I  $\implies$  (SUP x  $\in$  I. f (A x))  $\leq$  f (SUP i  $\in$  I. A i)"
  by (intro countable_complete_lattice_class.ccSUP_least monoD[OF (mono f)] ccSUP_upper)

end

end

```

11.0.1 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

```

12 Continuity and iterations

```

theory Order_Continuity
imports Complex_Main Countable_Complete_Lattices
begin

```

```

lemma SUP_nat_binary:
  "(sup A (SUP x $\in$ Collect ((<) (0::nat)). B)) = (sup A B::'a::countable_complete_lattice)"
  apply (subst image_constant)
  apply auto
  done

lemma INF_nat_binary:
  "inf A (INF x $\in$ Collect ((<) (0::nat)). B) = (inf A B::'a::countable_complete_lattice)"

```

```

apply (subst image_constant)
apply auto
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`

12.1 Continuity for complete lattices

definition

```

sup_continuous :: "('a::countable_complete_lattice  $\Rightarrow$  'b::countable_complete_lattice)  $\Rightarrow$  bool"

```

where

```

"sup_continuous F  $\longleftrightarrow$  ( $\forall M::nat \Rightarrow$  'a. mono M  $\longrightarrow$  F (SUP i. M i) = (SUP i. F (M i)))"

```

```

lemma sup_continuousD: "sup_continuous F  $\Longrightarrow$  mono M  $\Longrightarrow$  F (SUP i::nat. M i) = (SUP i. F (M i))"
  by (auto simp: sup_continuous_def)

```

lemma sup_continuous_mono:

```

"mono F" if "sup_continuous F"

```

proof

```

fix A B :: "'a"
assume "A  $\leq$  B"
let ?f = " $\lambda n::nat$ . if n = 0 then A else B"
from (A  $\leq$  B) have "incseq ?f"
  by (auto intro: monoI)
with (sup_continuous F) have *: "F (SUP i. ?f i) = (SUP i. F (?f i))"
  by (auto dest: sup_continuousD)
from (A  $\leq$  B) have "B = sup A B"
  by (simp add: le_iff_sup)
then have "F B = F (sup A B)"
  by simp
also have "... = sup (F A) (F B)"
  using * by (simp add: if_distrib SUP_nat_binary cong del: SUP_cong)
finally show "F A  $\leq$  F B"
  by (simp add: le_iff_sup)

```

qed

lemma [order_continuous_intros]:

```

shows sup_continuous_const: "sup_continuous ( $\lambda x$ . c)"
and sup_continuous_id: "sup_continuous ( $\lambda x$ . x)"
and sup_continuous_apply: "sup_continuous ( $\lambda f$ . f x)"
and sup_continuous_fun: "( $\bigwedge s$ . sup_continuous ( $\lambda x$ . P x s))  $\Longrightarrow$  sup_continuous P"
and sup_continuous_if: "sup_continuous F  $\Longrightarrow$  sup_continuous G  $\Longrightarrow$  sup_continuous ( $\lambda f$ . if C then F f else G f)"
  by (auto simp: sup_continuous_def image_comp)

```

lemma sup_continuous_compose:

```

assumes f: "sup_continuous f" and g: "sup_continuous g"
shows "sup_continuous ( $\lambda x$ . f (g x))"
unfolding sup_continuous_def

```

proof safe

```

fix M :: "nat  $\Rightarrow$  'c"
assume M: "mono M"
then have "mono ( $\lambda i$ . g (M i))"
  using sup_continuous_mono[OF g] by (auto simp: mono_def)
with M show "f (g (Sup (M ' UNIV))) = (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  $\implies$  sup_continuous g  $\implies$  sup_continuous ( $\lambda x. \text{sup } (f \ x) \ (g \ x)$ )"
by (simp add: sup_continuous_def ccSUP_sup_distrib)

lemma sup_continuous_inf[order_continuous_intros]:
  fixes P Q :: "'a :: countable_complete_lattice  $\Rightarrow$  'b :: countable_complete_distrib_lattice"
  assumes P: "sup_continuous P" and Q: "sup_continuous Q"
  shows "sup_continuous ( $\lambda x. \text{inf } (P \ x) \ (Q \ x)$ )"
  unfolding sup_continuous_def
proof (safe intro!: antisym)
  fix M :: "nat  $\Rightarrow$  'a" assume M: "incseq M"
  have "inf (P (SUP i. M i)) (Q (SUP i. M i))  $\leq$  (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 "...  $\leq$  (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))  $\leq$  (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))  $\leq$  (SUP i. inf (P (M i)) (Q (M i)))" .

  show "(SUP i. inf (P (M i)) (Q (M i)))  $\leq$  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  $\implies$  sup_continuous Q  $\implies$  sup_continuous ( $\lambda x. P \ x \ \wedge \ Q \ x$ )"
  using sup_continuous_inf[of P Q] by simp

lemma sup_continuous_or[order_continuous_intros]:
  "sup_continuous P  $\implies$  sup_continuous Q  $\implies$  sup_continuous ( $\lambda x. P \ x \ \vee \ Q \ x$ )"
  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  $\leq$  lfp F"
  proof (rule SUP_least)
    fix i show "(F ^^ i) bot  $\leq$  lfp F"
    proof (induct i)
      case (Suc i)
      have "(F ^^ Suc i) bot = F ((F ^^ i) bot)" by simp
      also have "...  $\leq$  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  $\leq$  ?U"
proof (rule lfp_lowerbound)
  have "mono ( $\lambda i::\text{nat}. (F ^^ i) \text{bot}$ )"
  proof -
    { fix i::nat have "(F ^^ i) bot  $\leq$  (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  $\alpha \implies \text{sup\_continuous } f \implies \text{sup\_continuous } g \implies$ 
  ( $\bigwedge x. \alpha \text{ bot} \leq g x$ )  $\implies (\bigwedge x. x \leq \text{lfp } f \implies \alpha (f x) = g (\alpha x)) \implies \alpha (\text{lfp } f) = \text{lfp } g$ "
by (rule lfp_transfer_bounded[where P=top]) (auto dest: sup_continuousD)

definition
  inf_continuous :: "('a::countable_complete_lattice  $\Rightarrow$  'b::countable_complete_lattice)  $\Rightarrow$  bool"
where
  "inf_continuous F  $\longleftrightarrow (\forall M::\text{nat} \Rightarrow 'a. \text{antimono } M \longrightarrow F (\text{INF } i. M i) = (\text{INF } i. F (M i)))$ "

lemma inf_continuousD: "inf_continuous F  $\implies \text{antimono } M \implies F (\text{INF } i::\text{nat}. M i) = (\text{INF } i. F (M i))$ "
  by (auto simp: inf_continuous_def)

lemma inf_continuous_mono:
  "mono F" if "inf_continuous F"
proof
  fix A B :: "'a"
  assume "A  $\leq$  B"
  let ?f = " $\lambda n::\text{nat}. \text{if } n = 0 \text{ then } B \text{ else } A$ "
  from (A  $\leq$  B) have "decseq ?f"
    by (auto intro: antimonoI)
  with (inf_continuous F) have *: "F (INF i. ?f i) = (INF i. F (?f i))"
    by (auto dest: inf_continuousD)
  from (A  $\leq$  B) have "A = inf B A"
    by (simp add: inf.absorb_iff2)
  then have "F A = F (inf B A)"
    by simp
  also have "... = inf (F B) (F A)"
    using * by (simp add: if_distrib INF_nat_binary cong del: INF_cong)
  finally show "F A  $\leq$  F B"
    by (simp add: inf.absorb_iff2)
qed

lemma [order_continuous_intros]:
  shows inf_continuous_const: "inf_continuous ( $\lambda x. c$ )"
    and inf_continuous_id: "inf_continuous ( $\lambda x. x$ )"
    and inf_continuous_apply: "inf_continuous ( $\lambda f. f x$ )"
    and inf_continuous_fun: " $(\bigwedge s. \text{inf\_continuous } (\lambda x. P x s)) \implies \text{inf\_continuous } P$ "
    and inf_continuous_if: "inf_continuous F  $\implies \text{inf\_continuous } G \implies \text{inf\_continuous } (\lambda f. \text{if } C \text{ then } F f \text{ else } G f)$ "
  by (auto simp: inf_continuous_def image_comp)

lemma inf_continuous_inf[order_continuous_intros]:
  "inf_continuous f  $\implies \text{inf\_continuous } g \implies \text{inf\_continuous } (\lambda x. \text{inf } (f x) (g x))$ "
  by (simp add: inf_continuous_def ccINF_inf_distrib)

lemma inf_continuous_sup[order_continuous_intros]:
  fixes P Q :: "'a :: countable_complete_lattice  $\Rightarrow$  'b :: countable_complete_distrib_lattice"
  assumes P: "inf_continuous P" and Q: "inf_continuous Q"
  shows "inf_continuous ( $\lambda x. \text{sup } (P x) (Q x)$ )"
  unfolding inf_continuous_def
proof (safe intro!: antisym)
  fix M :: "nat  $\Rightarrow$  'a" assume M: "decseq M"
  show "sup (P (INF i. M i)) (Q (INF i. M i))  $\leq$  (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)))  $\leq$  (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))  $\geq$  (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)
  end
end

```

```

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 (Inf (M ' UNIV))) = (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  $\alpha$ : "inf_continuous  $\alpha$ " and f: "inf_continuous f" and g: "inf_continuous g"
  assumes [simp]: " $\alpha$  top = top" " $\bigwedge x. \alpha (f x) = g (\alpha x)$ "
  shows " $\alpha (gfp f) = gfp g$ "
proof -
  have " $\alpha (gfp f) = (INF i. \alpha ((f^{^i}) top))$ "
    unfolding inf_continuous_gfp[OF f] by (intro f  $\alpha$  inf_continuousD antimono_funpow inf_continuous_mono)
  moreover have " $\alpha ((f^{^i}) top) = (g^{^i}) top$ " 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)" " $\bigwedge x. P x \implies P (f x)$ " " $\bigwedge M. antimono M \implies (\bigwedge i. P (M i)) \implies P (INF i::nat. M i)$ "
  assumes  $\alpha$ : " $\bigwedge M. antimono M \implies (\bigwedge i::nat. P (M i)) \implies \alpha (INF i. M i) = (INF i. \alpha (M i))$ "
  assumes f: "inf_continuous f" and g: "inf_continuous g"
  assumes [simp]: " $\bigwedge x. P x \implies \alpha (f x) = g (\alpha x)$ "
  assumes g_bound: " $\bigwedge x. g x \leq \alpha (f top)$ "
  shows " $\alpha (gfp f) = gfp g$ "
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 ( $\lambda i. (f ^ i) top$ )"
    unfolding antimono_iff_le_Suc
  proof
    fix i show "(f ^ Suc i) top  $\leq$  (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 ( $\lambda i. (f ^ i) (f top)$ )"
  proof
    show " $x \leq y \implies (f ^ y) (f top) \leq (f ^ x) (f top)$ " 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 " $\exists j \in UNIV. (f ^ j) (f top) \leq (f ^ i) top$ " for i
      by (intro bexI[of _ "i - 1"]) (auto simp: diff_Suc funpow_Suc_right simp del: funpow.simps(2)
split: nat.split)
    show " $\exists j \in UNIV. (f ^ j) top \leq (f ^ i) (f top)$ " 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 " $\alpha (gfp f) = (INF i. \alpha ((f^{^i}) (f top)))$ "
    unfolding gfp_f by (rule  $\alpha$ ) (auto intro!: P_pow antimono_pow2)
  also have "...  $\geq gfp g$ "
  proof (rule INF_greatest)
    fix i show "gfp g  $\leq \alpha ((f^{^i}) (f top))$ "

```

