

Mathematical Toolkit for Isabelle/UTP

Simon Foster

Pedro Ribeiro

Frank Zeyda

June 24, 2019

Abstract

This document describes our mathematical toolkit for Isabelle/UTP, which provides a foundational collection of definition, theorems, and proof facilities. This includes extensions to existing HOL libraries, such as for list and partial functions, and also new type definitions, theorems, and Isabelle/HOL commands.

Contents

1	Introduction	3
2	Lists: extra functions and properties	3
2.1	Useful Abbreviations	4
2.2	Folds	4
2.3	List Lookup	4
2.4	Extra List Theorems	4
2.4.1	Map	4
2.4.2	Sorted Lists	5
2.4.3	List Update	7
2.4.4	Drop While and Take While	8
2.4.5	Last and But Last	8
2.4.6	Prefixes and Strict Prefixes	9
2.4.7	Lexicographic Order	12
2.5	Distributed Concatenation	14
2.6	List Domain and Range	14
2.7	Extracting List Elements	14
2.8	Filtering a list according to a set	16
2.9	Minus on lists	17
2.10	Laws on <i>take</i> , <i>drop</i> , and <i>nths</i>	19
2.11	List power	19
3	Infinite Sequences	20
4	Finite Sets: extra functions and properties	25
5	Countable Sets: Extra functions and properties	30
5.1	Extra syntax	30
5.2	Countable set functions	30

6	Extra Relational Definitions and Theorems	37
6.1	Relational Function Operations	37
6.2	Domain Restriction	37
6.3	Relational Override	37
6.4	Functional Relations	38
6.5	Left-Total Relations	39
6.6	Relation Sets	39
6.7	Closure Properties	39
7	Map Type: extra functions and properties	40
7.1	Graphing Maps	40
7.2	Map Application	42
7.3	Map Membership	42
7.4	Preimage	43
7.5	Minus operation for maps	43
7.6	Map Bind	44
7.7	Range Restriction	44
7.8	Map Inverse and Identity	44
7.9	Merging of compatible maps	52
7.10	Conversion between lists and maps	52
7.11	Map Comprehension	53
7.12	Sorted lists from maps	54
7.13	Extra map lemmas	54
8	Alternative List Lexicographic Order	55
9	Partial Functions	56
9.1	Partial function type and operations	56
9.2	Algebraic laws	58
9.3	Lambda abstraction	59
9.4	Membership, application, and update	59
9.5	Domain laws	61
9.6	Range laws	62
9.7	Domain restriction laws	62
9.8	Range restriction laws	63
9.9	Graph laws	63
9.10	Entries	64
9.11	Summation	64
9.12	Partial Function Lens	65
10	Finite Functions	65
10.1	Finite function type and operations	65
10.2	Algebraic laws	67
10.3	Membership, application, and update	68
10.4	Domain laws	69
10.5	Range laws	70
10.6	Domain restriction laws	70
10.7	Range restriction laws	71
10.8	Graph laws	71
10.9	Partial Function Lens	71

11 Infinity Supplement	72
11.1 Type class <i>infinite</i>	72
11.2 Infinity Theorems	72
11.3 Instantiations	74
12 Positive Subtypes	74
12.1 Type Definition	74
12.2 Operators	74
12.3 Instantiations	75
12.4 Theorems	76
12.5 Transfer to Reals	77
13 Recall Undeclarations	77
13.1 ML File Import	78
13.2 Outer Commands	78
14 Meta-theory for UTP Toolkit	78

1 Introduction

This document contains the description of our mathematical toolkit for Isabelle/UTP [2, 3, 4, 7], a mechanisation of Hoare and He’s *Unifying Theories of Programming* [5, 1]. The toolkit provides a foundational collection of additional HOL theorems, new abstract types, and proof facilities, upon which Isabelle/UTP depends. In brief, the toolkit contains the following principal items:

- additional laws and functions for the list, map (partial functions), countable set, and finite set types;
- type definitions for partial and finite functions, together with additional functions and laws derived from the Z mathematical toolkit [6];
- positive subtypes of existing types;
- infinite sequences;
- the “total recall” package, which allows us to precisely control overriding of existing syntax annotations.

A few other theories exist that add smaller utilities and additional laws.

2 Lists: extra functions and properties

```
theory List-Extra
  imports
    HOL-Library.Sublist
    HOL-Library.Monad-Syntax
    HOL-Library.Prefix-Order
    Optics.Lens-Instances
begin
```

2.1 Useful Abbreviations

abbreviation $list\text{-}sum\ xs \equiv foldr\ (+)\ xs\ 0$

2.2 Folds

context $abel\text{-}semigroup$

begin

lemma $foldr\text{-}snoc$: $foldr\ (\ast)\ (xs\ @\ [x])\ k = (foldr\ (\ast)\ xs\ k) \ast x$
by ($induct\ xs$, $simp\text{-}all\ add$: $commute\ left\text{-}commute$)

end

2.3 List Lookup

The following variant of the standard nth function returns \perp if the index is out of range.

primrec

$nth\text{-}el :: 'a\ list \Rightarrow nat \Rightarrow 'a\ option\ (-\langle-\rangle_l\ [90,\ 0]\ 91)$

where

$\langle-\rangle_l = None$

$| (x\ \# \ xs)\langle i \rangle_l = (case\ i\ of\ 0 \Rightarrow Some\ x\ |\ Suc\ j \Rightarrow xs\ \langle j \rangle_l)$

lemma $nth\text{-}el\text{-}appendl[simp]$: $i < length\ xs \Longrightarrow (xs\ @\ ys)\langle i \rangle_l = xs\langle i \rangle_l$

apply ($induct\ xs\ arbitrary$: i)

apply $simp$

apply ($case\text{-}tac\ i$)

apply $simp\text{-}all$

done

lemma $nth\text{-}el\text{-}appendr[simp]$: $length\ xs \leq i \Longrightarrow (xs\ @\ ys)\langle i \rangle_l = ys\langle i - length\ xs \rangle_l$

apply ($induct\ xs\ arbitrary$: i)

apply $simp$

apply ($case\text{-}tac\ i$)

apply $simp\text{-}all$

done

2.4 Extra List Theorems

2.4.1 Map

lemma $map\text{-}nth\text{-}Cons\text{-}atLeastLessThan$:

$map\ (nth\ (x\ \# \ xs))\ [Suc\ m..<n] = map\ (nth\ xs)\ [m..<n - 1]$

proof –

have $nth\ xs = nth\ (x\ \# \ xs) \circ Suc$

by $auto$

hence $map\ (nth\ xs)\ [m..<n - 1] = map\ (nth\ (x\ \# \ xs) \circ Suc)\ [m..<n - 1]$

by $simp$

also have $\dots = map\ (nth\ (x\ \# \ xs))\ (map\ Suc\ [m..<n - 1])$

by $simp$

also have $\dots = map\ (nth\ (x\ \# \ xs))\ [Suc\ m..<n]$

by ($metis\ Suc\text{-}diff\text{-}1\ le\text{-}0\text{-}eq\ length\text{-}upt\ list.\text{simps}(8)\ list.\text{size}(3)\ map\text{-}Suc\text{-}upt\ not\text{-}less\ upt\text{-}0$)

finally show $?thesis\ \dots$

qed

2.4.2 Sorted Lists

lemma *sorted-last* [simp]: $\llbracket x \in \text{set } xs; \text{sorted } xs \rrbracket \implies x \leq \text{last } xs$
 by (induct xs, auto)

lemma *sorted-prefix*:

assumes $xs \leq ys$ sorted ys

shows sorted xs

proof –

obtain zs where $ys = xs @ zs$

using *Prefix-Order.prefixE* assms(1) by auto

thus ?thesis

using assms(2) sorted-append by blast

qed

lemma *sorted-map*: $\llbracket \text{sorted } xs; \text{mono } f \rrbracket \implies \text{sorted } (\text{map } f \text{ } xs)$
 by (simp add: monoD sorted-iff-nth-mono)

lemma *sorted-distinct* [intro]: $\llbracket \text{sorted } (xs); \text{distinct}(xs) \rrbracket \implies (\forall i < \text{length } xs - 1. xs[i] < xs[i + 1])$
 apply (induct xs)
 apply (auto)
 apply (metis (no-types, hide-lams) Suc-leI Suc-less-eq Suc-pred gr0-conv-Suc not-le not-less-iff-gr-or-eq
 nth-Cons-Suc nth-mem nth-non-equal-first-eq)
 done

Is the given list a permutation of the given set?

definition *is-sorted-list-of-set* :: $(\text{'a}::\text{ord}) \text{ set} \Rightarrow \text{'a list} \Rightarrow \text{bool}$ **where**

is-sorted-list-of-set A xs = $((\forall i < \text{length}(xs) - 1. xs[i] < xs[i + 1])) \wedge \text{set}(xs) = A$

lemma *sorted-is-sorted-list-of-set*:

assumes *is-sorted-list-of-set* A xs

shows sorted(xs) and distinct(xs)

using assms **proof** (induct xs arbitrary: A)

show sorted []

by auto

next

show distinct []

by auto

next

fix A :: 'a set

case (Cons x xs) **note** hyps = this

assume isl: *is-sorted-list-of-set* A (x # xs)

hence srt: $(\forall i < \text{length } xs - \text{Suc } 0. xs[i] < xs[i + \text{Suc } i])$

using less-diff-conv **by** (auto simp add: *is-sorted-list-of-set-def*)

with hyps(1) **have** srt_d: sorted xs

by (simp add: *is-sorted-list-of-set-def*)

with isl **show** sorted (x # xs)

apply (auto simp add: *is-sorted-list-of-set-def*)

apply (metis (mono-tags, lifting) all-nth-imp-all-set less-le-trans linorder-not-less not-less-iff-gr-or-eq
 nth-Cons-0 sorted-iff-nth-mono zero-order(3))

done

from srt hyps(2) **have** distinct xs

by (simp add: *is-sorted-list-of-set-def*)

with isl **show** distinct (x # xs)

proof –

have $(\forall n. \neg n < \text{length } (x \# xs) - 1 \vee (x \# xs)[n] < (x \# xs)[n + 1]) \wedge \text{set } (x \# xs) = A$

by (*meson* $\langle \text{is-sorted-list-of-set } A \ (x \# \ xs) \rangle$ *is-sorted-list-of-set-def*)
then show *?thesis*
by (*metis* $\langle \text{distinct } xs \rangle$ *add.commute add-diff-cancel-left'* *distinct.simps(2)* *leD length-Cons length-greater-0-conv*
length-pos-if-in-set less-le nth-Cons-0 nth-Cons-Suc plus-1-eq-Suc set-ConsD sorted.elims(2) srtd)
qed
qed

lemma *is-sorted-list-of-set-alt-def*:
 $\text{is-sorted-list-of-set } A \ xs \longleftrightarrow \text{sorted } (xs) \wedge \text{distinct } (xs) \wedge \text{set}(xs) = A$
apply (*auto intro: sorted-is-sorted-list-of-set*)
apply (*auto simp add: is-sorted-list-of-set-def*)
apply (*metis Nat.add-0-right One-nat-def add-Suc-right sorted-distinct*)
done

definition *sorted-list-of-set-alt* :: $('a::\text{ord}) \text{ set} \Rightarrow 'a \text{ list}$ **where**
sorted-list-of-set-alt $A =$
(if $(A = \{\})$ *then* \square *else* $(\text{THE } xs. \text{is-sorted-list-of-set } A \ xs))$

lemma *is-sorted-list-of-set*:
 $\text{finite } A \Longrightarrow \text{is-sorted-list-of-set } A \ (\text{sorted-list-of-set } A)$
apply (*simp add: is-sorted-list-of-set-def*)
apply (*metis One-nat-def add.right-neutral add-Suc-right sorted-distinct sorted-list-of-set*)
done

lemma *sorted-list-of-set-other-def*:
 $\text{finite } A \Longrightarrow \text{sorted-list-of-set}(A) = (\text{THE } xs. \text{sorted}(xs) \wedge \text{distinct}(xs) \wedge \text{set } xs = A)$
apply (*rule sym*)
apply (*rule the-equality*)
apply (*auto*)
apply (*simp add: sorted-distinct-set-unique*)
done

lemma *sorted-list-of-set-alt [simp]*:
 $\text{finite } A \Longrightarrow \text{sorted-list-of-set-alt}(A) = \text{sorted-list-of-set}(A)$
apply (*rule sym*)
apply (*auto simp add: sorted-list-of-set-alt-def is-sorted-list-of-set-alt-def sorted-list-of-set-other-def*)
done

Sorting lists according to a relation

definition *is-sorted-list-of-set-by* :: $'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**
is-sorted-list-of-set-by $R \ A \ xs = ((\forall \ i < \text{length}(xs) - 1. (xs[i], xs[i + 1]) \in R) \wedge \text{set}(xs) = A)$

definition *sorted-list-of-set-by* :: $'a \text{ rel} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ list}$ **where**
sorted-list-of-set-by $R \ A = (\text{THE } xs. \text{is-sorted-list-of-set-by } R \ A \ xs)$

definition *fin-set-lexord* :: $'a \text{ rel} \Rightarrow 'a \text{ set rel}$ **where**
fin-set-lexord $R = \{(A, B). \text{finite } A \wedge \text{finite } B \wedge$
 $(\exists \ xs \ ys. \text{is-sorted-list-of-set-by } R \ A \ xs \wedge \text{is-sorted-list-of-set-by } R \ B \ ys$
 $\wedge (xs, ys) \in \text{lexord } R)\}$

lemma *is-sorted-list-of-set-by-mono*:
 $\llbracket R \subseteq S; \text{is-sorted-list-of-set-by } R \ A \ xs \rrbracket \Longrightarrow \text{is-sorted-list-of-set-by } S \ A \ xs$
by (*auto simp add: is-sorted-list-of-set-by-def*)

lemma *lexord-mono'*:

$\llbracket (\bigwedge x y. f x y \longrightarrow g x y); (xs, ys) \in \text{lexord } \{(x, y). f x y\} \rrbracket \Longrightarrow (xs, ys) \in \text{lexord } \{(x, y). g x y\}$
by (metis case-prodD case-prodI lexord-take-index-conv mem-Collect-eq)

lemma *fin-set-lexord-mono* [mono]:

$(\bigwedge x y. f x y \longrightarrow g x y) \Longrightarrow (xs, ys) \in \text{fin-set-lexord } \{(x, y). f x y\} \longrightarrow (xs, ys) \in \text{fin-set-lexord } \{(x, y). g x y\}$

proof

assume

fin: $(xs, ys) \in \text{fin-set-lexord } \{(x, y). f x y\}$ **and**

hyp: $(\bigwedge x y. f x y \longrightarrow g x y)$

from *fin* **have** *finite xs finite ys*

using *fin-set-lexord-def* **by** *fastforce+*

with *fin hyp* **show** $(xs, ys) \in \text{fin-set-lexord } \{(x, y). g x y\}$

apply (auto simp add: *fin-set-lexord-def*)

apply (rename-tac *xs' ys'*)

apply (rule-tac *x=xs'* **in** *exI*)

apply (auto)

apply (metis case-prodD case-prodI is-sorted-list-of-set-by-def mem-Collect-eq)

apply (metis case-prodD case-prodI is-sorted-list-of-set-by-def lexord-mono' mem-Collect-eq)

done

qed

definition *distincts* :: 'a set \Rightarrow 'a list set **where**

distincts *A* = $\{xs \in \text{lists } A. \text{distinct}(xs)\}$

lemma *tl-element*:

$\llbracket x \in \text{set } xs; x \neq \text{hd}(xs) \rrbracket \Longrightarrow x \in \text{set}(\text{tl}(xs))$

by (metis in-set-insert insert-Nil list.collapse list.distinct(2) set-ConsD)

2.4.3 List Update

lemma *listsum-update*:

fixes *xs* :: 'a::ring list

assumes *i* < length *xs*

shows *list-sum* (*xs*[*i* := *v*]) = *list-sum xs* - *xs* ! *i* + *v*

using *assms* **proof** (induct *xs* arbitrary: *i*)

case *Nil*

then show ?*case* **by** (*simp*)

next

case (*Cons a xs*)

then show ?*case*

proof (*cases i*)

case 0

thus ?*thesis*

by (*simp add: add.commute*)

next

case (*Suc i'*)

with *Cons* **show** ?*thesis*

by (*auto*)

qed

qed

2.4.4 Drop While and Take While

lemma *dropWhile-sorted-le-above*:

```

 $\llbracket \text{sorted } xs; x \in \text{set } (\text{dropWhile } (\lambda x. x \leq n) xs) \rrbracket \implies x > n$ 
apply (induct xs)
apply (auto)
apply (rename-tac a xs)
apply (case-tac a  $\leq n$ )
apply (auto)

```

done

lemma *set-dropWhile-le*:

```

 $\text{sorted } xs \implies \text{set } (\text{dropWhile } (\lambda x. x \leq n) xs) = \{x \in \text{set } xs. x > n\}$ 
apply (induct xs)
apply (simp)
apply (rename-tac x xs)
apply (subgoal-tac sorted xs)
apply (simp)
apply (safe)
apply (auto)

```

done

lemma *set-takeWhile-less-sorted*:

```

 $\llbracket \text{sorted } I; x \in \text{set } I; x < n \rrbracket \implies x \in \text{set } (\text{takeWhile } (\lambda x. x < n) I)$ 

```

proof (induct I arbitrary: x)

```

case Nil thus ?case
by (simp)

```

next

```

case (Cons a I) thus ?case
by auto

```

qed

lemma *nth-le-takeWhile-ord*: $\llbracket \text{sorted } xs; i \geq \text{length } (\text{takeWhile } (\lambda x. x \leq n) xs); i < \text{length } xs \rrbracket \implies n \leq xs ! i$

```

apply (induct xs arbitrary: i, auto)
apply (rename-tac x xs i)
apply (case-tac x  $\leq n$ )
apply (auto)
apply (metis One-nat-def Suc-eq-plus1 le-less-linear le-less-trans less-imp-le list.size(4) nth-mem
set-ConsD)
done

```

lemma *length-takeWhile-less*:

```

 $\llbracket a \in \text{set } xs; \neg P a \rrbracket \implies \text{length } (\text{takeWhile } P xs) < \text{length } xs$ 
by (metis in-set-conv-nth length-takeWhile-le nat-neq-iff not-less set-takeWhileD takeWhile-nth)

```

lemma *nth-length-takeWhile-less*:

```

 $\llbracket \text{sorted } xs; \text{distinct } xs; (\exists a \in \text{set } xs. a \geq n) \rrbracket \implies xs ! \text{length } (\text{takeWhile } (\lambda x. x < n) xs) \geq n$ 
by (induct xs, auto)

```

2.4.5 Last and But Last

lemma *length-gt-zero-butlast-concat*:

```

assumes length ys > 0
shows butlast (xs @ ys) = xs @ (butlast ys)
using assms by (metis butlast-append length-greater-0-conv)

```


lemma *length-eq-zero-butlast-concat*:
assumes *length ys = 0*
shows *butlast (xs @ ys) = butlast xs*
using *assms* **by** (*metis append-Nil2 length-0-conv*)

lemma *butlast-single-element*:
shows *butlast [e] = []*
by (*metis butlast.simps(2)*)

lemma *last-single-element*:
shows *last [e] = e*
by (*metis last.simps*)

lemma *length-zero-last-concat*:
assumes *length t = 0*
shows *last (s @ t) = last s*
by (*metis append-Nil2 assms length-0-conv*)

lemma *length-gt-zero-last-concat*:
assumes *length t > 0*
shows *last (s @ t) = last t*
by (*metis assms last-append length-greater-0-conv*)

2.4.6 Prefixes and Strict Prefixes

lemma *prefix-length-eq*:
 $\llbracket \text{length } xs = \text{length } ys; \text{prefix } xs \text{ } ys \rrbracket \implies xs = ys$
by (*metis not-equal-is-parallel parallel-def*)

lemma *prefix-Cons-elim* [*elim*]:
assumes *prefix (x # xs) ys*
obtains *ys' where ys = x # ys' prefix xs ys'*
using *assms*
by (*metis append-Cons prefix-def*)

lemma *prefix-map-inj*:
 $\llbracket \text{inj-on } f \text{ (set } xs \cup \text{set } ys); \text{prefix (map } f \text{ } xs) \text{ (map } f \text{ } ys) \rrbracket \implies$
 $\text{prefix } xs \text{ } ys$
apply (*induct xs arbitrary:ys*)
apply (*simp-all*)
apply (*erule prefix-Cons-elim*)
apply (*auto*)
apply (*metis image-insert insertI1 insert-Diff-if singletonE*)
done

lemma *prefix-map-inj-eq* [*simp*]:
 $\text{inj-on } f \text{ (set } xs \cup \text{set } ys) \implies$
 $\text{prefix (map } f \text{ } xs) \text{ (map } f \text{ } ys) \longleftrightarrow \text{prefix } xs \text{ } ys$
using *map-mono-prefix prefix-map-inj* **by** *blast*