```

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

```

12.1.1 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) (simp add: image_comp)
  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
```

13 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
```

13.1 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"

```

13.2 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 il_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])

```

13.3 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)

```

13.4 Multiplication

```

instantiation enat :: "{comm_semiring_1, semiring_no_zero_divisors}"
begin

```

```

definition times_enat_def [nitpick_simp]:
  "m * n = (case m of  $\infty \Rightarrow$  if n = 0 then 0 else  $\infty$  | enat m  $\Rightarrow$ 
    (case n of  $\infty \Rightarrow$  if m = 0 then 0 else  $\infty$  | enat n  $\Rightarrow$  enat (m * n)))"

```

```

lemma times_enat_simps [simp, code]:
  "enat m * enat n = enat (m * n)"
  " $\infty$  *  $\infty$  = ( $\infty$ ::enat)"
  " $\infty$  * enat n = (if n = 0 then 0 else  $\infty$ )"
  "enat m *  $\infty$  = (if m = 0 then 0 else  $\infty$ )"
  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  $\neq$  0  $\Rightarrow$  b  $\neq$  0  $\Rightarrow$  a * b  $\neq$  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)

```

13.5 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)

```

13.6 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)

```

13.7 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 add_diff_assoc_enat: "z ≤ y ⇒ x + (y - z) = x + y - (z::enat)"
by (cases x) (auto simp add: diff_enat_def split: enat.split)
```

```
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 =  $\infty$ "

instance
  by standard (simp_all add: bot_enat_def top_enat_def)

```

end

```

lemma finite_enat_bounded:
  assumes le_fin: " $\bigwedge y. y \in A \implies y \leq \text{enat } n$ "
  shows "finite A"
proof (rule finite_subset)
  show "finite (enat ' {..n})" by blast
  have "A  $\subseteq$  {..enat n}" using le_fin by fastforce
  also have "...  $\subseteq$  enat ' {..n}"
    apply (rule subsetI)
    subgoal for x by (cases x) auto
  done
  finally show "A  $\subseteq$  enat ' {..n}" .
qed

```

13.8 Cancellation simprocs

```

lemma add_diff_cancel_enat[simp]: " $x \neq \infty \implies x + y - x = (y::\text{enat})$ "
by (metis add.commute add.right_neutral add_diff_assoc_enat idiff_self order_refl)

```

```

lemma enat_add_left_cancel: " $a + b = a + c \longleftrightarrow a = (\infty::\text{enat}) \vee b = c$ "
unfolding plus_enat_def by (simp split: enat.split)

```

```

lemma enat_add_left_cancel_le: " $a + b \leq a + c \longleftrightarrow a = (\infty::\text{enat}) \vee b \leq c$ "
unfolding plus_enat_def by (simp split: enat.split)

```

```

lemma enat_add_left_cancel_less: " $a + b < a + c \longleftrightarrow a \neq (\infty::\text{enat}) \wedge b < c$ "
unfolding plus_enat_def by (simp split: enat.split)

```

```

lemma plus_eq_infty_iff_enat: " $(m::\text{enat}) + n = \infty \longleftrightarrow m = \infty \vee n = \infty$ "
using enat_add_left_cancel by fastforce

```

```

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) = HLogic.mk_Trueprop (HLogic.mk_eq (a, b))
  end

structure Eq_Enat_Cancel = ExtractCommonTermFun
(open Cancel_Enat_Common
  val mk_bal = HLogic.mk_eq
  val dest_bal = HLogic.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 = HLogic.mk_binrel const_name Orderings.less_eq
  val dest_bal = HLogic.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 = HLogic.mk_binrel const_name Orderings.less
  val dest_bal = HLogic.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

```

13.9 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  $\infty$ "
  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: " $(\bigwedge n::\text{enat}. (\bigwedge m::\text{enat}. m < n \implies P m) \implies P n)$ "
  show "P n" by (blast intro: enat_less_induct hyp)
qed

```

13.10 Complete Lattice

```

instantiation enat :: complete_lattice
begin

definition inf_enat :: "enat  $\Rightarrow$  enat  $\Rightarrow$  enat" where
  "inf_enat = min"

definition sup_enat :: "enat  $\Rightarrow$  enat  $\Rightarrow$  enat" where
  "sup_enat = max"

definition Inf_enat :: "enat set  $\Rightarrow$  enat" where
  "Inf_enat A = (if A = {} then  $\infty$  else (LEAST x. x  $\in$  A))"

definition Sup_enat :: "enat set  $\Rightarrow$  enat" where
  "Sup_enat A = (if A = {} then 0 else if finite A then Max A else  $\infty$ )"

instance
proof
  fix x :: "enat" and A :: "enat set"
  { assume "x  $\in$  A" then show "Inf A  $\leq$  x"
    unfolding Inf_enat_def by (auto intro: Least_le) }
  { assume " $\bigwedge y. y \in A \implies x \leq y$ " then show "x  $\leq$  Inf A"
    unfolding Inf_enat_def
    by (cases "A = {}") (auto intro: LeastI2_ex) }
  { assume "x  $\in$  A" then show "x  $\leq$  Sup A"
    unfolding Sup_enat_def by (cases "finite A") auto }
  { assume " $\bigwedge y. y \in A \implies y \leq x$ " then show "Sup A  $\leq$  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  $\neq$  {}  $\implies$  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  $\implies$  sup_continuous ( $\lambda x. \text{eSuc } (f x)$ )"
  using eSuc_Sup [of "_ ' UNIV"] by (auto simp: sup_continuous_def image_comp)

```

13.11 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
```

14 Linear Temporal Logic on Streams

```
theory Linear_Temporal_Logic_on_Streams
  imports Stream Sublist Extended_Nat Infinite_Set
begin
```

15 Preliminaries

```
lemma shift_prefix:
  assumes "x1 @- xs = y1 @- ys" and "length x1 ≤ length y1"
  shows "prefix x1 y1"
  using assms proof(induct x1 arbitrary: y1 xs ys)
    case (Cons x x1 y1 xs ys)
    thus ?case by (cases y1) auto
  qed auto

lemma shift_prefix_cases:
  assumes "x1 @- xs = y1 @- ys"
  shows "prefix x1 y1 ∨ prefix y1 x1"
  using shift_prefix[OF assms]
  by (cases "length x1 ≤ length y1") (metis, metis assms nat_le_linear shift_prefix)
```

16 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  $\varphi \equiv \lambda xs. \neg \varphi xs$ "

abbreviation (input) "true  $\equiv \lambda xs. \text{True}$ "

abbreviation (input) "false  $\equiv \lambda xs. \text{False}$ "

lemma impl_not_or: " $\varphi \text{ impl } \psi = (\text{not } \varphi) \text{ or } \psi$ "
by blast

lemma not_or: " $\text{not } (\varphi \text{ or } \psi) = (\text{not } \varphi) \text{ aand } (\text{not } \psi)$ "
by blast

lemma not_aand: " $\text{not } (\varphi \text{ aand } \psi) = (\text{not } \varphi) \text{ or } (\text{not } \psi)$ "
by blast

lemma non_not[simp]: " $\text{not } (\text{not } \varphi) = \varphi$ " by simp

  Temporal (LTL) connectives:
fun holds where "holds  $P xs \longleftrightarrow P (\text{shd } xs)$ "
fun nxt where " $\text{nxt } \varphi xs = \varphi (\text{stl } xs)$ "

definition "HLD  $s = \text{holds } (\lambda x. x \in s)$ "

abbreviation HLD_nxt (infixr "." 65) where
  " $s \cdot P \equiv \text{HLD } s \text{ aand } \text{nxt } P$ "

context
  notes [[inductive_internals]]
begin

inductive ev for  $\varphi$  where
base: " $\varphi xs \implies \text{ev } \varphi xs$ "
|
step: " $\text{ev } \varphi (\text{stl } xs) \implies \text{ev } \varphi xs$ "

coinductive alw for  $\varphi$  where
alw: " $\llbracket \varphi xs; \text{alw } \varphi (\text{stl } xs) \rrbracket \implies \text{alw } \varphi xs$ "

— weak until:
coinductive UNTIL (infix "until" 60) for  $\varphi \psi$  where
base: " $\psi xs \implies (\varphi \text{ until } \psi) xs$ "
|
step: " $\llbracket \varphi xs; (\varphi \text{ until } \psi) (\text{stl } xs) \rrbracket \implies (\varphi \text{ until } \psi) xs$ "

end

lemma holds_mono:
assumes holds: "holds  $P xs$ " and 0: " $\bigwedge x. P x \implies Q x$ "
shows "holds  $Q xs$ "
using assms by auto

lemma holds_aand:
"(holds  $P$  aand holds  $Q$ ) steps  $\longleftrightarrow \text{holds } (\lambda \text{step}. P \text{ step} \wedge Q \text{ step}) \text{ steps}$ " by auto

lemma HLD_iff: " $\text{HLD } s \omega \longleftrightarrow \text{shd } \omega \in s$ "
by (simp add: HLD_def)

lemma HLD_Stream[simp]: " $\text{HLD } X (x \## \omega) \longleftrightarrow x \in X$ "
by (simp add: HLD_iff)

lemma nxt_mono:

```

```

assumes nxt: "nxt  $\varphi$  xs" and 0: " $\bigwedge$  xs.  $\varphi$  xs  $\implies$   $\psi$  xs"
shows "nxt  $\psi$  xs"
using assms by auto

declare ev.intros[intro]
declare alw.cases[elim]

lemma ev_induct_strong[consumes 1, case_names base step]:
  "ev  $\varphi$  x  $\implies$  ( $\bigwedge$  xs.  $\varphi$  xs  $\implies$  P xs)  $\implies$  ( $\bigwedge$  xs. ev  $\varphi$  (stl xs)  $\implies$   $\neg$   $\varphi$  xs  $\implies$  P (stl xs)  $\implies$  P xs)
 $\implies$  P x"
  by (induct rule: ev.induct) auto

lemma alw_coinduct[consumes 1, case_names alw stl]:
  "X x  $\implies$  ( $\bigwedge$  x. X x  $\implies$   $\varphi$  x)  $\implies$  ( $\bigwedge$  x. X x  $\implies$   $\neg$  alw  $\varphi$  (stl x)  $\implies$  X (stl x))  $\implies$  alw  $\varphi$  x"
  using alw.coinduct[of X x  $\varphi$ ] by auto

lemma ev_mono:
assumes ev: "ev  $\varphi$  xs" and 0: " $\bigwedge$  xs.  $\varphi$  xs  $\implies$   $\psi$  xs"
shows "ev  $\psi$  xs"
using ev by induct (auto simp: 0)

lemma alw_mono:
assumes alw: "alw  $\varphi$  xs" and 0: " $\bigwedge$  xs.  $\varphi$  xs  $\implies$   $\psi$  xs"
shows "alw  $\psi$  xs"
using alw by coinduct (auto simp: 0)

lemma until_monol:
assumes until: " $(\varphi$  until  $\psi)$  xs" and 0: " $\bigwedge$  xs.  $\varphi$ 1 xs  $\implies$   $\varphi$ 2 xs"
shows " $(\varphi$ 2 until  $\psi)$  xs"
using until by coinduct (auto elim: UNTIL.cases simp: 0)

lemma until_monor:
assumes until: " $(\varphi$  until  $\psi$ 1) xs" and 0: " $\bigwedge$  xs.  $\psi$ 1 xs  $\implies$   $\psi$ 2 xs"
shows " $(\varphi$  until  $\psi$ 2) xs"
using until by coinduct (auto elim: UNTIL.cases simp: 0)

lemma until_mono:
assumes until: " $(\varphi$ 1 until  $\psi$ 1) xs" and
0: " $\bigwedge$  xs.  $\varphi$ 1 xs  $\implies$   $\varphi$ 2 xs" " $\bigwedge$  xs.  $\psi$ 1 xs  $\implies$   $\psi$ 2 xs"
shows " $(\varphi$ 2 until  $\psi$ 2) xs"
using until by coinduct (auto elim: UNTIL.cases simp: 0)

lemma until_false: " $\varphi$  until false = alw  $\varphi$ "
proof-
  {fix xs assume " $(\varphi$  until false) xs" hence "alw  $\varphi$  xs"
    by coinduct (auto elim: UNTIL.cases)
  }
  moreover
  {fix xs assume "alw  $\varphi$  xs" hence " $(\varphi$  until false) xs"
    by coinduct auto
  }
  ultimately show ?thesis by blast
qed

lemma ev_nxt: "ev  $\varphi$  = ( $\varphi$  or nxt (ev  $\varphi$ ))"
by (rule ext) (metis ev.simps nxt.simps)

lemma alw_nxt: "alw  $\varphi$  = ( $\varphi$  aand nxt (alw  $\varphi$ ))"
by (rule ext) (metis alw.simps nxt.simps)

lemma ev_ev[simp]: "ev (ev  $\varphi$ ) = ev  $\varphi$ "

```



```

proof-
  {fix xs
   assume "ev (ev  $\varphi$ ) xs" hence "ev  $\varphi$  xs"
   by induct auto
  }
  thus ?thesis by auto
qed

lemma alw_alw[simp]: "alw (alw  $\varphi$ ) = alw  $\varphi$ "
proof-
  {fix xs
   assume "alw  $\varphi$  xs" hence "alw (alw  $\varphi$ ) xs"
   by coinduct auto
  }
  thus ?thesis by auto
qed

lemma ev_shift:
  assumes "ev  $\varphi$  xs"
  shows "ev  $\varphi$  (xl @- xs)"
  using assms by (induct xl) auto

lemma ev_imp_shift:
  assumes "ev  $\varphi$  xs" shows " $\exists$  xl xs2. xs = xl @- xs2  $\wedge$   $\varphi$  xs2"
  using assms by induct (metis shift.simps(1), metis shift.simps(2) stream.collapse)+

lemma alw_ev_shift: "alw  $\varphi$  xs1  $\implies$  ev (alw  $\varphi$ ) (xl @- xs1)"
  by (auto intro: ev_shift)

lemma alw_shift:
  assumes "alw  $\varphi$  (xl @- xs)"
  shows "alw  $\varphi$  xs"
  using assms by (induct xl) auto

lemma ev_ex_nxt:
  assumes "ev  $\varphi$  xs"
  shows " $\exists$  n. (nxt  $^{\wedge}$  n)  $\varphi$  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  $^{\wedge}$  n)  $\varphi$  (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  $\varphi$  xs" shows "alw  $\varphi$  (sdrop n xs)"
  by (metis alw_shift assms stake_sdrop)

lemma nxt_sdrop: "(nxt  $^{\wedge}$  n)  $\varphi$  xs  $\longleftrightarrow$   $\varphi$  (sdrop n xs)"
  by (induct n arbitrary: xs) auto

definition "wait  $\varphi$  xs  $\equiv$  LEAST n. (nxt  $^{\wedge}$  n)  $\varphi$  xs"

lemma nxt_wait:
  assumes "ev  $\varphi$  xs" shows "(nxt  $^{\wedge}$  (wait  $\varphi$  xs))  $\varphi$  xs"
  unfolding wait_def using ev_ex_nxt[OF assms] by (rule LeastI_ex)

lemma nxt_wait_least:
  assumes ev: "ev  $\varphi$  xs" and nxt: "(nxt  $^{\wedge}$  n)  $\varphi$  xs" shows "wait  $\varphi$  xs  $\leq$  n"
  unfolding wait_def using ev_ex_nxt[OF ev] by (metis Least_le nxt)

```

```

lemma sdrop_wait:
  assumes "ev  $\varphi$  xs" shows " $\varphi$  (sdrop (wait  $\varphi$  xs) xs)"
  using nxt_wait[OF assms] unfolding nxt_sdrop .

lemma sdrop_wait_least:
  assumes ev: "ev  $\varphi$  xs" and nxt: " $\varphi$  (sdrop n xs)" shows "wait  $\varphi$  xs  $\leq$  n"
  using assms nxt_wait_least unfolding nxt_sdrop by auto

lemma nxt_ev: "(nxt  $\hat{\phantom{x}}$  n)  $\varphi$  xs  $\implies$  ev  $\varphi$  xs"
  by (induct n arbitrary: xs) auto

lemma not_ev: "not (ev  $\varphi$ ) = alw (not  $\varphi$ )"
  proof(rule ext, safe)
    fix xs assume "not (ev  $\varphi$ ) xs" thus "alw (not  $\varphi$ ) xs"
    by (coinduct) auto
  next
    fix xs assume "ev  $\varphi$  xs" and "alw (not  $\varphi$ ) xs" thus False
    by (induct) auto
  qed

lemma not_alw: "not (alw  $\varphi$ ) = ev (not  $\varphi$ )"
  proof-
    have "not (alw  $\varphi$ ) = not (alw (not (not  $\varphi$ )))" by simp
    also have "... = ev (not  $\varphi$ )" unfolding not_ev[symmetric] by simp
    finally show ?thesis .
  qed

lemma not_ev_not[simp]: "not (ev (not  $\varphi$ )) = alw  $\varphi$ "
  unfolding not_ev by simp

lemma not_alw_not[simp]: "not (alw (not  $\varphi$ )) = ev  $\varphi$ "
  unfolding not_alw by simp

lemma alw_ev_sdrop:
  assumes "alw (ev  $\varphi$ ) (sdrop m xs)"
  shows "alw (ev  $\varphi$ ) 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  $\varphi$ ) xs" shows "alw (ev  $\varphi$ ) xs"
  using assms by induct (metis (full_types) alw_mono ev.base, metis alw alw_nxt ev.step)

lemma alw_aand: "alw ( $\varphi$  aand  $\psi$ ) = alw  $\varphi$  aand alw  $\psi$ "
  proof-
    {fix xs assume "alw ( $\varphi$  aand  $\psi$ ) xs" hence "(alw  $\varphi$  aand alw  $\psi$ ) xs"
      by (auto elim: alw_mono)
    }
    moreover
    {fix xs assume "(alw  $\varphi$  aand alw  $\psi$ ) xs" hence "alw ( $\varphi$  aand  $\psi$ ) xs"
      by coinduct auto
    }
    ultimately show ?thesis by blast
  qed

lemma ev_or: "ev ( $\varphi$  or  $\psi$ ) = ev  $\varphi$  or ev  $\psi$ "
  proof-
    {fix xs assume "(ev  $\varphi$  or ev  $\psi$ ) xs" hence "ev ( $\varphi$  or  $\psi$ ) xs"
      by (auto elim: ev_mono)
    }
  qed

```

```

    moreover
    {fix xs assume "ev ( $\varphi$  or  $\psi$ ) xs" hence "(ev  $\varphi$  or ev  $\psi$ ) xs"
    by induct auto
  }
  ultimately show ?thesis by blast
qed

lemma ev_alw_aand:
  assumes  $\varphi$ : "ev (alw  $\varphi$ ) xs" and  $\psi$ : "ev (alw  $\psi$ ) xs"
  shows "ev (alw ( $\varphi$  aand  $\psi$ )) xs"
proof-
  obtain x1 xs1 where xs1: "xs = x1 @- xs1" and  $\varphi\varphi$ : "alw  $\varphi$  xs1"
  using  $\varphi$  by (metis ev_imp_shift)
  moreover obtain y1 ys1 where xs2: "xs = y1 @- ys1" and  $\psi\psi$ : "alw  $\psi$  ys1"
  using  $\psi$  by (metis ev_imp_shift)
  ultimately have 0: "x1 @- xs1 = y1 @- ys1" by auto
  hence "prefix x1 y1  $\vee$  prefix y1 x1" using shift_prefix_cases by auto
  thus ?thesis proof
    assume "prefix x1 y1"
    then obtain y11 where y1: "y1 = x1 @ y11" by (elim prefixE)
    have xs1': "xs1 = y11 @- xs1" using 0 unfolding y1 by simp
    have "alw  $\varphi$  ys1" using  $\varphi\varphi$  unfolding xs1' by (metis alw_shift)
    hence "alw ( $\varphi$  aand  $\psi$ ) ys1" using  $\psi\psi$  unfolding alw_aand by auto
    thus ?thesis unfolding xs2 by (auto intro: alw_ev_shift)
  next
    assume "prefix y1 x1"
    then obtain x11 where x1: "x1 = y1 @ x11" by (elim prefixE)
    have ys1': "ys1 = x11 @- xs1" using 0 unfolding x1 by simp
    have "alw  $\psi$  xs1" using  $\psi\psi$  unfolding ys1' by (metis alw_shift)
    hence "alw ( $\varphi$  aand  $\psi$ ) xs1" using  $\varphi\varphi$  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  $\varphi$ ) xs" and "alw (alw  $\varphi$  impl ev  $\psi$ ) xs"
  shows "ev  $\psi$  xs"
  using assms by induct auto

lemma ev_alw_stl[simp]: "ev (alw  $\varphi$ ) (stl x)  $\longleftrightarrow$  ev (alw  $\varphi$ ) x"
  by (metis (full_types) alw_nxt ev_nxt nxt.simps)

lemma alw_alw_impl_ev:
  "alw (alw  $\varphi$  impl ev  $\psi$ ) = (ev (alw  $\varphi$ ) impl alw (ev  $\psi$ ))" (is "?A = ?B")
proof-
  {fix xs assume "?A xs  $\wedge$  ev (alw  $\varphi$ ) xs" hence "alw (ev  $\psi$ ) 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  $\varphi$  xs" and "alw ( $\varphi$  impl  $\psi$ ) xs" shows "ev  $\psi$  xs"
  using assms by induct auto

lemma ev_alw_impl_ev:
  assumes "ev  $\varphi$  xs" and "alw ( $\varphi$  impl ev  $\psi$ ) xs" shows "ev  $\psi$  xs"