lemma *strict-prefix-Cons-elim* [*elim*]:
assumes *strict-prefix (x # xs) ys*
obtains *ys' where ys = x # ys' strict-prefix xs ys'*
using *assms*
by (*metis Sublist.strict-prefixE' Sublist.strict-prefixI' append-Cons*)

```

lemma strict-prefix-map-inj:
   $\llbracket \text{inj-on } f \text{ (set } xs \cup \text{set } ys); \text{strict-prefix (map } f \text{ } xs) \text{ (map } f \text{ } ys) \rrbracket \implies$ 
     $\text{strict-prefix } xs \text{ } ys$ 
  apply (induct xs arbitrary:ys)
  apply (auto)
  using prefix-bot.bot.not-eq-extremum apply fastforce
  apply (erule strict-prefix-Cons-elim)
  apply (auto)
  apply (metis (hide-lams, full-types) image-insert insertI1 insert-Diff-if singletonE)
  done

lemma strict-prefix-map-inj-eq [simp]:
   $\text{inj-on } f \text{ (set } xs \cup \text{set } ys) \implies$ 
     $\text{strict-prefix (map } f \text{ } xs) \text{ (map } f \text{ } ys) \longleftrightarrow \text{strict-prefix } xs \text{ } ys$ 
  by (simp add: inj-on-map-eq-map strict-prefix-def)

lemma prefix-drop:
   $\llbracket \text{drop (length } xs) \text{ } ys = zs; \text{prefix } xs \text{ } ys \rrbracket$ 
     $\implies ys = xs @ zs$ 
  by (metis append-eq-conv-conj prefix-def)

lemma list-append-prefixD [dest]:  $x @ y \leq z \implies x \leq z$ 
  using append-prefixD less-eq-list-def by blast

lemma prefix-not-empty:
  assumes strict-prefix xs ys and xs  $\neq []$ 
  shows  $ys \neq []$ 
  using Sublist.strict-prefix-simps(1) assms(1) by blast

lemma prefix-not-empty-length-gt-zero:
  assumes strict-prefix xs ys and xs  $\neq []$ 
  shows  $\text{length } ys > 0$ 
  using assms prefix-not-empty by auto

lemma butlast-prefix-suffix-not-empty:
  assumes strict-prefix (butlast xs) ys
  shows  $ys \neq []$ 
  using assms prefix-not-empty-length-gt-zero by fastforce

lemma prefix-and-concat-prefix-is-concat-prefix:
  assumes prefix s t prefix (e @ t) u
  shows prefix (e @ s) u
  using Sublist.same-prefix-prefix assms(1) assms(2) prefix-order.dual-order.trans by blast

lemma prefix-eq-exists:
   $\text{prefix } s \text{ } t \longleftrightarrow (\exists xs . s @ xs = t)$ 
  using prefix-def by auto

lemma strict-prefix-eq-exists:
   $\text{strict-prefix } s \text{ } t \longleftrightarrow (\exists xs . s @ xs = t \wedge (\text{length } xs) > 0)$ 
  using prefix-def strict-prefix-def by auto

lemma butlast-strict-prefix-eq-butlast:
  assumes  $\text{length } s = \text{length } t$  and strict-prefix (butlast s) t

```

shows *strict-prefix* (butlast s) t \longleftrightarrow (butlast s) = (butlast t)
by (metis append-butlast-last-id append-eq-append-conv assms(1) assms(2) length-0-conv length-butlast
strict-prefix-eq-exists)

lemma *butlast-eq-if-eq-length-and-prefix*:
assumes *length* s > 0 *length* z > 0
length s = *length* z *strict-prefix* (butlast s) t *strict-prefix* (butlast z) t
shows (butlast s) = (butlast z)
using assms **by** (auto simp add: *strict-prefix-eq-exists*)

lemma *prefix-imp-length-lteq*:
assumes *prefix* s t
shows *length* s \leq *length* t
using assms **by** (simp add: *Sublist.prefix-length-le*)

lemma *prefix-imp-length-not-gt*:
assumes *prefix* s t
shows \neg (*length* t < *length* s)
using assms **by** (simp add: *Sublist.prefix-length-le leD*)

lemma *prefix-and-eq-length-imp-eq-list*:
assumes *prefix* s t **and** *length* t = *length* s
shows s = t
using assms **by** (simp add: *prefix-length-eq*)

lemma *butlast-prefix-imp-length-not-gt*:
assumes *length* s > 0 *strict-prefix* (butlast s) t
shows \neg (*length* t < *length* s)
using assms *prefix-length-less* **by** fastforce

lemma *length-not-gt-iff-eq-length*:
assumes *length* s > 0 **and** *strict-prefix* (butlast s) t
shows (\neg (*length* s < *length* t)) = (*length* s = *length* t)
proof –
have (\neg (*length* s < *length* t)) = ((*length* t < *length* s) \vee (*length* s = *length* t))
by (metis *not-less-iff-gr-or-eq*)
also have ... = (*length* s = *length* t)
using assms
by (simp add: *butlast-prefix-imp-length-not-gt*)

finally show ?thesis .
qed

Greatest common prefix

fun *gcp* :: 'a list \Rightarrow 'a list \Rightarrow 'a list **where**
gcp [] *ys* = [] |
gcp (x # *xs*) (y # *ys*) = (if (x = y) then x # *gcp xs ys* else []) |
gcp - - = []

lemma *gcp-right* [simp]: *gcp* *xs* [] = []
by (induct *xs*, auto)

lemma *gcp-append* [simp]: *gcp* (*xs* @ *ys*) (*xs* @ *zs*) = *xs* @ *gcp ys zs*
by (induct *xs*, auto)

```

lemma gcp-lb1: prefix (gcp xs ys) xs
  apply (induct xs arbitrary: ys, auto)
  apply (case-tac ys, auto)
done

```

```

lemma gcp-lb2: prefix (gcp xs ys) ys
  apply (induct ys arbitrary: xs, auto)
  apply (case-tac xs, auto)
done

```

interpretation *prefix-semilattice*: *semilattice-inf gcp prefix strict-prefix*

proof

```

  fix xs ys :: 'a list
  show prefix (gcp xs ys) xs
    by (induct xs arbitrary: ys, auto, case-tac ys, auto)
  show prefix (gcp xs ys) ys
    by (induct ys arbitrary: xs, auto, case-tac xs, auto)

```

next

```

  fix xs ys zs :: 'a list
  assume prefix xs ys prefix xs zs
  thus prefix xs (gcp ys zs)
    by (simp add: prefix-def, auto)

```

qed

2.4.7 Lexicographic Order

lemma *lexord-append*:

```

  assumes (xs1 @ ys1, xs2 @ ys2) ∈ lexord R length(xs1) = length(xs2)
  shows (xs1, xs2) ∈ lexord R ∨ (xs1 = xs2 ∧ (ys1, ys2) ∈ lexord R)
using assms
proof (induct xs2 arbitrary: xs1)
  case (Cons x2 xs2') note hyps = this
  from hyps(3) obtain x1 xs1' where xs1: xs1 = x1 # xs1' length(xs1') = length(xs2')
    by (auto, metis Suc-length-conv)
  with hyps(2) have xcases: (x1, x2) ∈ R ∨ (xs1' @ ys1, xs2' @ ys2) ∈ lexord R
    by (auto)
  show ?case
  proof (cases (x1, x2) ∈ R)
    case True with xs1 show ?thesis
      by (auto)
  next
    case False
    with xcases have (xs1' @ ys1, xs2' @ ys2) ∈ lexord R
      by (auto)
    with hyps(1) xs1 have dichot: (xs1', xs2') ∈ lexord R ∨ (xs1' = xs2' ∧ (ys1, ys2) ∈ lexord R)
      by (auto)
    have x1 = x2
      using False hyps(2) xs1(1) by auto
    with dichot xs1 show ?thesis
      by (simp)
  qed
next
  case Nil thus ?case
    by auto
qed

```

lemma *strict-prefix-lexord-rel*:

strict-prefix $xs\ ys \implies (xs, ys) \in \text{lexord } R$
by (*metis* *Sublist.strict-prefixE'* *lexord-append-rightI*)

lemma *strict-prefix-lexord-left*:

assumes *trans* $R\ (xs, ys) \in \text{lexord } R$ *strict-prefix* $xs'\ xs$
shows $(xs', ys) \in \text{lexord } R$
by (*metis* *assms lexord-trans strict-prefix-lexord-rel*)

lemma *prefix-lexord-right*:

assumes *trans* $R\ (xs, ys) \in \text{lexord } R$ *strict-prefix* $ys\ ys'$
shows $(xs, ys') \in \text{lexord } R$
by (*metis* *assms lexord-trans strict-prefix-lexord-rel*)

lemma *lexord-eq-length*:

assumes $(xs, ys) \in \text{lexord } R$ *length* $xs = \text{length } ys$
shows $\exists i. (xs!i, ys!i) \in R \wedge i < \text{length } xs \wedge (\forall j < i. xs!j = ys!j)$

using *assms* **proof** (*induct* xs *arbitrary*: ys)

case (*Cons* $x\ xs$) **note** *hyps* = *this*

then obtain $y\ ys'$ **where** $ys: ys = y \# ys'$ *length* $ys' = \text{length } xs$

by (*metis* *Suc-length-conv*)

show *?case*

proof (*cases* $(x, y) \in R$)

case *True* **with** ys **show** *?thesis*

by (*rule-tac* $x=0$ **in** *exI*, *simp*)

next

case *False*

with ys *hyps*(2) **have** $xy: x = y\ (xs, ys') \in \text{lexord } R$

by *auto*

with *hyps*(1,3) ys **obtain** i **where** $(xs!i, ys!i) \in R\ i < \text{length } xs\ (\forall j < i. xs!j = ys!j)$

by *force*

with $xy\ ys$ **show** *?thesis*

apply (*rule-tac* $x=\text{Suc } i$ **in** *exI*)

apply (*auto* *simp* *add: less-Suc-eq-0-disj*)

done

qed

next

case *Nil* **thus** *?case* **by** (*auto*)

qed

lemma *lexord-intro-elems*:

assumes *length* $xs > i$ *length* $ys > i$ $(xs!i, ys!i) \in R\ \forall j < i. xs!j = ys!j$

shows $(xs, ys) \in \text{lexord } R$

using *assms* **proof** (*induct* i *arbitrary*: $xs\ ys$)

case 0 **thus** *?case*

by (*auto*, *metis* *lexord-cons-cons list.exhaust nth-Cons-0*)

next

case (*Suc* i) **note** *hyps* = *this*

then obtain $x'\ y'\ xs'\ ys'$ **where** $xs = x' \# xs'\ ys = y' \# ys'$

by (*metis* *Suc-length-conv Suc-lessE*)

moreover with *hyps*(5) **have** $\forall j < i. xs'!j = ys'!j$

by (*auto*)

ultimately show *?case* **using** *hyps*

by (*auto*)

qed

2.5 Distributed Concatenation

definition $uncurry :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \times 'b \Rightarrow 'c)$ **where**
 $[simp]: uncurry f = (\lambda(x, y). f x y)$

definition $dist-concat ::$
 $'a \text{ list set} \Rightarrow 'a \text{ list set} \Rightarrow 'a \text{ list set}$ (**infixr** $\hat{\cap}$ 100) **where**
 $dist-concat \text{ ls1 } \text{ ls2} = (uncurry \text{ (@) } ' (\text{ ls1 } \times \text{ ls2}))$

lemma $dist-concat-left-empty$ $[simp]:$
 $\{\} \hat{\cap} \text{ ys} = \{\}$
by ($simp$ $add: dist-concat-def$)

lemma $dist-concat-right-empty$ $[simp]:$
 $\text{ xs } \hat{\cap} \{\} = \{\}$
by ($simp$ $add: dist-concat-def$)

lemma $dist-concat-insert$ $[simp]:$
 $insert \text{ l } \text{ ls1 } \hat{\cap} \text{ ls2} = ((\text{@}) \text{ l } ' (\text{ ls2})) \cup (\text{ ls1 } \hat{\cap} \text{ ls2})$
by ($auto$ $simp$ $add: dist-concat-def$)

2.6 List Domain and Range

abbreviation $seq-dom :: 'a \text{ list} \Rightarrow \text{ nat set } (dom_l)$ **where**
 $seq-dom \text{ xs} \equiv \{0..<length \text{ xs}\}$

abbreviation $seq-ran :: 'a \text{ list} \Rightarrow 'a \text{ set } (ran_l)$ **where**
 $seq-ran \text{ xs} \equiv \text{ set } \text{ xs}$