```

```

using ev_alw_impl[OF assms] by simp

lemma alw_mp:
  assumes "alw  $\varphi$  xs" and "alw ( $\varphi$  impl  $\psi$ ) xs"
  shows "alw  $\psi$  xs"
proof-
  {assume "alw  $\varphi$  xs  $\wedge$  alw ( $\varphi$  impl  $\psi$ ) xs" hence ?thesis
    by coinduct auto}
  thus ?thesis using assms by auto
qed

lemma all_imp_alw:
  assumes " $\bigwedge$  xs.  $\varphi$  xs" shows "alw  $\varphi$  xs"
proof-
  {assume " $\forall$  xs.  $\varphi$  xs"
    hence ?thesis by coinduct auto}
  thus ?thesis using assms by auto
qed

lemma alw_impl_ev_alw:
  assumes "alw ( $\varphi$  impl ev  $\psi$ ) xs"
  shows "alw (ev  $\varphi$  impl ev  $\psi$ ) xs"
  using assms by coinduct (auto dest: ev_alw_impl)

lemma ev_holds_sset:
  "ev (holds P) xs  $\longleftrightarrow$  ( $\exists$  x  $\in$  sset xs. P x)" (is "?L  $\longleftrightarrow$  ?R")
proof safe
  assume ?L thus ?R by induct (metis holds.simps stream.set_sel(1), metis stl_sset)
next
  fix x assume "x  $\in$  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 " $\varphi$  xs" and "alw ( $\varphi$  impl nxt  $\varphi$ ) xs"
  shows "alw  $\varphi$  xs"
proof-
  {assume " $\varphi$  xs  $\wedge$  alw ( $\varphi$  impl nxt  $\varphi$ ) xs" hence ?thesis
    by coinduct auto}
  thus ?thesis using assms by auto
qed

lemma variance:
  assumes 1: " $\varphi$  xs" and 2: "alw ( $\varphi$  impl ( $\psi$  or nxt  $\varphi$ )) xs"
  shows "(alw  $\varphi$  or ev  $\psi$ ) xs"
proof-
  {assume " $\neg$  ev  $\psi$  xs" hence "alw (not  $\psi$ ) xs" unfolding not_ev[symmetric] .
    moreover have "alw (not  $\psi$  impl ( $\varphi$  impl nxt  $\varphi$ )) xs"
    using 2 by coinduct auto
    ultimately have "alw ( $\varphi$  impl nxt  $\varphi$ ) xs" by(auto dest: alw_mp)
    with 1 have "alw  $\varphi$  xs" by(rule alw_invar)}
  thus ?thesis by blast
qed

lemma ev_alw_imp_nxt:
  assumes e: "ev  $\varphi$  xs" and a: "alw ( $\varphi$  impl (nxt  $\varphi$ )) xs"

```

```

shows "ev (alw  $\varphi$ ) xs"
proof-
  obtain xl xs1 where xs: "xs = xl @- xs1" and  $\varphi$ : " $\varphi$  xs1"
  using e by (metis ev_imp_shift)
  have " $\varphi$  xs1  $\wedge$  alw ( $\varphi$  impl (nxt  $\varphi$ )) xs1" using a  $\varphi$  unfolding xs by (metis alw_shift)
  hence "alw  $\varphi$  xs1" by (coinduct xs1 rule: alw.coinduct) auto
  thus ?thesis unfolding xs by (auto intro: alw_ev_shift)
qed

inductive ev_at :: "('a stream  $\Rightarrow$  bool)  $\Rightarrow$  nat  $\Rightarrow$  'a stream  $\Rightarrow$  bool" for P :: "'a stream  $\Rightarrow$  bool" where
  base: "P  $\omega \Rightarrow$  ev_at P 0  $\omega$ "
| step: " $\neg$  P  $\omega \Rightarrow$  ev_at P n (stl  $\omega$ )  $\Rightarrow$  ev_at P (Suc n)  $\omega$ "

inductive_simps ev_at_0[simp]: "ev_at P 0  $\omega$ "
inductive_simps ev_at_Suc[simp]: "ev_at P (Suc n)  $\omega$ "

lemma ev_at_imp_snth: "ev_at P n  $\omega \Rightarrow$  P (sdrop n  $\omega$ )"
  by (induction n arbitrary:  $\omega$ ) auto

lemma ev_at_HLD_imp_snth: "ev_at (HLD X) n  $\omega \Rightarrow$   $\omega$  !! n  $\in$  X"
  by (auto dest!: ev_at_imp_snth simp: HLD_iff)

lemma ev_at_HLD_single_imp_snth: "ev_at (HLD {x}) n  $\omega \Rightarrow$   $\omega$  !! n = x"
  by (drule ev_at_HLD_imp_snth) simp

lemma ev_at_unique: "ev_at P n  $\omega \Rightarrow$  ev_at P m  $\omega \Rightarrow$  n = m"
proof (induction arbitrary: m rule: ev_at.induct)
  case (base  $\omega$ ) then show ?case
    by (simp add: ev_at.simps[of _ _  $\omega$ ])
next
  case (step  $\omega$  n) from step.prem1 step.hyps step.IH[of "m - 1"] show ?case
    by (auto simp add: ev_at.simps[of _ _  $\omega$ ])
qed

lemma ev_iff_ev_at: "ev P  $\omega \longleftrightarrow (\exists n. \text{ev\_at } P \ n \ \omega)"$ "
proof
  assume "ev P  $\omega$ " then show " $\exists n. \text{ev\_at } P \ n \ \omega$ "
    by (induction rule: ev_induct_strong) (auto intro: ev_at.intros)
next
  assume " $\exists n. \text{ev\_at } P \ n \ \omega$ "
  then obtain n where "ev_at P n  $\omega$ "
    by auto
  then show "ev P  $\omega$ "
    by induction auto
qed

lemma ev_at_shift: "ev_at (HLD X) i (stake (Suc i)  $\omega$  @-  $\omega'$  :: 's stream)  $\longleftrightarrow$  ev_at (HLD X) i  $\omega$ "
  by (induction i arbitrary:  $\omega$ ) (auto simp: HLD_iff)

lemma ev_iff_ev_at_unique: "ev P  $\omega \longleftrightarrow (\exists !n. \text{ev\_at } P \ n \ \omega)"$ "
  by (auto intro: ev_at_unique simp: ev_iff_ev_at)

lemma alw_HLD_iff_streams: "alw (HLD X)  $\omega \longleftrightarrow \omega \in \text{streams } X$ "
proof
  assume "alw (HLD X)  $\omega$ " then show " $\omega \in \text{streams } X$ "
  proof (coinduction arbitrary:  $\omega$ )
    case (streams  $\omega$ ) then show ?case by (cases  $\omega$ ) auto
  qed
next
  assume " $\omega \in \text{streams } X$ " then show "alw (HLD X)  $\omega$ "

```

```

proof (coinduction arbitrary:  $\omega$ )
  case (alw  $\omega$ ) then show ?case by (cases  $\omega$ ) auto
qed
qed

lemma not_HLD: "not (HLD X) = HLD ( $\neg$  X)"
  by (auto simp: HLD_iff)

lemma not_alw_iff: " $\neg$  (alw P  $\omega$ )  $\longleftrightarrow$  ev (not P)  $\omega$ "
  using not_alw[of P] by (simp add: fun_eq_iff)

lemma not_ev_iff: " $\neg$  (ev P  $\omega$ )  $\longleftrightarrow$  alw (not P)  $\omega$ "
  using not_alw_iff[of "not P"  $\omega$ , symmetric] by simp

lemma ev_Stream: "ev P (x ## s)  $\longleftrightarrow$  P (x ## s)  $\vee$  ev P s"
  by (auto elim: ev.cases)

lemma alw_ev_imp_ev_alw:
  assumes "alw (ev P)  $\omega$ " shows "ev (P aand alw (ev P))  $\omega$ "
proof -
  have "ev P  $\omega$ " using assms by auto
  from this assms show ?thesis
    by induct auto
qed

lemma ev_False: "ev ( $\lambda x$ . False)  $\omega$   $\longleftrightarrow$  False"
proof
  assume "ev ( $\lambda x$ . False)  $\omega$ " then show False
    by induct auto
qed auto

lemma alw_False: "alw ( $\lambda x$ . False)  $\omega$   $\longleftrightarrow$  False"
  by auto

lemma ev_iff_sdrop: "ev P  $\omega$   $\longleftrightarrow$  ( $\exists m$ . P (sdrop m  $\omega$ ))"
proof safe
  assume "ev P  $\omega$ " then show " $\exists m$ . P (sdrop m  $\omega$ )"
    by (induct rule: ev_induct_strong) (auto intro: exI[of _ 0] exI[of _ "Suc n" for n])
next
  fix m assume "P (sdrop m  $\omega$ )" then show "ev P  $\omega$ "
    by (induct m arbitrary:  $\omega$ ) auto
qed

lemma alw_iff_sdrop: "alw P  $\omega$   $\longleftrightarrow$  ( $\forall m$ . P (sdrop m  $\omega$ ))"
proof safe
  fix m assume "alw P  $\omega$ " then show "P (sdrop m  $\omega$ )"
    by (induct m arbitrary:  $\omega$ ) auto
next
  assume " $\forall m$ . P (sdrop m  $\omega$ )" then show "alw P  $\omega$ "
    by (coinduction arbitrary:  $\omega$ ) (auto elim: allE[of _ 0] allE[of _ "Suc n" for n])
qed

lemma infinite_iff_alw_ev: "infinite {m. P (sdrop m  $\omega$ )}  $\longleftrightarrow$  alw (ev P)  $\omega$ "
  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: " $\bigwedge s$ . f (stl s) = stl (f s)"
  shows "alw P (f s)  $\longleftrightarrow$  alw ( $\lambda x$ . P (f x)) s"
proof
  assume "alw P (f s)" then show "alw ( $\lambda x$ . P (f x)) s"