2.7 Extracting List Elements

definition $seq-extract :: \text{ nat set} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ (**infix** \upharpoonright_l 80) **where**
 $seq-extract \text{ A } \text{ xs} = \text{ nth s } \text{ xs } \text{ A}$

lemma $seq-extract-Nil$ $[simp]: \text{ A } \upharpoonright_l [] = []$
by ($simp$ $add: seq-extract-def$)

lemma $seq-extract-Cons:$
 $\text{ A } \upharpoonright_l (\text{ x } \# \text{ xs}) = (\text{ if } 0 \in \text{ A } \text{ then } [\text{ x }] \text{ else } []) \text{ @ } \{j. \text{ Suc } j \in \text{ A}\} \upharpoonright_l \text{ xs}$
by ($simp$ $add: seq-extract-def$ $nths-Cons$)

lemma $seq-extract-empty$ $[simp]: \{\} \upharpoonright_l \text{ xs} = []$
by ($simp$ $add: seq-extract-def$)

lemma $seq-extract-ident$ $[simp]: \{0..<length \text{ xs}\} \upharpoonright_l \text{ xs} = \text{ xs}$
unfolding $list-eq\text{-}iff\text{-}nth\text{-}eq$
by ($auto$ $simp$ $add: seq-extract-def$ $length\text{-}nths$ $atLeast0LessThan$)

lemma $seq-extract-split:$
assumes $i \leq length \text{ xs}$
shows $\{0..<i\} \upharpoonright_l \text{ xs} \text{ @ } \{i..<length \text{ xs}\} \upharpoonright_l \text{ xs} = \text{ xs}$
using $assms$
proof ($induct \text{ xs } arbitrary: i$)
case Nil **thus** $?case$ **by** ($simp$ $add: seq-extract-def$)
next
case (Cons x xs) **note** $hyp = this$

have $\{j. \text{Suc } j < i\} = \{0..<i - 1\}$
by (*auto*)
moreover have $\{j. i \leq \text{Suc } j \wedge j < \text{length } xs\} = \{i - 1..<\text{length } xs\}$
by (*auto*)
ultimately show *?case*
using *hyp* **by** (*force simp add: seq-extract-def nth-Cons*)
qed

lemma *seq-extract-append*:
 $A \upharpoonright_l (xs @ ys) = (A \upharpoonright_l xs) @ (\{j. j + \text{length } xs \in A\} \upharpoonright_l ys)$
by (*simp add: seq-extract-def nth-append*)

lemma *seq-extract-range*: $A \upharpoonright_l xs = (A \cap \text{dom}_l(xs)) \upharpoonright_l xs$
apply (*auto simp add: seq-extract-def nth-def*)
apply (*metis (no-types, lifting) atLeastLessThan-iff filter-cong in-set-zip nth-mem set-upt*)
done

lemma *seq-extract-out-of-range*:
 $A \cap \text{dom}_l(xs) = \{\} \implies A \upharpoonright_l xs = []$
by (*metis seq-extract-def seq-extract-range nth-empty*)

lemma *seq-extract-length* [*simp*]:
 $\text{length } (A \upharpoonright_l xs) = \text{card } (A \cap \text{dom}_l(xs))$
proof –
have $\{i. i < \text{length}(xs) \wedge i \in A\} = (A \cap \{0..<\text{length}(xs)\})$
by (*auto*)
thus *?thesis*
by (*simp add: seq-extract-def length-nth*)
qed

lemma *seq-extract-Cons-atLeastLessThan*:
assumes $m < n$
shows $\{m..<n\} \upharpoonright_l (x \# xs) = (\text{if } (m = 0) \text{ then } x \# (\{0..<n-1\} \upharpoonright_l xs) \text{ else } \{m-1..<n-1\} \upharpoonright_l xs)$
proof –
have $\{j. \text{Suc } j < n\} = \{0..<n - \text{Suc } 0\}$
by (*auto*)
moreover have $\{j. m \leq \text{Suc } j \wedge \text{Suc } j < n\} = \{m - \text{Suc } 0..<n - \text{Suc } 0\}$
by (*auto*)

ultimately show *?thesis* **using** *assms*
by (*auto simp add: seq-extract-Cons*)
qed

lemma *seq-extract-singleton*:
assumes $i < \text{length } xs$
shows $\{i\} \upharpoonright_l xs = [xs ! i]$
using *assms*
apply (*induct xs arbitrary: i*)
apply (*auto simp add: seq-extract-Cons*)
apply (*rename-tac xs i*)
apply (*subgoal-tac* $\{j. \text{Suc } j = i\} = \{i - 1\}$)
apply (*auto*)
done

lemma *seq-extract-as-map*:

```

  assumes  $m < n$   $n \leq \text{length } xs$ 
  shows  $\{m..<n\} \upharpoonright_l xs = \text{map } (\text{nth } xs) [m..<n]$ 
using assms proof (induct  $xs$  arbitrary:  $m$   $n$ )
  case Nil thus ?case by simp
next
  case (Cons  $x$   $xs$ )
  have  $[m..<n] = m \# [m+1..<n]$ 
    using Cons.prem1(1) upt-eq-Cons-conv by blast
  moreover have  $\text{map } (\text{nth } (x \# xs)) [Suc\ m..<n] = \text{map } (\text{nth } xs) [m..<n-1]$ 
    by (simp add: map-nth-Cons-atLeastLessThan)
  ultimately show ?case
    using Cons upt-rec
    by (auto simp add: seq-extract-Cons-atLeastLessThan)
qed

lemma seq-append-as-extract:
   $xs = ys @ zs \longleftrightarrow (\exists i \leq \text{length}(xs). ys = \{0..<i\} \upharpoonright_l xs \wedge zs = \{i..<\text{length}(xs)\} \upharpoonright_l xs)$ 
proof
  assume  $xs = ys @ zs$ 
  moreover have  $ys = \{0..<\text{length } ys\} \upharpoonright_l (ys @ zs)$ 
    by (simp add: seq-extract-append)
  moreover have  $zs = \{\text{length } ys..<\text{length } ys + \text{length } zs\} \upharpoonright_l (ys @ zs)$ 
  proof -
    have  $\{\text{length } ys..<\text{length } ys + \text{length } zs\} \cap \{0..<\text{length } ys\} = \{\}$ 
      by auto
    moreover have  $s1: \{j. j < \text{length } zs\} = \{0..<\text{length } zs\}$ 
      by auto
    ultimately show ?thesis
      by (simp add: seq-extract-append seq-extract-out-of-range)
  qed
  ultimately show  $(\exists i \leq \text{length}(xs). ys = \{0..<i\} \upharpoonright_l xs \wedge zs = \{i..<\text{length}(xs)\} \upharpoonright_l xs)$ 
    by (rule-tac  $x = \text{length } ys$  in exI, auto)
next
  assume  $\exists i \leq \text{length } xs. ys = \{0..<i\} \upharpoonright_l xs \wedge zs = \{i..<\text{length } xs\} \upharpoonright_l xs$ 
  thus  $xs = ys @ zs$ 
    by (auto simp add: seq-extract-split)
qed

```

2.8 Filtering a list according to a set

definition seq-filter :: 'a list \Rightarrow 'a set \Rightarrow 'a list (infix \upharpoonright_l 80) **where**
 seq-filter xs $A = \text{filter } (\lambda x. x \in A) xs$

lemma seq-filter-Cons-in [simp]:
 $x \in cs \implies (x \# xs) \upharpoonright_l cs = x \# (xs \upharpoonright_l cs)$
 by (simp add: seq-filter-def)

lemma seq-filter-Cons-out [simp]:
 $x \notin cs \implies (x \# xs) \upharpoonright_l cs = (xs \upharpoonright_l cs)$
 by (simp add: seq-filter-def)

lemma seq-filter-Nil [simp]: $[] \upharpoonright_l A = []$
 by (simp add: seq-filter-def)

lemma seq-filter-empty [simp]: $xs \upharpoonright_l \{\} = []$
 by (simp add: seq-filter-def)

lemma *seq-filter-append*: $(xs @ ys) \upharpoonright_l A = (xs \upharpoonright_l A) @ (ys \upharpoonright_l A)$
by (*simp add: seq-filter-def*)

lemma *seq-filter-UNIV* [*simp*]: $xs \upharpoonright_l UNIV = xs$
by (*simp add: seq-filter-def*)

lemma *seq-filter-twice* [*simp*]: $(xs \upharpoonright_l A) \upharpoonright_l B = xs \upharpoonright_l (A \cap B)$
by (*simp add: seq-filter-def*)

2.9 Minus on lists

instantiation *list* :: (*type*) *minus*
begin

We define list minus so that if the second list is not a prefix of the first, then an arbitrary list longer than the combined length is produced. Thus we can always determined from the output whether the minus is defined or not.

definition $xs - ys = (if (prefix\ ys\ xs) then\ drop\ (length\ ys)\ xs\ else\ [])$

instance ..
end

lemma *minus-cancel* [*simp*]: $xs - xs = []$
by (*simp add: minus-list-def*)

lemma *append-minus* [*simp*]: $(xs @ ys) - xs = ys$
by (*simp add: minus-list-def*)

lemma *minus-right-nil* [*simp*]: $xs - [] = xs$
by (*simp add: minus-list-def*)

lemma *list-concat-minus-list-concat*: $(s @ t) - (s @ z) = t - z$
by (*simp add: minus-list-def*)

lemma *length-minus-list*: $y \leq x \implies length(x - y) = length(x) - length(y)$
by (*simp add: less-eq-list-def minus-list-def*)

lemma *map-list-minus*:
 $xs \leq ys \implies map\ f\ (ys - xs) = map\ f\ ys - map\ f\ xs$
by (*simp add: drop-map less-eq-list-def map-mono-prefix minus-list-def*)

lemma *list-minus-first-tl* [*simp*]:
 $[x] \leq xs \implies (xs - [x]) = tl\ xs$
by (*metis Prefix-Order.prefixE append.left-neutral append-minus list.sel(3) not-Cons-self2 tl-append2*)