```

```

      by (coinduction arbitrary: s rule: alw_coinduct)
        (auto simp: stl)
next
  assume "alw ( $\lambda x. P (f x)$ ) s" then show "alw P (f s)"
    by (coinduction arbitrary: s rule: alw_coinduct) (auto simp flip: stl)
qed

lemma ev_inv:
  assumes stl: " $\bigwedge s. f (stl\ s) = stl\ (f\ s)$ "
  shows "ev P (f s)  $\longleftrightarrow$  ev ( $\lambda x. P (f x)$ ) s"
proof
  assume "ev P (f s)" then show "ev ( $\lambda x. P (f x)$ ) s"
    by (induction "f s" arbitrary: s) (auto simp: stl)
next
  assume "ev ( $\lambda x. P (f x)$ ) s" then show "ev P (f s)"
    by induction (auto simp flip: stl)
qed

lemma alw_smap: "alw P (smap f s)  $\longleftrightarrow$  alw ( $\lambda x. P (smap\ f\ x)$ ) s"
  by (rule alw_inv) simp

lemma ev_smap: "ev P (smap f s)  $\longleftrightarrow$  ev ( $\lambda x. P (smap\ f\ x)$ ) s"
  by (rule ev_inv) simp

lemma alw_cong:
  assumes P: "alw P  $\omega$ " and eq: " $\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$ "
  shows "alw Q1  $\omega \longleftrightarrow$  alw Q2  $\omega$ "
proof -
  from eq have "(alw P aand Q1) = (alw P aand Q2)" by auto
  then have "alw (alw P aand Q1)  $\omega$  = alw (alw P aand Q2)  $\omega$ " by auto
  with P show "alw Q1  $\omega \longleftrightarrow$  alw Q2  $\omega$ "
    by (simp add: alw_aand)
qed

lemma ev_cong:
  assumes P: "alw P  $\omega$ " and eq: " $\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega$ "
  shows "ev Q1  $\omega \longleftrightarrow$  ev Q2  $\omega$ "
proof -
  from P have "alw ( $\lambda xs. Q1\ xs \longrightarrow Q2\ xs$ )  $\omega$ " by (rule alw_mono) (simp add: eq)
  moreover from P have "alw ( $\lambda xs. Q2\ xs \longrightarrow Q1\ xs$ )  $\omega$ " by (rule alw_mono) (simp add: eq)
  moreover note ev_alw_impl[of Q1  $\omega$  Q2] ev_alw_impl[of Q2  $\omega$  Q1]
  ultimately show "ev Q1  $\omega \longleftrightarrow$  ev Q2  $\omega$ "
    by auto
qed

lemma alwD: "alw P x  $\implies$  P x"
  by auto

lemma alw_alwD: "alw P  $\omega \implies$  alw (alw P)  $\omega$ "
  by simp

lemma alw_ev_stl: "alw (ev P) (stl  $\omega$ )  $\longleftrightarrow$  alw (ev P)  $\omega$ "
  by (auto intro: alw.intros)

lemma holds_Stream: "holds P (x ## s)  $\longleftrightarrow$  P x"
  by simp

lemma holds_eq1[simp]: "holds ((=) x) = HLD {x}"
  by rule (auto simp: HLD_iff)

lemma holds_eq2[simp]: "holds ( $\lambda y. y = x$ ) = 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  $\varphi$   $\psi$  where
  base: " $\psi \ \omega \implies (\varphi \text{ suntil } \psi) \ \omega$ "
| step: " $\varphi \ \omega \implies (\varphi \text{ suntil } \psi) \ (\text{stl } \omega) \implies (\varphi \text{ suntil } \psi) \ \omega$ "

inductive_simps suntil_Stream: " $(\varphi \text{ suntil } \psi) \ (x \ \#\ s)$ "

end

lemma suntil_induct_strong[consumes 1, case_names base step]:
  " $(\varphi \text{ suntil } \psi) \ x \implies$ 
    ( $\bigwedge \omega. \ \psi \ \omega \implies P \ \omega$ )  $\implies$ 
    ( $\bigwedge \omega. \ \varphi \ \omega \implies \neg \psi \ \omega \implies (\varphi \text{ suntil } \psi) \ (\text{stl } \omega) \implies P \ (\text{stl } \omega) \implies P \ \omega \implies P \ x$ )"
  using suntil.induct[of  $\varphi \ \psi \ x \ P$ ] by blast

lemma ev_suntil: " $(\varphi \text{ suntil } \psi) \ \omega \implies \text{ev } \psi \ \omega$ "
  by (induct rule: suntil.induct) auto

lemma suntil_inv:
  assumes stl: " $\bigwedge s. \ f \ (\text{stl } s) = \text{stl } (f \ s)$ "
  shows " $(P \text{ suntil } Q) \ (f \ s) \longleftrightarrow ((\lambda x. \ P \ (f \ x)) \text{ suntil } (\lambda x. \ Q \ (f \ x))) \ s$ "
proof
  assume "( $P \text{ suntil } Q$ )  $(f \ s)$ " then show " $((\lambda x. \ P \ (f \ x)) \text{ suntil } (\lambda x. \ Q \ (f \ x))) \ s$ "
    by (induction "f s" arbitrary: s) (auto simp: stl intro: suntil.intros)
next
  assume " $((\lambda x. \ P \ (f \ x)) \text{ suntil } (\lambda x. \ Q \ (f \ x))) \ s$ " then show " $(P \text{ suntil } Q) \ (f \ s)$ "
    by induction (auto simp flip: stl intro: suntil.intros)
qed

lemma suntil_smap: " $(P \text{ suntil } Q) \ (\text{smap } f \ s) \longleftrightarrow ((\lambda x. \ P \ (\text{smap } f \ x)) \text{ suntil } (\lambda x. \ Q \ (\text{smap } f \ x))) \ s$ "
  by (rule suntil_inv) simp

lemma hld_smap: " $\text{HLD } x \ (\text{smap } f \ s) = \text{holds } (\lambda y. \ f \ y \in x) \ s$ "
  by (simp add: HLD_def)

lemma suntil_mono:
  assumes eq: " $\bigwedge \omega. \ P \ \omega \implies Q1 \ \omega \implies Q2 \ \omega$ " " $\bigwedge \omega. \ P \ \omega \implies R1 \ \omega \implies R2 \ \omega$ "
  assumes *: " $(Q1 \text{ suntil } R1) \ \omega$ " " $\text{alw } P \ \omega$ " shows " $(Q2 \text{ suntil } R2) \ \omega$ "
  using * by induct (auto intro: eq suntil.intros)

lemma suntil_cong:
  " $\text{alw } P \ \omega \implies (\bigwedge \omega. \ P \ \omega \implies Q1 \ \omega \longleftrightarrow Q2 \ \omega) \implies (\bigwedge \omega. \ P \ \omega \implies R1 \ \omega \longleftrightarrow R2 \ \omega) \implies$ 
     $(Q1 \text{ suntil } R1) \ \omega \longleftrightarrow (Q2 \text{ suntil } R2) \ \omega$ "
  using suntil_mono[of P Q1 Q2 R1 R2  $\omega$ ] suntil_mono[of P Q2 Q1 R2 R1  $\omega$ ] by auto

lemma ev_suntil_iff: " $\text{ev } (P \text{ suntil } Q) \ \omega \longleftrightarrow \text{ev } Q \ \omega$ "
proof
  assume " $\text{ev } (P \text{ suntil } Q) \ \omega$ " then show " $\text{ev } Q \ \omega$ "
    by induct (auto dest: ev_suntil)
next
  assume " $\text{ev } Q \ \omega$ " then show " $\text{ev } (P \text{ suntil } Q) \ \omega$ "
    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
qed
```

```
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 ( $\lambda x. Q$  (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)  $\longleftrightarrow$  (not (holds Q) until (holds (Q and P))) s"
proof
  assume "holds P (sfilter Q s)" with assms show "(not (holds Q) until (holds (Q and P))) s"
  by (induct rule: ev_induct_strong) (auto intro: until.intros)
next
  assume "(not (holds Q) until (holds (Q and P))) s" then show "holds P (sfilter Q s)"
  by induct auto
qed

lemma until_and_nxt:
  " $(\varphi$  until ( $\varphi$  and nxt  $\psi$ ))  $\omega \longleftrightarrow (\varphi$  and nxt ( $\varphi$  until  $\psi$ ))  $\omega$ "
proof
  assume " $(\varphi$  until ( $\varphi$  and nxt  $\psi$ ))  $\omega$ " then show " $(\varphi$  and nxt ( $\varphi$  until  $\psi$ ))  $\omega$ "
  by induction (auto intro: until.intros)
next
  assume " $(\varphi$  and nxt ( $\varphi$  until  $\psi$ ))  $\omega$ "
  then have " $(\varphi$  until  $\psi$ ) (stl  $\omega$ )" "  $\varphi \omega$ "
  by auto
  then show " $(\varphi$  until ( $\varphi$  and nxt  $\psi$ ))  $\omega$ "
  by (induction "stl  $\omega$ " arbitrary:  $\omega$ )
    (auto elim: until.cases intro: until.intros)
qed

lemma alw_sconst: "alw P (sconst x)  $\longleftrightarrow$  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)  $\longleftrightarrow$  P (sconst x)"
proof
  assume "ev P (sconst x)" then show "P (sconst x)"
  by (induction "sconst x") auto
qed auto

lemma until_sconst: " $(\varphi$  until  $\psi$ ) (sconst x)  $\longleftrightarrow \psi$  (sconst x)"
proof
  assume " $(\varphi$  until  $\psi$ ) (sconst x)" then show " $\psi$  (sconst x)"
  by (induction "sconst x") auto
qed (auto intro: until.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)  $\omega$ "

```

```

    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 until P) ω"
proof
  assume "ev P ω" then show "((λxs. ¬ P xs) until P) ω"
    by (induction rule: ev_induct_strong) (auto intro: until.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 :: registers
  event :: event
  "output" :: outputs

type_synonym property = "state stream ⇒ 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 ⇒ registers ⇒ event ⇒ (nat option × outputs × registers)" 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)))
  )"

lemma ltl_step_alt: "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)

primcorec make_full_observation :: "transition_matrix ⇒ nat option ⇒ registers ⇒ event stream ⇒
state stream" 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  $\Rightarrow$  event stream  $\Rightarrow$  state stream" where
"watch e i  $\equiv$  (make_full_observation e (Some 0) <> i)"

definition Outputs :: "nat  $\Rightarrow$  state stream  $\Rightarrow$  value option" where
"Outputs n s  $\equiv$  nth (output (shd s)) n"

definition Inputs :: "nat  $\Rightarrow$  state stream  $\Rightarrow$  value" where
"Inputs n s  $\equiv$  nth (inputs (shd s)) (n-1)"

definition Registers :: "nat  $\Rightarrow$  state stream  $\Rightarrow$  value option" where
"Registers n s  $\equiv$  datastate (shd s) $ n"

definition StateEq :: "nat option  $\Rightarrow$  state stream  $\Rightarrow$  bool" where
"StateEq v s  $\equiv$  statename (shd s) = v"

lemma StateEq_None_not_Some: "StateEq None s  $\implies \neg$  StateEq (Some n) s"
by (simp add: StateEq_def)

definition LabelEq :: "string  $\Rightarrow$  state stream  $\Rightarrow$  bool" where
"LabelEq v s  $\equiv$  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)

definition InputEq :: "value list  $\Rightarrow$  state stream  $\Rightarrow$  bool" where
"InputEq v s  $\equiv$  inputs (shd s) = v"

definition EventEq :: "(string  $\times$  inputs)  $\Rightarrow$  state stream  $\Rightarrow$  bool" where
"EventEq e = LabelEq (fst e) aand InputEq (snd e)"

definition OutputEq :: "value option list  $\Rightarrow$  state stream  $\Rightarrow$  bool" where
"OutputEq v s  $\equiv$  output (shd s) = v"

definition InputLength :: "nat  $\Rightarrow$  state stream  $\Rightarrow$  bool" where
"InputLength v s  $\equiv$  length (inputs (shd s)) = v"

definition OutputLength :: "nat  $\Rightarrow$  state stream  $\Rightarrow$  bool" where
"OutputLength v s  $\equiv$  length (output (shd s)) = v"

fun "checkInx" :: "ior  $\Rightarrow$  nat  $\Rightarrow$  (value option  $\Rightarrow$  value option  $\Rightarrow$  trilean)  $\Rightarrow$  value option  $\Rightarrow$  state
stream  $\Rightarrow$  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) $ n) v = trilean.true)"

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 -
  obtain ss :: "((String.literal × value list) stream ⇒ state stream) ⇒ (String.literal × value list)
  stream" where
    "∀ f p s. f (stl (ss f)) ≠ stl (f (ss f)) ∨ alw p (f s) = alw (λs. p (f s)) s"
  by (metis (no_types) alw_inv)
  then show ?thesis
  by (simp add: StateEq_def all_imp_alw)
qed

lemma no_output_none: "alw (OutputEq []) (make_full_observation e None r t)"
proof -
  obtain ss :: "((String.literal × value list) stream ⇒ state stream) ⇒ (String.literal × value list)
  stream" where
    "∀ f p s. f (stl (ss f)) ≠ stl (f (ss f)) ∨ alw p (f s) = alw (λs. p (f s)) s"
  by (metis (no_types) alw_inv)
  then show ?thesis
  by (simp add: OutputEq_def all_imp_alw)
qed

lemma no_updates_none: "alw (λx. datastate (shd x) = r) (make_full_observation e None r t)"
proof -
  obtain ss :: "((String.literal × value list) stream ⇒ state stream) ⇒ (String.literal × value list)
  stream" where
    "∀ f p s. f (stl (ss f)) ≠ stl (f (ss f)) ∨ alw p (f s) = alw (λs. p (f s)) s"
  by (metis (no_types) alw_inv)
  then show ?thesis
  by (simp add: all_imp_alw)
qed

lemma no_updates_none_individual: "alw (checkInx rg n ValueEq (r $ n)) (make_full_observation e None
r t)"
proof -
  obtain ss :: "((String.literal × value list) stream ⇒ state stream) ⇒ (String.literal × value list)
  stream" where
    "∀ f p s. f (stl (ss f)) ≠ stl (f (ss f)) ∨ alw p (f s) = alw (λs. p (f s)) s"
  by (metis (no_types) alw_inv)
  then show ?thesis
  by (simp add: ValueEq_def all_imp_alw)
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)

end
theory Coin_Tea
  imports "../EFSM_LTL"
begin

declare One_nat_def [simp del]
declare ValueLt_def [simp]
declare ltl_step_alt [simp]
definition init :: transition where