Extra lemmas about *prefix* and *strict-prefix*

lemma *prefix-concat-minus*:
assumes *prefix xs ys*
shows $xs @ (ys - xs) = ys$
using *assms* **by** (*metis minus-list-def prefix-drop*)

lemma *prefix-minus-concat*:
assumes *prefix s t*
shows $(t - s) @ z = (t @ z) - s$

using *assms* **by** (*simp* *add*: *Sublist.prefix-length-le minus-list-def*)

lemma *strict-prefix-minus-not-empty*:
assumes *strict-prefix xs ys*
shows $ys - xs \neq []$
using *assms* **by** (*metis* *append-Nil2 prefix-concat-minus strict-prefix-def*)

lemma *strict-prefix-diff-minus*:
assumes *prefix xs ys* **and** $xs \neq ys$
shows $(ys - xs) \neq []$
using *assms* **by** (*simp* *add*: *strict-prefix-minus-not-empty*)

lemma *length-tl-list-minus-butlast-gt-zero*:
assumes $\text{length } s < \text{length } t$ **and** *strict-prefix (butlast s) t* **and** $\text{length } s > 0$
shows $\text{length } (tl (t - (butlast s))) > 0$
using *assms*
by (*metis* *Nitpick.size-list-simp(2)* *butlast-snoc hd-Cons-tl length-butlast length-greater-0-conv length-tl less-trans nat-neq-iff strict-prefix-minus-not-empty prefix-order.dual-order.strict-implies-order prefix-concat-minus*)

lemma *list-minus-butlast-eq-butlast-list*:
assumes $\text{length } t = \text{length } s$ **and** *strict-prefix (butlast s) t*
shows $t - (butlast s) = [last\ t]$
using *assms*
by (*metis* *append-butlast-last-id append-eq-append-conv butlast.simps(1) length-butlast less-numeral-extra(3) list.size(3) prefix-order.dual-order.strict-implies-order prefix-concat-minus prefix-length-less*)

lemma *butlast-strict-prefix-length-lt-imp-last-tl-minus-butlast-eq-last*:
assumes $\text{length } s > 0$ *strict-prefix (butlast s) t* $\text{length } s < \text{length } t$
shows $last\ (tl (t - (butlast s))) = (last\ t)$
using *assms* **by** (*metis* *last-append last-tl length-tl-list-minus-butlast-gt-zero less-numeral-extra(3) list.size(3) append-minus strict-prefix-eq-exists*)

lemma *tl-list-minus-butlast-not-empty*:
assumes *strict-prefix (butlast s) t* **and** $\text{length } s > 0$ **and** $\text{length } t > \text{length } s$
shows $tl (t - (butlast s)) \neq []$
using *assms* *length-tl-list-minus-butlast-gt-zero* **by** *fastforce*

lemma *tl-list-minus-butlast-empty*:
assumes *strict-prefix (butlast s) t* **and** $\text{length } s > 0$ **and** $\text{length } t = \text{length } s$
shows $tl (t - (butlast s)) = []$
using *assms* **by** (*simp* *add*: *list-minus-butlast-eq-butlast-list*)

lemma *concat-minus-list-concat-butlast-eq-list-minus-butlast*:
assumes *prefix (butlast u) s*
shows $(t @ s) - (t @ (butlast u)) = s - (butlast u)$
using *assms* **by** (*metis* *append-assoc prefix-concat-minus append-minus*)

lemma *tl-list-minus-butlast-eq-empty*:
assumes *strict-prefix (butlast s) t* **and** $\text{length } s = \text{length } t$
shows $tl (t - (butlast s)) = []$
using *assms* **by** (*metis* *list.sel(3) list-minus-butlast-eq-butlast-list*)

lemma *prefix-length-tl-minus*:
assumes *strict-prefix s t*

shows $\text{length } (\text{tl } (t-s)) = (\text{length } (t-s)) - 1$
by (*auto*)

lemma *length-list-minus*:
assumes *strict-prefix s t*
shows $\text{length}(t - s) = \text{length}(t) - \text{length}(s)$
using *assms* **by** (*simp add: minus-list-def prefix-order.dual-order.strict-implies-order*)

2.10 Laws on take, drop, and nth

lemma *take-prefix*: $m \leq n \implies \text{take } m \text{ } xs \leq \text{take } n \text{ } xs$
by (*metis Prefix-Order.prefixI append-take-drop-id min-absorb2 take-append take-take*)

lemma *nth-atLeastAtMost-0-take*: $\text{nth } xs \{0..m\} = \text{take } (\text{Suc } m) \text{ } xs$
by (*metis atLeast0AtMost lessThan-Suc-atMost nth-upt-eq-take*)

lemma *nth-atLeastLessThan-0-take*: $\text{nth } xs \{0..<m\} = \text{take } m \text{ } xs$
by (*simp add: atLeast0LessThan*)

lemma *nth-atLeastAtMost-prefix*: $m \leq n \implies \text{nth } xs \{0..m\} \leq \text{nth } xs \{0..n\}$
by (*simp add: nth-atLeastAtMost-0-take take-prefix*)

lemma *sorted-nth-atLeastAtMost-0*: $\llbracket m \leq n; \text{sorted } (\text{nth } xs \{0..n\}) \rrbracket \implies \text{sorted } (\text{nth } xs \{0..m\})$
using *nth-atLeastAtMost-prefix sorted-prefix* **by** *blast*

lemma *sorted-nth-atLeastLessThan-0*: $\llbracket m \leq n; \text{sorted } (\text{nth } xs \{0..<n\}) \rrbracket \implies \text{sorted } (\text{nth } xs \{0..<m\})$
by (*metis atLeast0LessThan nth-upt-eq-take sorted-prefix take-prefix*)

lemma *list-augment-as-update*:
 $k < \text{length } xs \implies \text{list-augment } xs \text{ } k \text{ } x = \text{list-update } xs \text{ } k \text{ } x$
by (*metis list-augment-def list-augment-idem list-update-overwrite*)

lemma *nth-list-update-out*: $k \notin A \implies \text{nth } (\text{list-update } xs \text{ } k \text{ } x) \text{ } A = \text{nth } xs \text{ } A$
apply (*induct xs arbitrary: k x A*)
apply (*auto*)
apply (*rename-tac a xs k x A*)
apply (*case-tac k*)
apply (*auto simp add: nth-Cons*)
done

lemma *nth-list-augment-out*: $\llbracket k < \text{length } xs; k \notin A \rrbracket \implies \text{nth } (\text{list-augment } xs \text{ } k \text{ } x) \text{ } A = \text{nth } xs \text{ } A$
by (*simp add: list-augment-as-update nth-list-update-out*)

2.11 List power

overloading

listpow $\equiv \text{compow} :: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$
begin

fun *listpow* :: $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
 $\text{listpow } 0 \text{ } xs = []$
 $\mid \text{listpow } (\text{Suc } n) \text{ } xs = xs @ \text{listpow } n \text{ } xs$

end

lemma *listpow-Nil* [*simp*]: $[] ^ n = []$

```

by (induct n) simp-all

lemma listpow-Suc-right:  $xs \hat{=} \text{Suc } n = xs \hat{=} n @ xs$ 
by (induct n) simp-all

lemma listpow-add:  $xs \hat{=} (m + n) = xs \hat{=} m @ xs \hat{=} n$ 
by (induct m) simp-all

end

```

3 Infinite Sequences

theory *Sequence*

imports

HOL.Real

List-Extra

HOL-Library.Sublist

HOL-Library.Nat-Bijection

begin

```

typedef 'a seq = UNIV :: (nat  $\Rightarrow$  'a) set
by (auto)

```

setup-lifting *type-definition-seq*

```

definition ssubstr :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a seq  $\Rightarrow$  'a list where
ssubstr i j xs = map (Rep-seq xs) [i ..<j]

```

```

lift-definition nth-seq :: 'a seq  $\Rightarrow$  nat  $\Rightarrow$  'a (infixl !s 100)
is  $\lambda f i. f i$  .

```

```

abbreviation sinit :: nat  $\Rightarrow$  'a seq  $\Rightarrow$  'a list where
sinit i xs  $\equiv$  ssubstr 0 i xs

```

```

lemma sinit-len [simp]:
length (sinit i xs) = i
by (simp add: ssubstr-def)

```

```

lemma sinit-0 [simp]: sinit 0 xs = []
by (simp add: ssubstr-def)

```

```

lemma prefix-upt-0 [intro]:
 $i \leq j \implies \text{prefix } [0..<i] [0..<j]$ 
by (induct i, auto, metis append-prefixD le0 prefix-order.lift-Suc-mono-le prefix-order.order-refl upt-Suc)

```

```

lemma sinit-prefix:
 $i \leq j \implies \text{prefix } (sinit i xs) (sinit j xs)$ 
by (simp add: map-mono-prefix prefix-upt-0 ssubstr-def)

```

```

lemma sinit-strict-prefix:
 $i < j \implies \text{strict-prefix } (sinit i xs) (sinit j xs)$ 
by (metis sinit-len sinit-prefix le-less nat-neq-iff prefix-order.dual-order.strict-iff-order)

```

```

lemma nth-sinit:
 $i < n \implies sinit n xs ! i = xs !_s i$ 

```

```

  apply (auto simp add: ssubstr-def)
  apply (transfer, auto)
done

lemma sinit-append-split:
  assumes  $i < j$ 
  shows  $\text{sinit } j \text{ } xs = \text{sinit } i \text{ } xs @ \text{ssubstr } i \text{ } j \text{ } xs$ 
proof -
  have  $[0..<i] @ [i..<j] = [0..<j]$ 
    by (metis assms le0 le-add-diff-inverse le-less upt-add-eq-append)
  thus ?thesis
    by (auto simp add: ssubstr-def, transfer, simp add: map-append[THEN sym])
qed

lemma sinit-linear-asy-lemma1:
  assumes  $\text{asym } R \ i < j \ (\text{sinit } i \text{ } xs, \text{sinit } i \text{ } ys) \in \text{lexord } R \ (\text{sinit } j \text{ } ys, \text{sinit } j \text{ } xs) \in \text{lexord } R$ 
  shows False
proof -
  have sinit-xs:  $\text{sinit } j \text{ } xs = \text{sinit } i \text{ } xs @ \text{ssubstr } i \text{ } j \text{ } xs$ 
    by (metis assms(2) sinit-append-split)
  have sinit-ys:  $\text{sinit } j \text{ } ys = \text{sinit } i \text{ } ys @ \text{ssubstr } i \text{ } j \text{ } ys$ 
    by (metis assms(2) sinit-append-split)
  from sinit-xs sinit-ys assms(4)
  have  $(\text{sinit } i \text{ } ys, \text{sinit } i \text{ } xs) \in \text{lexord } R \vee (\text{sinit } i \text{ } ys = \text{sinit } i \text{ } xs \wedge (\text{ssubstr } i \text{ } j \text{ } ys, \text{ssubstr } i \text{ } j \text{ } xs) \in \text{lexord } R)$ 
    by (auto dest: lexord-append)
  with assms lexord-asymmetric show False
    by (force)
qed

lemma sinit-linear-asy-lemma2:
  assumes  $\text{asym } R \ (\text{sinit } i \text{ } xs, \text{sinit } i \text{ } ys) \in \text{lexord } R \ (\text{sinit } j \text{ } ys, \text{sinit } j \text{ } xs) \in \text{lexord } R$ 
  shows False
proof (cases  $i \text{ } j$  rule: linorder-cases)
  case less with assms show ?thesis
    by (auto dest: sinit-linear-asy-lemma1)
next
  case equal with assms show ?thesis
    by (simp add: lexord-asymmetric)
next
  case greater with assms show ?thesis
    by (auto dest: sinit-linear-asy-lemma1)
qed

lemma range-ext:
  assumes  $\forall i :: \text{nat}. \forall x \in \{0..<i\}. f(x) = g(x)$ 
  shows  $f = g$ 
proof (rule ext)
  fix  $x :: \text{nat}$ 
  obtain  $i :: \text{nat}$  where  $i > x$ 
    by (metis lessI)
  with assms show  $f(x) = g(x)$ 
    by (auto)
qed

```

lemma *sinit-ext*:

$(\forall i. \text{sinit } i \text{ } xs = \text{sinit } i \text{ } ys) \implies xs = ys$
by (*simp add: ssubstr-def, transfer, auto intro: range-ext*)

definition *seq-lexord* :: 'a rel \Rightarrow ('a seq) rel **where**

seq-lexord $R = \{(xs, ys). (\exists i. (\text{sinit } i \text{ } xs, \text{sinit } i \text{ } ys) \in \text{lexord } R))\}$

lemma *seq-lexord-irreflexive*:

$\forall x. (x, x) \notin R \implies (xs, xs) \notin \text{seq-lexord } R$
by (*auto dest: lexord-irreflexive simp add: irrefl-def seq-lexord-def*)

lemma *seq-lexord-irrefl*:

irrefl $R \implies \text{irrefl } (\text{seq-lexord } R)$
by (*simp add: irrefl-def seq-lexord-irreflexive*)

lemma *seq-lexord-transitive*:

assumes *trans* R
shows *trans* (*seq-lexord* R)

unfolding *seq-lexord-def*

proof (*rule transI, clarify*)

fix *xs ys zs* :: 'a seq **and** *m n* :: nat

assume *las*: $(\text{sinit } m \text{ } xs, \text{sinit } m \text{ } ys) \in \text{lexord } R$ $(\text{sinit } n \text{ } ys, \text{sinit } n \text{ } zs) \in \text{lexord } R$

hence *inz*: $m > 0$

using *gr0I* **by** *force*

from *las*(1) **obtain** *i* **where** *sinitm*: $(\text{sinit } m \text{ } xs!i, \text{sinit } m \text{ } ys!i) \in R$ $i < m \forall j < i. \text{sinit } m \text{ } xs!j = \text{sinit } m \text{ } ys!j$

using *lexord-eq-length* **by** *force*

from *las*(2) **obtain** *j* **where** *sinitn*: $(\text{sinit } n \text{ } ys!j, \text{sinit } n \text{ } zs!j) \in R$ $j < n \forall k < j. \text{sinit } n \text{ } ys!k = \text{sinit } n \text{ } zs!k$

using *lexord-eq-length* **by** *force*

show $\exists i. (\text{sinit } i \text{ } xs, \text{sinit } i \text{ } zs) \in \text{lexord } R$

proof (*cases* $i \leq j$)

case *True* **note** *lt* = *this*

with *sinitm sinitn* **have** $(\text{sinit } n \text{ } xs!i, \text{sinit } n \text{ } zs!i) \in R$

by (*metis assms le-eq-less-or-eq le-less-trans nth-sinit transD*)

moreover from *lt sinitm sinitn* **have** $\forall j < i. \text{sinit } m \text{ } xs!j = \text{sinit } m \text{ } zs!j$

by (*metis less-le-trans less-trans nth-sinit*)

ultimately have $(\text{sinit } n \text{ } xs, \text{sinit } n \text{ } zs) \in \text{lexord } R$ **using** *sinitm*(2) *sinitn*(2) *lt*

apply (*rule-tac lexord-intro-elems*)

apply (*auto*)

apply (*metis less-le-trans less-trans nth-sinit*)

done

thus *?thesis* **by** *auto*

next

case *False*

then have *ge*: $i > j$ **by** *auto*

with *assms sinitm sinitn* **have** $(\text{sinit } n \text{ } xs!j, \text{sinit } n \text{ } zs!j) \in R$

by (*metis less-trans nth-sinit*)

moreover from *ge sinitm sinitn* **have** $\forall k < j. \text{sinit } m \text{ } xs!k = \text{sinit } m \text{ } zs!k$

by (*metis dual-order.strict-trans nth-sinit*)

ultimately have $(\text{sinit } n \text{ } xs, \text{sinit } n \text{ } zs) \in \text{lexord } R$ **using** *sinitm*(2) *sinitn*(2) *ge*

apply (*rule-tac lexord-intro-elems*)

apply (*auto*)

apply (*metis less-trans nth-sinit*)

done

thus ?thesis by auto
 qed
 qed

lemma *seq-lexord-trans*:
 $\llbracket (xs, ys) \in seq\text{-}lexord\ R; (ys, zs) \in seq\text{-}lexord\ R; trans\ R \rrbracket \implies (xs, zs) \in seq\text{-}lexord\ R$
 by (meson seq-lexord-transitive transE)

lemma *seq-lexord-antisym*:
 $\llbracket asym\ R; (a, b) \in seq\text{-}lexord\ R \rrbracket \implies (b, a) \notin seq\text{-}lexord\ R$
 by (auto dest: sinit-linear-asym-lemma2 simp add: seq-lexord-def)

lemma *seq-lexord-asym*:
 assumes *asym* *R*
 shows *asym* (*seq-lexord* *R*)
 by (meson assms *asym*.simps seq-lexord-antisym seq-lexord-irrefl)

lemma *seq-lexord-total*:
 assumes *total* *R*
 shows *total* (*seq-lexord* *R*)
 using assms by (auto simp add: total-on-def seq-lexord-def, meson lexord-linear sinit-ext)

lemma *seq-lexord-strict-linear-order*:
 assumes *strict-linear-order* *R*
 shows *strict-linear-order* (*seq-lexord* *R*)
 using assms
 by (auto simp add: strict-linear-order-on-def partial-order-on-def preorder-on-def
 intro: seq-lexord-transitive seq-lexord-irrefl seq-lexord-total)

lemma *seq-lexord-linear*:
 assumes $(\forall\ a\ b. (a, b) \in R \vee a = b \vee (b, a) \in R)$
 shows $(x, y) \in seq\text{-}lexord\ R \vee x = y \vee (y, x) \in seq\text{-}lexord\ R$
proof –
 have *total* *R*
 using assms total-on-def by blast
 hence *total* (*seq-lexord* *R*)
 using seq-lexord-total by blast
 thus ?thesis
 by (auto simp add: total-on-def)
 qed

instantiation *seq* :: (*ord*) *ord*
begin

definition *less-seq* :: '*a* *seq* \Rightarrow '*a* *seq* \Rightarrow bool **where**
less-seq *xs* *ys* $\longleftrightarrow (xs, ys) \in seq\text{-}lexord\ \{(xs, ys). xs < ys\}$

definition *less-eq-seq* :: '*a* *seq* \Rightarrow '*a* *seq* \Rightarrow bool **where**
less-eq-seq *xs* *ys* = (*xs* = *ys* \vee *xs* < *ys*)

instance ..

end

instance *seq* :: (*order*) *order*

```

proof
  fix  $xs :: 'a \text{ seq}$ 
  show  $xs \leq xs$  by (simp add: less-eq-seq-def)
next
  fix  $xs \ ys \ zs :: 'a \text{ seq}$ 
  assume  $xs \leq ys$  and  $ys \leq zs$ 
  then show  $xs \leq zs$ 
    by (force dest: seq-lexord-trans simp add: less-eq-seq-def less-seq-def trans-def)
next
  fix  $xs \ ys :: 'a \text{ seq}$ 
  assume  $xs \leq ys$  and  $ys \leq xs$ 
  then show  $xs = ys$ 
    apply (auto simp add: less-eq-seq-def less-seq-def)
    apply (rule seq-lexord-irreflexive [THEN notE])
    defer
    apply (rule seq-lexord-trans)
    apply (auto intro: transI)
    done
next
  fix  $xs \ ys :: 'a \text{ seq}$ 
  show  $xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs$ 
    apply (auto simp add: less-seq-def less-eq-seq-def)
    defer
    apply (rule seq-lexord-irreflexive [THEN notE])
    apply auto
    apply (rule seq-lexord-irreflexive [THEN notE])
    defer
    apply (rule seq-lexord-trans)
    apply (auto intro: transI)
    apply (simp add: seq-lexord-irreflexive)
    done
qed

instance seq :: (linorder) linorder
proof
  fix  $xs \ ys :: 'a \text{ seq}$ 
  have  $(xs, ys) \in \text{seq-lexord } \{(u, v). u < v\} \vee xs = ys \vee (ys, xs) \in \text{seq-lexord } \{(u, v). u < v\}$ 
    by (rule seq-lexord-linear) auto
  then show  $xs \leq ys \vee ys \leq xs$ 
    by (auto simp add: less-eq-seq-def less-seq-def)
qed

lemma seq-lexord-mono [mono]:
   $(\bigwedge x \ y. f \ x \ y \longrightarrow g \ x \ y) \Longrightarrow (xs, ys) \in \text{seq-lexord } \{(x, y). f \ x \ y\} \longrightarrow (xs, ys) \in \text{seq-lexord } \{(x, y). g \ x \ y\}$ 
  apply (auto simp add: seq-lexord-def)
  apply (metis case-prodD case-prodI lexord-take-index-conv mem-Collect-eq)
  done

fun insort-rel :: 'a rel  $\Rightarrow$  'a  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
insort-rel R x [] = [x] |
insort-rel R x (y # ys) = (if ( $x = y \vee (x, y) \in R$ ) then  $x \# y \# ys$  else  $y \# \text{insort-rel } R \ x \ ys$ )

inductive sorted-rel :: 'a rel  $\Rightarrow$  'a list  $\Rightarrow$  bool where
Nil-rel [iff]: sorted-rel R [] |

```


Cons-rel: $\forall y \in \text{set } xs. (x = y \vee (x, y) \in R) \implies \text{sorted-rel } R \text{ } xs \implies \text{sorted-rel } R (x \# xs)$

definition *list-of-set* :: 'a rel \Rightarrow 'a set \Rightarrow 'a list **where**
list-of-set $R = \text{folding.F } (\text{insort-rel } R) []$

lift-definition *seq-inj* :: 'a seq seq \Rightarrow 'a seq **is**
 $\lambda f i. f (\text{fst } (\text{prod-decode } i)) (\text{snd } (\text{prod-decode } i))$.

lift-definition *seq-proj* :: 'a seq \Rightarrow 'a seq seq **is**
 $\lambda f i j. f (\text{prod-encode } (i, j))$.

lemma *seq-inj-inverse*: *seq-proj* (*seq-inj* x) = x
by (*transfer*, *simp*)

lemma *seq-proj-inverse*: *seq-inj* (*seq-proj* x) = x
by (*transfer*, *simp*)

lemma *seq-inj*: *inj seq-inj*
by (*metis injI seq-inj-inverse*)

lemma *seq-inj-surj*: *bij seq-inj*
apply (*rule bijI*)
apply (*auto simp add: seq-inj*)
apply (*metis rangeI seq-proj-inverse*)
done
end

4 Finite Sets: extra functions and properties

theory *FSet-Extra*
imports
HOL-Library.FSet
HOL-Library.Countable-Set-Type
begin

setup-lifting *type-definition-fset*

notation *fempty* ($\{\!\!\}$)
notation *fset* ($\langle\!\!\rangle_f$)
notation *fminus* (**infixl** $-_f$ 65)

syntax
 $-FinFset :: \text{args} \Rightarrow 'a \text{ fset} \quad (\{\!(-)\!\})$

translations
 $\{\!x, xs\!\} == \text{CONST } \text{finsert } x \{\!xs\!\}$
 $\{\!x\!\} == \text{CONST } \text{finsert } x \{\!\!\}$

term *fBall*

syntax
 $-fBall :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \forall \text{ -}|\in| \text{ -} / \text{ -}) [0, 0, 10] 10)$
 $-fBex :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \exists \text{ -}|\in| \text{ -} / \text{ -}) [0, 0, 10] 10)$

translations

$\forall x \in A. P == \text{CONST } fBall\ A\ (\%x. P)$
 $\exists x \in A. P == \text{CONST } fBex\ A\ (\%x. P)$

definition *FUnion* :: 'a fset fset \Rightarrow 'a fset ($\bigcup_f [90]$ 90) **where**
FUnion *xs* = *Abs-fset* ($\bigcup_{x \in \langle xs \rangle_f} \langle x \rangle_f$)

definition *FInter* :: 'a fset fset \Rightarrow 'a fset ($\bigcap_f [90]$ 90) **where**
FInter *xs* = *Abs-fset* ($\bigcap_{x \in \langle xs \rangle_f} \langle x \rangle_f$)