```

```
"init ≡ (|
  Label = (STR ''init''),
  Arity = 0,
  Guard = [],
  Outputs = [],
  Updates = [(1, (L (Num 0)))]
|)"
```

definition coin :: transition where

```
"coin ≡ (|
  Label = (STR ''coin''),
  Arity = 0,
  Guard = [],
  Outputs = [],
  Updates = [(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 <> 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: " \neg (a = STR ''init'' \wedge b = []) \implies possible_steps drinks 0 <> a b = {}"
 apply (simp add: possible_steps_def Abs_ffilter Set.filter_def drinks_def)
 apply clarify
 by (simp add: init_def)

lemma aux1: " \neg StateEq (Some 2)
 (make_full_observation drinks (fst (ltl_step drinks (Some 0) <> (shd t)))
 (snd (snd (ltl_step drinks (Some 0) <> (shd t)))) (stl t))"

proof-

```
  show ?thesis
  apply (case_tac "shd t")
  apply simp
  apply (case_tac "a = STR ''init''  $\wedge$  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) <> (shd t)))
 (snd (snd (ltl_step drinks (Some 0) <> (shd t))))"

```

    (stl t) ≠
    make_full_observation drinks (Some 0) <> t"
  apply (case_tac "ltl_step drinks (Some 0) <> (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 <> 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 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 (<>(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 apply_guards)

lemma possible_steps_vend_sufficient: "n > 0 ⇒ possible_steps drinks 1 (<>(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 apply_guards)

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_vend: "apply_updates (Updates vend) i r = r"
  by (simp add: vend_def)

lemma less_than_zero_not_nxt_2:
  "n ≤ 0 ⇒
  statename (shd (stl (make_full_observation drinks (Some 1) (<>(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 shd_not_lt_zero: "0 ≤ n ⇒ (λxs. MaybeBoolInt (<) (datastate (shd xs) $ 1) (Some (Num 0))) ≠
trilean.true) (make_full_observation drinks None (<>(1 := Num n)) t)"
  by simp

lemma nxt_not_lt_zero: "0 ≤ n ⇒ nxt (λxs. MaybeBoolInt (<) (datastate (shd xs) $ 1) (Some (Num 0)))
≠ trilean.true) (make_full_observation drinks None (<>(1 := Num n)) t)"
  by simp

lemma once_none_remains_not_lt_zero: "0 ≤ n ⇒ alw (λxs. MaybeBoolInt (<) (datastate (shd xs) $ 1)
(Some (Num 0))) ≠ trilean.true) (make_full_observation drinks None (<>(1 := Num n)) t)"
  using no_updates_none
  by (simp add: alw_iff_sdrop)

lemma once_none_null_remains_not_lt_zero: "alw (λxs. MaybeBoolInt (<) (datastate (shd xs) $ 1) (Some
(Num 0))) ≠ trilean.true) (make_full_observation drinks None <> t)"
  using no_updates_none
  by (simp add: alw_iff_sdrop)

```



```

lemma stop_at_2: "0 ≤ n ⇒
  alw (λxs. MaybeBoolInt (<) (datastate (shd xs) $ 1) (Some (Num 0)) ≠ trilean.true) (make_full_observation
drinks (Some 2) (<>(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 (not (checkInx rg 1 ValueLt (Some (Num 0)))) (make_full_observation drinks (Some 1) (<>(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'', [])")
  apply (simp add: possible_steps_coin datastate coin_def value_plus_def)
  by (simp add: invalid_possible_steps_1)

lemma not_initialised: "alw (λxs. StateEq (Some 1) xs ∧
  MaybeBoolInt (<) (datastate (shd xs) $ (1)) (Some (Num 0)) = trilean.true ∧ LabelEq ''vend''
xs ∧ InputEq [] xs →
  StateEq (Some 2) (stl xs))
  (make_full_observation drinks None <> 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 <> 1 i = finset x S' ⇒ finset x S' = {/(1, init)/}"
  apply (case_tac "1 = 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 (<>(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 apply_guards)

lemma updates_init: "apply_updates (Updates init) (λx. None) <> = (<>(1 := Num 0))"
  by (simp add: init_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'' until LabelEq ''coin'')) until 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) <> 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  $\implies$   $\neg$ ev ( $\lambda$ s. statename (shd s) = Some 2)
(make_full_observation drinks (Some 1) ( $\langle \rangle$ (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  $\neq$  (STR ''coin'', [])  $\implies$ 
x1  $\neq$  (STR ''vend'', [])  $\implies$ 
possible_steps drinks 1 ( $\langle \rangle$ (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 apply_guards)

lemma invalid_gets_stuck: "x1  $\neq$  (STR ''coin'', [])  $\implies$ 
x1  $\neq$  (STR ''vend'', [])  $\implies$ 
 $\neg$ ev ( $\lambda$ s. statename (shd s) = Some 2) (make_full_observation drinks (Some
1) ( $\langle \rangle$ (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'')) until LabelEq ''coin''))
    (watch drinks t)"
oops
lemma LTL_must_pay_correct:
  "((ev (StateEq (Some 2))) impl
    (not (LabelEq ''vend'')) until 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 until.step)
  apply simp
  apply (simp add: possible_steps_init updates_init)
  apply (case_tac "shd (stl t) = (STR ''coin'', [])")
  apply (simp add: until.base)
  apply (case_tac "shd (stl t) = (STR ''vend'', [])")
  apply (rule until.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
imports "../.. /FSM"
begin
definition coin :: transition where
  "coin ≡ (|
    Label = (STR ''coin''),
    Arity = 0,
    Guard = [],
    Outputs = [],
    Updates = [(1, (Plus (V (R 1)) (L (Num 1))))])"

end
theory XXXlinkedin_ext
imports "../.. /FSM_LTL"
begin

```

```

definition I :: "nat  $\Rightarrow$  vname" where
  "I n = vname.I (n-1)"
declare I_def [simp]

declare One_nat_def [simp del]

definition "login" :: "transition" where
"login  $\equiv$  ( $\{$ 
  Label = STR ''login'',
  Arity = 1,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''free''))
  ],
  Outputs = [],
  Updates = []
 $\}$ )"

definition "login1" :: "transition" where
"login1  $\equiv$  ( $\{$ 
  Label = STR ''login'',
  Arity = 1,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''paid''))
  ],
  Outputs = [],
  Updates = []
 $\}$ )"

definition "view" :: "transition" where
"view  $\equiv$  ( $\{$ 
  Label = STR ''view'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''friendID'')),
    GExp.Eq (V (I 2)) (L (Str ''name'')),
    GExp.Eq (V (I 3)) (L (Str ''HM8p''))
  ],
  Outputs = [],
  Updates = []
 $\}$ )"

definition "view1" :: "transition" where
"view1  $\equiv$  ( $\{$ 
  Label = STR ''view'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''otherID'')),
    GExp.Eq (V (I 2)) (L (Str ''OUT_OF_NETWORK'')),
    GExp.Eq (V (I 3)) (L (Str ''Mn5''))
  ],
  Outputs = [],
  Updates = []
 $\}$ )"

definition "view2" :: "transition" where
"view2  $\equiv$  ( $\{$ 
  Label = STR ''view'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''otherID'')),

```

```

        GExp.Eq (V (I 2)) (L (Str ''name'')),
        GExp.Eq (V (I 3)) (L (Str ''4zoF''))
    ],
    Outputs = [],
    Updates = []
]
"

definition "view3" :: "transition" where
"view3 ≡ [
    Label = STR ''view'',
    Arity = 3,
    Guard = [
        GExp.Eq (V (I 1)) (L (Str ''otherID'')),
        GExp.Eq (V (I 2)) (L (Str ''name'')),
        GExp.Eq (V (I 3)) (L (Str ''Mn5''))
    ],
    Outputs = [],
    Updates = []
]
"

definition "pdf" :: "transition" where
"pdf ≡ [
    Label = STR ''pdf'',
    Arity = 3,
    Guard = [
        GExp.Eq (V (I 1)) (L (Str ''friendID'')),
        GExp.Eq (V (I 2)) (L (Str ''name'')),
        GExp.Eq (V (I 3)) (L (Str ''HM8p''))
    ],
    Outputs = [
        (L (Str ''detailed_pdf_of_friendID''))
    ],
    Updates = []
]
"

definition "pdf1" :: "transition" where
"pdf1 ≡ [
    Label = STR ''pdf'',
    Arity = 3,
    Guard = [
        GExp.Eq (V (I 1)) (L (Str ''otherID'')),
        GExp.Eq (V (I 2)) (L (Str ''OUT_OF_NETWORK'')),
        GExp.Eq (V (I 3)) (L (Str ''Mn5''))
    ],
    Outputs = [
        (L (Str ''summary_pdf_of_otherID''))
    ],
    Updates = []
]
"

definition "pdf2" :: "transition" where
"pdf2 ≡ [
    Label = STR ''pdf'',
    Arity = 3,
    Guard = [
        GExp.Eq (V (I 1)) (L (Str ''otherID'')),
        GExp.Eq (V (I 2)) (L (Str ''name'')),
        GExp.Eq (V (I 3)) (L (Str ''4zoF''))
    ],
    Outputs = [
        (L (Str ''detailed_pdf_of_otherID''))
    ],
    Updates = []
]
"