Finite power set

definition *FinPow* :: 'a fset \Rightarrow 'a fset fset **where**
FinPow *xs* = *Abs-fset* (*Abs-fset* ' Pow $\langle xs \rangle_f$)

Set of all finite subsets of a set

definition *Fow* :: 'a set \Rightarrow 'a fset set **where**
Fow *A* = $\{x. \langle x \rangle_f \subseteq A\}$

declare *Abs-fset-inverse* [*simp*]

lemma *fset-intro*:

$fset\ x = fset\ y \implies x = y$
by (*simp add:fset-inject*)

lemma *fset-elim*:

$\llbracket x = y; fset\ x = fset\ y \implies P \rrbracket \implies P$
by (*auto*)

lemma *fmember-intro*:

$\llbracket x \in fset(xs) \rrbracket \implies x \in xs$
by (*metis fmember.rep-eq*)

lemma *fmember-elim*:

$\llbracket x \in xs; x \in fset(xs) \implies P \rrbracket \implies P$
by (*metis fmember.rep-eq*)

lemma *fnmember-intro* [*intro*]:

$\llbracket x \notin fset(xs) \rrbracket \implies x \notin xs$
by (*metis fmember.rep-eq*)

lemma *fnmember-elim* [*elim*]:

$\llbracket x \notin xs; x \notin fset(xs) \implies P \rrbracket \implies P$
by (*metis fmember.rep-eq*)

lemma *fsubset-intro* [*intro*]:

$\langle xs \rangle_f \subseteq \langle ys \rangle_f \implies xs \subseteq ys$
by (*metis less-eq-fset.rep-eq*)

lemma *fsubset-elim* [*elim*]:

$\llbracket xs \subseteq ys; \langle xs \rangle_f \subseteq \langle ys \rangle_f \implies P \rrbracket \implies P$
by (*metis less-eq-fset.rep-eq*)

lemma *fBall-intro* [*intro*]:

$Ball\ \langle A \rangle_f\ P \implies fBall\ A\ P$
by (*metis (poly-guards-query) fBallI fmember.rep-eq*)

```

lemma fBall-elim [elim]:
   $\llbracket fBall\ A\ P; Ball\ \langle A \rangle_f\ P \implies Q \rrbracket \implies Q$ 
  by (metis fBallE fmember.rep-eq)

```

lift-definition *finset* :: 'a list \Rightarrow 'a fset **is** set ..

```

context linorder
begin

```

```

lemma sorted-list-of-set-inj:
   $\llbracket finite\ xs; finite\ ys; sorted-list-of-set\ xs = sorted-list-of-set\ ys \rrbracket$ 
   $\implies xs = ys$ 
  apply (simp add:sorted-list-of-set-def)
  apply (induct xs rule:finite-induct)
  apply (induct ys rule:finite-induct)
  apply (simp-all)
  apply (metis finite.insertI insert-not-empty sorted-list-of-set-def sorted-list-of-set-empty sorted-list-of-set-eq-Nil-iff)
  apply (metis finite.insertI finite-list set-remdups set-sort sorted-list-of-set-def sorted-list-of-set-sort-remdups)
  done

```

definition *flist* :: 'a fset \Rightarrow 'a list **where**
flist *xs* = *sorted-list-of-set* (*fset* *xs*)

```

lemma flist-inj: inj flist
  apply (simp add:flist-def inj-on-def)
  apply (clarify)
  apply (rename-tac x y)
  apply (subgoal-tac fset x = fset y)
  apply (simp add:fset-inject)
  apply (rule sorted-list-of-set-inj, simp-all)
  done

```

```

lemma flist-props [simp]:
  sorted (flist xs)
  distinct (flist xs)
  by (simp-all add:flist-def)

```

```

lemma flist-empty [simp]:
  flist  $\{\}$  =  $\square$ 
  by (simp add:flist-def)

```

```

lemma flist-inv [simp]: finset (flist xs) = xs
  by (simp add:finset-def flist-def fset-inverse)

```

```

lemma flist-set [simp]: set (flist xs) = fset xs
  by (simp add:finset-def flist-def fset-inverse)

```

```

lemma fset-inv [simp]:  $\llbracket sorted\ xs; distinct\ xs \rrbracket \implies flist\ (finset\ xs) = xs$ 
  apply (simp add:finset-def flist-def fset-inverse)
  apply (metis local.sorted-list-of-set-sort-remdups local.sorted-sort-id remdups-id-iff-distinct)
  done

```

```

lemma fcard-flist:
  fcard xs = length (flist xs)
  apply (simp add:fcard-def)

```

```

apply (fold flist-set)
apply (unfold distinct-card[OF flist-props(2)])
apply (rule refl)
done

lemma flist-nth:
   $i < \text{fcard } vs \implies \text{flist } vs ! i \in vs$ 
apply (simp add: fmember-def flist-def fcard-def)
apply (metis fcard.rep-eq fcard-flist finset.rep-eq flist-def flist-inv nth-mem)
done

definition fmax :: 'a fset  $\Rightarrow$  'a where
  fmax xs = (if (xs =  $\{\}$ ) then undefined else last (flist xs))

end

definition flists :: 'a fset  $\Rightarrow$  'a list set where
  flists A = {xs. distinct xs  $\wedge$  finset xs = A}

lemma flists-nonempty:  $\exists xs. xs \in \text{flists } A$ 
apply (simp add: flists-def)
apply (metis Abs-fset-cases Abs-fset-inverse finite-distinct-list finite-fset finset.rep-eq)
done

lemma flists-elem-uniq:  $\llbracket x \in \text{flists } A; x \in \text{flists } B \rrbracket \implies A = B$ 
by (simp add: flists-def)

definition flist-arb :: 'a fset  $\Rightarrow$  'a list where
  flist-arb A = (SOME xs. xs  $\in$  flists A)

lemma flist-arb-distinct [simp]: distinct (flist-arb A)
by (metis (mono-tags) flist-arb-def flists-def flists-nonempty mem-Collect-eq someI-ex)

lemma flist-arb-inv [simp]: finset (flist-arb A) = A
by (metis (mono-tags) flist-arb-def flists-def flists-nonempty mem-Collect-eq someI-ex)

lemma flist-arb-inj:
  inj flist-arb
by (metis flist-arb-inv injI)

lemma flist-arb-lists: flist-arb 'Fow A  $\subseteq$  lists A
apply (auto)
using Fow-def finset.rep-eq apply fastforce
done

lemma countable-Fow:
  fixes A :: 'a set
  assumes countable A
  shows countable (Fow A)
proof -
from assms obtain to-nat-list :: 'a list  $\Rightarrow$  nat where inj-on to-nat-list (lists A)
by blast
thus ?thesis
apply (simp add: countable-def)
apply (rule-tac x=to-nat-list  $\circ$  flist-arb in exI)

```

apply (*rule comp-inj-on*)
apply (*metis flist-arb-inv inj-on-def*)
apply (*simp add: flist-arb-lists subset-inj-on*)
done
qed

definition *flub* :: 'a fset set \Rightarrow 'a fset \Rightarrow 'a fset **where**
flub A t = (if ($\forall a \in A. a \sqsubseteq t$) then Abs-fset ($\bigcup x \in A. \langle x \rangle_f$) else t)

lemma *finite-Union-subsets*:

$\llbracket \forall a \in A. a \subseteq b; \text{finite } b \rrbracket \Longrightarrow \text{finite } (\bigcup A)$
by (*metis Sup-le-iff finite-subset*)

lemma *finite-UN-subsets*:

$\llbracket \forall a \in A. B a \subseteq b; \text{finite } b \rrbracket \Longrightarrow \text{finite } (\bigcup a \in A. B a)$
by (*metis UN-subset-iff finite-subset*)

lemma *flub-rep-eq*:

$\langle \text{flub } A \ t \rangle_f = (\text{if } (\forall a \in A. a \sqsubseteq t) \text{ then } (\bigcup x \in A. \langle x \rangle_f) \text{ else } \langle t \rangle_f)$
apply (*subgoal-tac* (if ($\forall a \in A. a \sqsubseteq t$) then $(\bigcup x \in A. \langle x \rangle_f)$ else $\langle t \rangle_f \in \{x. \text{finite } x\}$))
apply (*auto simp add: flub-def*)
apply (*rule finite-UN-subsets[of - - $\langle t \rangle_f$]*)
apply (*auto*)
done

definition *fglb* :: 'a fset set \Rightarrow 'a fset \Rightarrow 'a fset **where**

fglb A t = (if ($A = \{\}$) then t else Abs-fset ($\bigcap x \in A. \langle x \rangle_f$))

lemma *fglb-rep-eq*:

$\langle \text{fglb } A \ t \rangle_f = (\text{if } (A = \{\}) \text{ then } \langle t \rangle_f \text{ else } (\bigcap x \in A. \langle x \rangle_f))$
apply (*subgoal-tac* (if ($A = \{\}$) then $\langle t \rangle_f$ else $(\bigcap x \in A. \langle x \rangle_f) \in \{x. \text{finite } x\}$))
apply (*metis Abs-fset-inverse fglb-def*)
apply (*auto*)
apply (*metis finite-INT finite-fset*)
done

lemma *FinPow-rep-eq [simp]*:

$\text{fset } (\text{FinPow } xs) = \{ys. ys \sqsubseteq xs\}$
apply (*subgoal-tac finite (Abs-fset 'Pow $\langle xs \rangle_f$)*)
apply (*auto simp add: fmember-def FinPow-def*)
apply (*rename-tac x' y'*)
apply (*subgoal-tac finite x'*)
apply (*auto*)
apply (*metis finite-fset finite-subset*)
apply (*metis (full-types) Pow-iff fset-inverse imageI less-eq-fset.rep-eq*)
done

lemma *FUnion-rep-eq [simp]*:

$\langle \bigcup_f xs \rangle_f = (\bigcup x \in \langle xs \rangle_f. \langle x \rangle_f)$
by (*simp add: FUnion-def*)

lemma *FInter-rep-eq [simp]*:

$xs \neq \{\} \Longrightarrow \langle \bigcap_f xs \rangle_f = (\bigcap x \in \langle xs \rangle_f. \langle x \rangle_f)$
apply (*simp add: FInter-def*)
apply (*subgoal-tac finite ($\bigcap x \in \langle xs \rangle_f. \langle x \rangle_f$)*)

```

  apply (simp)
  apply (metis (poly-guards-query) bot-fset.rep-eq fglb-rep-eq finite-fset fset-inverse)
done

```

```

lemma FUnion-empty [simp]:
   $\bigcup_f \{\} = \{\}$ 
  by (auto simp add:FUnion-def fmember-def)

```

```

lemma FinPow-member [simp]:
   $xs \in FinPow\ xs$ 
  by (auto simp add:fmember-def)

```

```

lemma FUnion-FinPow [simp]:
   $\bigcup_f (FinPow\ x) = x$ 
  by (auto simp add:fmember-def less-eq-fset-def)

```

```

lemma Fow-mem [iff]:  $x \in Fow\ A \longleftrightarrow \langle x \rangle_f \subseteq A$ 
  by (auto simp add:Fow-def)

```

```

lemma Fow-UNIV [simp]:  $Fow\ UNIV = UNIV$ 
  by (simp add:Fow-def)

```

```

lift-definition FMax :: ('a::linorder) fset  $\Rightarrow$  'a is Max .