```

```

],
Updates = []
)"

definition "linkedIn" :: "transition_matrix" where
"linkedIn ≡ {|
  ((0, 1), login),
  ((0, 1), login1),
  ((1, 2), view),
  ((1, 4), view1),
  ((1, 6), view2),
  ((1, 6), view3),
  ((2, 3), pdf),
  ((4, 5), pdf1),
  ((6, 7), pdf2)
|}"

lemma implode_login: "String.implode ''login'' = STR ''login''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_pdf: "String.implode ''pdf'' = STR ''pdf''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_friendID: "String.implode ''friendID'' = STR ''friendID''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_otherID: "String.implode ''otherID'' = STR ''otherID''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_HM8p: "String.implode ''HM8p'' = STR ''HM8p''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_MNn5: "String.implode ''MNn5'' = STR ''MNn5''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_4zoF: "String.implode ''4zoF'' = STR ''4zoF''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_name: "String.implode ''name'' = STR ''name''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_OON: "String.implode ''OUT_OF_NETWORK'' = STR ''OUT_OF_NETWORK''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_detailedPDF: "String.implode ''detailed_pdf_of_otherID'' = STR ''detailed_pdf_of_otherID''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemmas implode = implode_detailedPDF implode_OON implode_name implode_4zoF implode_MNn5
  implode_HM8p implode_friendID implode_otherID implode_pdf implode_login

lemma login_free: "possible_steps linkedIn 0 <> STR ''login'' [EFSM.Str ''free''] = {|(1, login)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  apply (simp_all add: apply_guards login_def login1_def Str_def)
  using trilean.distinct(1) by presburger

lemma not_view: "1 ≠ STR ''view'' ⇒
  possible_steps linkedIn 1 r 1 i = {|}|"
  apply (simp add: possible_steps_empty linkedIn_def)
  apply safe
  by (simp_all add: view_def view1_def view2_def view3_def)

```

```

lemma view_fuzz: "possible_steps linkedIn 1 <> STR ''view'' [EFSM.Str ''otherID'', EFSM.Str ''name'',
EFSM.Str ''MNn5''] = {|(6, view3)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  by (simp_all add: view_def view1_def view2_def view3_def apply_guards Str_def implode_otherID implode_friendID
implode_name implode_00N implode_MNn5 implode_4zoF numeral_2_eq_2)

lemma not_pdf_6: "1 ≠ STR ''pdf'' ⇒ possible_steps linkedIn 6 <> 1 i = {|}|"
  apply (simp add: possible_steps_empty linkedIn_def)
  apply safe
  by (simp_all add: pdf2_def)

lemma pdf_6_invalid_inputs: "i ≠ [EFSM.Str ''otherID'', EFSM.Str ''name'', EFSM.Str ''4zoF''] ⇒
possible_steps linkedIn 6 <> STR ''pdf'' i = {|}|"
  apply (case_tac i)
  apply (simp add: possible_steps_empty pdf2_def linkedIn_def)
  apply (case_tac list)
  apply (simp add: possible_steps_empty pdf2_def linkedIn_def)
  apply (case_tac lista)
  apply (simp add: possible_steps_empty pdf2_def linkedIn_def)
  apply (case_tac listb)
  apply (simp add: possible_steps_empty pdf2_def linkedIn_def apply_guards numeral_2_eq_2)
  by (simp add: possible_steps_empty pdf2_def linkedIn_def)

lemma pdf_fuzz: "possible_steps linkedIn 6 <> STR ''pdf'' [EFSM.Str ''otherID'', EFSM.Str ''name'',
EFSM.Str ''4zoF''] = {|(7, pdf2)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  by (simp_all add: pdf2_def apply_guards numeral_2_eq_2)
lemma contradiction: "fst (shd (stl (stl i))) = STR ''pdf'' ⇒
  snd (shd (stl (stl i))) = [value.Str STR ''otherID'', value.Str STR ''name'', value.Str STR ''4zoF'']
⇒ False"
oops

lemma "alw (λxs. label (shd xs) = STR ''pdf'' ∧ ValueEq (Some (Inputs 0 xs)) (Some (value.Str STR
''otherID''))) = trilean.true →
  output (shd xs) ≠ [Some (value.Str STR ''detailed_pdf_of_otherID'')]"
  (make_full_observation linkedIn (Some 6) <> (stl (stl i)))"
proof(coinduction)
  case alw
  then show ?case
  apply (simp add: ltl_step_alt)
  apply (case_tac "(fst (shd (stl (stl i)))) = STR ''pdf''")
  defer
  apply (simp add: not_pdf_6)
  using no_output_none[of linkedIn "<>" "stl (stl (stl i))"]
  unfolding OutputEq_def
  apply (simp add: alw_mono)
  apply simp
  apply (case_tac "(snd (shd (stl (stl i)))) = [Str ''otherID'', Str ''name'', Str ''4zoF'']")
  defer
  apply (simp add: pdf_6_invalid_inputs)
  using no_output_none[of linkedIn "<>" "stl (stl (stl i))"]
  unfolding OutputEq_def
  apply (simp add: alw_mono)
  apply (simp add: pdf_fuzz pdf2_def)
  apply standard
  apply standard
  apply (simp add: apply_outputs Str_def implode)
  apply (simp add: ValueEq_def Inputs_def)

```



```

oops

lemma after_login: "alw (λxs. label (shd xs) = String.implode ''pdf'' ∧ ValueEq (Some (Inputs 0 xs))
(Some (EFSM.Str ''otherID'')) = trilean.true →
  ¬ OutputEq [Some (EFSM.Str ''detailed_pdf_of_otherID'')] xs)
  (make_full_observation linkedIn (Some 1) <> (stl i))"
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: ltl_step_alt Str_def implode)
    apply (case_tac "(fst (shd (stl i))) = STR ''view''")
    defer
    apply (simp add: not_view)
    apply standard
    apply (simp add: OutputEq_def ltl_step_alt not_view)
    apply (simp add: OutputEq_def ValueEq_def)
    using no_output_none[of linkedIn "<>" "stl (stl i)"]
    unfolding OutputEq_def
    apply (simp add: alw_mono)
    apply (case_tac "(snd (shd (stl i))) = [Str ''otherID'', Str ''name'', Str ''Mn5'']")
    apply (simp add: ltl_step_alt view_fuzz view3_def)
    apply (rule disjI2)
    oops
lemma LTL_neverDetailed:
  "(((LabelEq ''login'' aand InputEq [Str ''free'']) impl
  (nxt (alw ((LabelEq ''pdf'' aand
  checkInx ip 1 ValueEq (Some (Str ''otherID'')))) impl
  (not (OutputEq [Some (Str ''detailed_pdf_of_otherID'')])))))))))
  (watch linkedIn i)"

  apply (simp add: watch_def ltl_step_alt)
  apply (simp add: InputEq_def LabelEq_def implode_login)
  apply (simp add: login_free login_def)
  apply standard
  oops

end
theory XXXlinkedIn_ext_fixed
imports "../EFSM_LTL"
begin

definition I :: "nat ⇒ vname" where
  "I n = vname.I (n-1)"
declare I_def [simp]

declare One_nat_def [simp del]

definition "login" :: "transition" where
  "login ≡ [
    Label = STR ''login'',
    Arity = 1,
    Guard = [],
    Outputs = [],
    Updates = [
      (1, (V (I 1)))
    ]
  ]"

definition "view" :: "transition" where
  "view ≡ [
    Label = STR ''view'',

```

```

    Arity = 3,
    Guard = [
      GExp.Eq (V (I 1)) (L (Str ''friendID'')),
      GExp.Eq (V (I 2)) (L (Str ''name'')),
      GExp.Eq (V (I 3)) (L (Str ''HM8p''))
    ],
    Outputs = [],
    Updates = []
  ]"

```

```

definition "view1" :: "transition" where
"view1" ≡ ()
  Label = STR ''view'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (R 1)) (L (Str ''free'')),
    GExp.Eq (V (I 1)) (L (Str ''otherID'')),
    GExp.Eq (V (I 2)) (L (Str ''OUT_OF_NETWORK'')),
    GExp.Eq (V (I 3)) (L (Str ''MNn5''))
  ],
  Outputs = [],
  Updates = []
]"

```

```

definition "view2" :: "transition" where
"view2" ≡ ()
  Label = STR ''view'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (R 1)) (L (Str ''free'')),
    GExp.Eq (V (I 1)) (L (Str ''otherID'')),
    GExp.Eq (V (I 2)) (L (Str ''name'')),
    GExp.Eq (V (I 3)) (L (Str ''4zoF''))
  ],
  Outputs = [],
  Updates = []
]"

```

```

definition "view3" :: "transition" where
"view3" ≡ ()
  Label = STR ''view'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (R 1)) (L (Str ''paid'')),
    GExp.Eq (V (I 1)) (L (Str ''otherID'')),
    GExp.Eq (V (I 2)) (L (Str ''name'')),
    GExp.Eq (V (I 3)) (L (Str ''MNn5''))
  ],
  Outputs = [],
  Updates = []
]"

```

```

definition "pdf" :: "transition" where
"pdf" ≡ ()
  Label = STR ''pdf'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''friendID'')),
    GExp.Eq (V (I 2)) (L (Str ''name'')),
    GExp.Eq (V (I 3)) (L (Str ''HM8p''))
  ],

```

```

    Outputs = [
      (L (Str ''detailed_pdf_of_friendID''))
    ],
    Updates = []
  }"

```

```

definition "pdf1" :: "transition" where
"pdf1" ≡ {
  Label = STR ''pdf'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''otherID'')),
    GExp.Eq (V (I 2)) (L (Str ''OUT_OF_NETWORK'')),
    GExp.Eq (V (I 3)) (L (Str ''MNn5''))
  ],
  Outputs = [
    (L (Str ''summary_pdf_of_otherID''))
  ],
  Updates = []
}"

```

```

definition "pdf2" :: "transition" where
"pdf2" ≡ {
  Label = STR ''pdf'',
  Arity = 3,
  Guard = [
    GExp.Eq (V (I 1)) (L (Str ''otherID'')),
    GExp.Eq (V (I 2)) (L (Str ''name'')),
    GExp.Eq (V (I 3)) (L (Str ''4zoF''))
  ],
  Outputs = [
    (L (Str ''detailed_pdf_of_otherID''))
  ],
  Updates = []
}"

```

```

definition "linkedIn" :: "transition_matrix" where
"linkedIn" ≡ {
  ((0, 1), login),
  ((1, 2), view),
  ((1, 4), view1),
  ((1, 4), view2),
  ((1, 6), view3),
  ((2, 3), pdf),
  ((4, 5), pdf1),
  ((6, 7), pdf2)
}"

```

```

lemmas transitions = login_def view_def view1_def view2_def view3_def pdf_def pdf1_def pdf2_def

```

```

lemma implode_login: "String.implode ''login'' = STR ''login''"
by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

```

```

lemma implode_pdf: "String.implode ''pdf'' = STR ''pdf''"
by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

```

```

lemma implode_friendID: "String.implode ''friendID'' = STR ''friendID''"
by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

```

```

lemma implode_otherID: "String.implode ''otherID'' = STR ''otherID''"
by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

```

```

lemma implode_HM8p: "String.implode ''HM8p'' = STR ''HM8p'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_MNn5: "String.implode ''MNn5'' = STR ''MNn5'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_4zoF: "String.implode ''4zoF'' = STR ''4zoF'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_name: "String.implode ''name'' = STR ''name'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_OON: "String.implode ''OUT_OF_NETWORK'' = STR ''OUT_OF_NETWORK'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_detailedPDF: "String.implode ''detailed_pdf_of_otherID'' = STR ''detailed_pdf_of_otherID'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_summaryPDF: "String.implode ''summary_pdf_of_otherID'' = STR ''summary_pdf_of_otherID'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_detailedPDF_friend: "String.implode ''detailed_pdf_of_friendID'' = STR ''detailed_pdf_of_friendID'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_paid: "String.implode ''paid'' = STR ''paid'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_free: "String.implode ''free'' = STR ''free'"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemmas implode = implode_summaryPDF implode_detailedPDF_friend implode_detailedPDF
  implode_OON implode_name implode_4zoF implode_MNn5 implode_HM8p
  implode_friendID implode_otherID implode_pdf implode_login implode_paid implode_free

lemma login_user: "possible_steps linkedIn 0 <> STR ''login'' [u] = {|(1, login)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  by (simp_all add: login_def)

lemma apply_updates_login: "apply_updates (Updates XXXlinkedin_ext_fixed.login) (join_ir [EFSM.Str
''free''] <>) <> = (<>(1 := Str ''free''))"
  by (simp add: login_def datastate)

lemma not_view_1: "1 ≠ STR ''view'' ⇒ possible_steps linkedIn 1 r l i = {|}|"
  by (simp add: possible_steps_empty linkedIn_def transitions)

lemma view_friend: "possible_steps linkedIn 1 (<>(1 := EFSM.Str ''free'')) STR ''view''
[EFSM.Str ''friendID'', EFSM.Str ''name'', EFSM.Str ''HM8p''] = {|(2, view)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  by (simp_all add: transitions apply_guards implode Str_def numeral_2_eq_2)

lemma not_pdf_2: "1 ≠ STR ''pdf'' ⇒ possible_steps linkedIn 2 r l i = {|}|"
  by (simp add: possible_steps_empty linkedIn_def transitions)

lemma pdf_friend: "possible_steps linkedIn 2 (<>(1 := EFSM.Str ''free'')) STR ''pdf''
[EFSM.Str ''friendID'', EFSM.Str ''name'', EFSM.Str ''HM8p''] = {|(3, pdf)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  by (simp_all add: transitions apply_guards implode Str_def numeral_2_eq_2)

```

```

lemma pdf_2_invalid: "i ≠ [Str ''friendID'', Str ''name'', Str ''HM8p''] ⇒
possible_steps linkedIn 2 (<>(1 := EFSM.Str ''free'')) STR ''pdf'' i = {||}"
  apply (case_tac i)
  apply (simp add: possible_steps_empty linkedIn_def pdf_def)
  apply (case_tac list)
  apply (simp add: possible_steps_empty linkedIn_def pdf_def)
  apply (case_tac lista)
  apply (simp add: possible_steps_empty linkedIn_def pdf_def)
  apply (case_tac listb)
  apply (simp add: possible_steps_empty linkedIn_def pdf_def apply_guards numeral_2_eq_2)
  by (simp add: possible_steps_empty linkedIn_def pdf_def)

lemma stop_at_3: "possible_steps linkedIn 3 r l i = {||}"
  by (simp add: possible_steps_empty linkedIn_def)

lemma stop_at_5: "possible_steps linkedIn 5 r l i = {||}"
  by (simp add: possible_steps_empty linkedIn_def)

lemma stop_at_7: "possible_steps linkedIn 7 r l i = {||}"
  by (simp add: possible_steps_empty linkedIn_def)

lemma s2_ok: "alw (λxs. label (shd xs) = STR ''pdf'' ∧ ValueEq (Some (Inputs 0 xs)) (Some (EFSM.Str
''otherID''))) = trilean.true →
  output (shd xs) ≠ [Some (EFSM.Str ''detailed_pdf_of_otherID'')]"
  (make_full_observation linkedIn (Some 2) (<>(1 := EFSM.Str ''free'')) i)"
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: ltl_step_alt)
    apply (case_tac "(fst (shd i)) = STR ''pdf''")
    defer
    apply (simp add: not_pdf_2)
    using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "(stl i)"]
    unfolding OutputEq_def
    apply (simp add: alw_mono)
    apply (case_tac "(snd (shd i)) = [Str ''friendID'', Str ''name'', Str ''HM8p'']")
    defer
    apply (simp add: pdf_2_invalid)
    using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "(stl i)"]
    unfolding OutputEq_def
    apply (simp add: alw_mono)
    apply (simp add: pdf_friend)
    apply standard
    apply (simp add: pdf_def apply_outputs Str_def implode)
    apply (rule disjI2)
    apply (simp add: pdf_def)
  proof(coinduction)
    case alw
    then show ?case
      apply (simp add: ltl_step_alt stop_at_3)
      using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "stl (stl i)"]
      unfolding OutputEq_def
      by (simp add: alw_mono)
  qed
qed

lemma view_other: "possible_steps linkedIn 1 (<>(1 := EFSM.Str ''free'')) STR ''view''
  [EFSM.Str ''otherID'', EFSM.Str ''OUT_OF_NETWORK'', EFSM.Str ''MNn5''] = {(4, view1)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe

```

```

by (simp_all add: transitions apply_guards implode Str_def numeral_2_eq_2)

lemma view_other_fuzz_foiled: "possible_steps linkedIn 1 (<>(1 := EFSM.Str ''free'')) STR ''view''
  [EFSM.Str ''otherID'', EFSM.Str ''name'', EFSM.Str ''4zoF'' ] = {|(4, view2)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  by (simp_all add: transitions apply_guards implode Str_def numeral_2_eq_2)

lemma pdf_summary: "possible_steps linkedIn 4 (<>(1 := EFSM.Str ''free'')) STR ''pdf''
  [EFSM.Str ''otherID'', EFSM.Str ''OUT_OF_NETWORK'', EFSM.Str ''MNn5'' ] = {|(5,
pdf1)|}"
  apply (simp add: possible_steps_singleton linkedIn_def)
  apply safe
  by (simp_all add: transitions apply_guards implode Str_def numeral_2_eq_2)

lemma not_pdf_4: "l ≠ STR ''pdf'' ⇒ possible_steps linkedIn 4 r l i = {|}"
  by (simp add: possible_steps_empty linkedIn_def transitions numeral_2_eq_2)

lemma pdf_4_invalid_inputs: "i ≠ [EFSM.Str ''otherID'', EFSM.Str ''OUT_OF_NETWORK'', EFSM.Str ''MNn5'' ]
⇒
possible_steps linkedIn 4 r l i = {|}"
  apply (case_tac i)
  apply (simp add: possible_steps_empty linkedIn_def pdf1_def)
  apply (case_tac list)
  apply (simp add: possible_steps_empty linkedIn_def pdf1_def)
  apply (case_tac lista)
  apply (simp add: possible_steps_empty linkedIn_def pdf1_def)
  apply (case_tac listb)
  apply (simp add: possible_steps_empty linkedIn_def pdf1_def apply_guards numeral_2_eq_2)
  by (simp add: possible_steps_empty linkedIn_def pdf1_def)

lemma s4_ok: "alw (λxs. label (shd xs) = STR ''pdf'' ∧ ValueEq (Some (Inputs 0 xs)) (Some (EFSM.Str
''otherID''))) = trilean.true →
  output (shd xs) ≠ [Some (EFSM.Str ''detailed_pdf_of_otherID'')]"
  (make_full_observation linkedIn (Some 4) (<>(1 := EFSM.Str ''free'')) i)"
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: ltl_step_alt)
    apply (case_tac "fst (shd i) = STR ''pdf''")
    defer
    apply (simp add: not_pdf_4 )
    using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "(stl i)"]
    unfolding OutputEq_def
    apply (simp add: alw_mono)
    apply (case_tac "(snd (shd i)) = [EFSM.Str ''otherID'', EFSM.Str ''OUT_OF_NETWORK'', EFSM.Str ''MNn5'' ]")
    defer
    apply (simp add: pdf_4_invalid_inputs)
    using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "(stl i)"]
    unfolding OutputEq_def
    apply (simp add: alw_mono)
    apply (simp add: pdf_summary)
    apply standard
    apply (simp add: pdf1_def apply_outputs Str_def implode)
    apply (simp add: pdf1_def)
    apply (rule disjI2)
  proof(coinduction)
    case alw
    then show ?case
      apply (simp add: ltl_step_alt stop_at_5)
      using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "stl (stl i)"]

```

```

    unfolding OutputEq_def
  by (simp add: alw_mono)
qed
qed

lemma invalid_input_1:
  "i ≠ [EFSM.Str ''friendID'', EFSM.Str ''name'', EFSM.Str ''HM8p''] ⇒
   i ≠ [EFSM.Str ''otherID'', EFSM.Str ''OUT_OF_NETWORK'', EFSM.Str ''MNn5''] ⇒
   i ≠ [EFSM.Str ''otherID'', EFSM.Str ''name'', EFSM.Str ''4zoF''] ⇒
   possible_steps linkedIn 1 (<>(1 := EFSM.Str ''free'')) 1 i = {}"
  apply (case_tac i)
  apply (simp add: possible_steps_empty linkedIn_def)
  apply safe[1]
  apply (simp add: transitions)+
  apply (case_tac list)
  apply (simp add: possible_steps_empty linkedIn_def)
  apply safe[1]
  apply (simp add: transitions)+
  apply (case_tac lista)
  apply (simp add: possible_steps_empty linkedIn_def)
  apply safe[1]
  apply (simp add: transitions)+
  apply (case_tac listb)
  apply (simp add: possible_steps_empty linkedIn_def)
  apply safe[1]
  by (simp_all add: transitions apply_guards Str_def implode possible_steps_empty linkedIn_def numeral_2_eq_2)

lemma after_login: "alw (λxs. label (shd xs) = STR ''pdf'' ∧ ValueEq (Some (Inputs 0 xs)) (Some (EFSM.Str ''otherID''))) = trilean.true →
  ¬ OutputEq [Some (EFSM.Str ''detailed_pdf_of_otherID'')] xs)
  (make_full_observation XXXlinkedin_ext_fixed.linkedIn (Some 1) (<>(1 := EFSM.Str ''free'')) i)"
proof(coinduction)
  case alw
  then show ?case
  apply (simp add: ltl_step_alt)
  apply (case_tac "(fst (shd i)) = STR ''view''")
  defer
  apply (simp add: not_view_1)
  using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "(stl i)"]
  unfolding OutputEq_def
  apply (simp add: alw_mono ValueEq_def ltl_step_alt not_view_1)
  apply simp
  apply (case_tac "(snd (shd i)) = [Str ''friendID'', Str ''name'', Str ''HM8p'']")
  apply (simp add: view_friend view_def s2_ok)
  apply (case_tac "(snd (shd i)) = [Str ''otherID'', Str ''OUT_OF_NETWORK'', Str ''MNn5'']")
  apply (simp add: view_other view1_def s4_ok)
  apply (case_tac "(snd (shd i)) = [Str ''otherID'', Str ''name'', Str ''4zoF'']")
  apply (simp add: view_other_fuzz_foiled view2_def s4_ok)
  apply (simp add: invalid_input_1)
  using no_output_none[of linkedIn "(<>(1 := EFSM.Str ''free''))" "(stl i)"]
  unfolding OutputEq_def
  by (simp add: alw_mono)
qed

lemma LTL_neverDetailed:
  "(((LabelEq ''login'' aand InputEq [Str ''free'']) impl
   (nxt (alw ((LabelEq ''pdf'' aand
    checkInx ip 1 ValueEq (Some (Str ''otherID''))) impl
    (not (OutputEq [Some (Str ''detailed_pdf_of_otherID'')]))))))))
   (watch linkedIn i))"
  apply (simp add: watch_def ltl_step_alt)
  apply (simp add: InputEq_def LabelEq_def)

```

```
apply (simp add: implode login_user apply_updates_login)
by (simp add: after_login)
```

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>.