```

```

end

```

5 Countable Sets: Extra functions and properties

```

theory Countable-Set-Extra
imports
  HOL-Library.Countable-Set-Type
  Sequence
  FSet-Extra
  HOL-Library.Bit
begin

```

5.1 Extra syntax

```

notation cempty ( $\{\}_c$ )
notation cin (infix  $\in_c$  50)
notation cUn (infixl  $\cup_c$  65)
notation cInt (infixl  $\cap_c$  70)
notation cDiff (infixl  $-_c$  65)
notation cUnion ( $\bigcup_c$  [900] 900)
notation cimage (infixr  $'_c$  90)

```

```

abbreviation csubseq :: 'a cset  $\Rightarrow$  'a cset  $\Rightarrow$  bool ((-/  $\subseteq_c$  -) [51, 51] 50)
where  $A \subseteq_c B \equiv A \leq B$ 

```

```

abbreviation csubset :: 'a cset  $\Rightarrow$  'a cset  $\Rightarrow$  bool ((-/  $\subset_c$  -) [51, 51] 50)
where  $A \subset_c B \equiv A < B$ 

```

5.2 Countable set functions

```

setup-lifting type-definition-cset

```

lift-definition *cnin* :: 'a \Rightarrow 'a cset \Rightarrow bool (**infix** \notin_c 50) **is** (\notin) .

definition *cBall* :: 'a cset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **where**
cBall A P = ($\forall x. x \in_c A \longrightarrow P x$)

definition *cBex* :: 'a cset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **where**
cBex A P = ($\exists x. x \in_c A \longrightarrow P x$)

declare *cBall-def* [*mono, simp*]
declare *cBex-def* [*mono, simp*]

syntax

-*cBall* :: *pttrn* \Rightarrow 'a cset \Rightarrow bool \Rightarrow bool (($\exists \forall$ \in_c ./ -) [0, 0, 10] 10)
-*cBex* :: *pttrn* \Rightarrow 'a cset \Rightarrow bool \Rightarrow bool (($\exists \exists$ \in_c ./ -) [0, 0, 10] 10)

translations

$\forall x \in_c A. P == \text{CONST } cBall \ A \ (\%x. P)$
 $\exists x \in_c A. P == \text{CONST } cBex \ A \ (\%x. P)$

definition *cset-Collect* :: ('a \Rightarrow bool) \Rightarrow 'a cset **where**
cset-Collect = (*acset* o *Collect*)

lift-definition *cset-Coll* :: 'a cset \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a cset **is** $\lambda A P. \{x \in A. P x\}$
by (*auto*)

lemma *cset-Coll-equiv*: *cset-Coll* A P = *cset-Collect* ($\lambda x. x \in_c A \wedge P x$)
by (*simp add: cset-Collect-def cset-Coll-def cin-def*)

declare *cset-Collect-def* [*simp*]

syntax

-*cColl* :: *pttrn* \Rightarrow bool \Rightarrow 'a cset ((1{./ -}_c)

translations

$\{x . P\}_c \equiv (\text{CONST } cset-Collect) (\lambda x . P)$

syntax (*xsymbols*)

-*cCollect* :: *pttrn* \Rightarrow 'a cset \Rightarrow bool \Rightarrow 'a cset ((1{ \in_c ./ -}_c)

translations

$\{x \in_c A. P\}_c \Rightarrow \text{CONST } cset-Coll \ A \ (\lambda x. P)$

lemma *cset-CollectI*: $P (a :: 'a::countable) \Longrightarrow a \in_c \{x. P x\}_c$
by (*simp add: cin-def*)

lemma *cset-CollI*: $\llbracket a \in_c A; P a \rrbracket \Longrightarrow a \in_c \{x \in_c A. P x\}_c$
by (*simp add: cin.rep-eq cset-Coll.rep-eq*)

lemma *cset-CollectD*: $(a :: 'a::countable) \in_c \{x. P x\}_c \Longrightarrow P a$
by (*simp add: cin-def*)

lemma *cset-Collect-cong*: $(\bigwedge x. P x = Q x) \Longrightarrow \{x. P x\}_c = \{x. Q x\}_c$
by *simp*

Avoid eta-contraction for robust pretty-printing.

print-translation \langle

$[Syntax-Trans.preserve-binder-abs-tr'$
 $\quad @\{const-syntax\ cset-Collect\} @\{syntax-const\ -cColl\}]$
 \rangle

lift-definition $cset-set :: 'a\ list \Rightarrow 'a\ cset$ **is** set
using $countable-finite$ **by** $blast$

lemma $countable-finite-power$:

$countable(A) \implies countable\ \{B.\ B \subseteq A \wedge finite(B)\}$
by $(metis\ Collect-conj-eq\ Int-commute\ countable-Collect-finite-subset)$

lift-definition $cINTER :: 'a\ cset \Rightarrow ('a \Rightarrow 'b\ cset) \Rightarrow 'b\ cset$ **is**
 $\lambda A\ f.\ if\ (A = \{\})\ then\ \{\}\ else\ INTER\ A\ f$
by $(auto)$

definition $cInter :: 'a\ cset\ cset \Rightarrow 'a\ cset\ (\bigcap_c\ [900]\ 900)$ **where**
 $\bigcap_c\ A = cINTER\ A\ id$

lift-definition $cfinite :: 'a\ cset \Rightarrow bool$ **is** $finite$.

lift-definition $cInfinite :: 'a\ cset \Rightarrow bool$ **is** $infinite$.

lift-definition $clist :: 'a::linorder\ cset \Rightarrow 'a\ list$ **is** $sorted-list-of-set$.

lift-definition $ccard :: 'a\ cset \Rightarrow nat$ **is** $card$.

lift-definition $cPow :: 'a\ cset \Rightarrow 'a\ cset\ cset$ **is** $\lambda A.\ \{B.\ B \subseteq_c A \wedge cfinite(B)\}$

proof –

fix A

have $\{B :: 'a\ cset.\ B \subseteq_c A \wedge cfinite\ B\} = acset\ '\{B :: 'a\ set.\ B \subseteq rcset\ A \wedge finite\ B\}$
apply $(auto\ simp\ add:\ cfinite.rep-eq\ cin-def\ less-eq-cset-def\ countable-finite)$
using $image-iff$ **apply** $fastforce$
done

moreover have $countable\ \{B :: 'a\ set.\ B \subseteq rcset\ A \wedge finite\ B\}$
by $(auto\ intro:\ countable-finite-power)$

ultimately show $countable\ \{B.\ B \subseteq_c A \wedge cfinite\ B\}$
by $simp$

qed

definition $CCollect :: ('a \Rightarrow bool\ option) \Rightarrow 'a\ cset\ option$ **where**
 $CCollect\ p = (if\ (None \notin range\ p)\ then\ Some\ (cset-Collect\ (the\ \circ\ p))\ else\ None)$

definition $cset-mapM :: 'a\ option\ cset \Rightarrow 'a\ cset\ option$ **where**
 $cset-mapM\ A = (if\ (None \in_c A)\ then\ None\ else\ Some\ (the\ ' _c\ A))$

lemma $cset-mapM-Some-image\ [simp]$:

$cset-mapM\ (cimage\ Some\ A) = Some\ A$
apply $(auto\ simp\ add:\ cset-mapM-def)$
apply $(metis\ cimage-cinsert\ cinsertI1\ option.sel\ set-cinsert)$
done

definition $CCollect-ext :: ('a \Rightarrow 'b\ option) \Rightarrow ('a \Rightarrow bool\ option) \Rightarrow 'b\ cset\ option$ **where**
 $CCollect-ext\ f\ p = do\ \{xs \leftarrow CCollect\ p;\ cset-mapM\ (f\ ' _c\ xs)\}$

lemma $the-Some-image\ [simp]$:

$the\ 'Some\ 'xs = xs$


```

by (auto simp add:image-iff)

lemma CCollect-ext-Some [simp]:
  CCollect-ext Some xs = CCollect xs
  apply (case-tac CCollect xs)
  apply (auto simp add:CCollect-ext-def)
  done

lift-definition list-of-cset :: 'a :: linorder cset  $\Rightarrow$  'a list is sorted-list-of-set .

lift-definition fset-cset :: 'a fset  $\Rightarrow$  'a cset is id
  using uncountable-infinite by auto

definition cset-count :: 'a cset  $\Rightarrow$  'a  $\Rightarrow$  nat where
  cset-count A =
    (if (finite (rcset A))
      then (SOME f::'a $\Rightarrow$ nat. inj-on f (rcset A))
      else (SOME f::'a $\Rightarrow$ nat. bij-betw f (rcset A) UNIV))

lemma cset-count-inj-seq:
  inj-on (cset-count A) (rcset A)
proof (cases finite (rcset A))
  case True note fin = this
  obtain count :: 'a  $\Rightarrow$  nat where count-inj: inj-on count (rcset A)
    by (metis countable-def mem-Collect-eq rcset)
  with fin show ?thesis
    by (metis (poly-guards-query) cset-count-def someI-ex)
next
  case False note inf = this
  obtain count :: 'a  $\Rightarrow$  nat where count-bij: bij-betw count (rcset A) UNIV
    by (metis countableE-infinite inf mem-Collect-eq rcset)
  with inf have bij-betw (cset-count A) (rcset A) UNIV
    by (metis (poly-guards-query) cset-count-def someI-ex)
  thus ?thesis
    by (metis bij-betw-imp-inj-on)
qed

lemma cset-count-infinite-bij:
  assumes infinite (rcset A)
  shows bij-betw (cset-count A) (rcset A) UNIV
proof -
  from assms obtain count :: 'a  $\Rightarrow$  nat where count-bij: bij-betw count (rcset A) UNIV
    by (metis countableE-infinite mem-Collect-eq rcset)
  with assms show ?thesis
    by (metis (poly-guards-query) cset-count-def someI-ex)
qed

definition cset-seq :: 'a cset  $\Rightarrow$  (nat  $\rightarrow$  'a) where
  cset-seq A i = (if (i  $\in$  range (cset-count A)  $\wedge$  inv-into (rcset A) (cset-count A) i  $\in_c$  A)
    then Some (inv-into (rcset A) (cset-count A) i)
    else None)

lemma cset-seq-ran: ran (cset-seq A) = rcset(A)
  apply (auto simp add: ran-def cset-seq-def cin.rep-eq)
  apply (metis cset-count-inj-seq inv-into-f-f rangeI)

```

done

lemma *cset-seq-inj*: *inj cset-seq*
proof (*rule injI*)
 fix *A B* :: 'a cset
 assume *cset-seq A = cset-seq B*
 thus *A = B*
 by (*metis cset-seq-ran rcset-inverse*)
qed

lift-definition *cset2seq* :: 'a cset \Rightarrow 'a seq
is ($\lambda A i.$ if ($i \in \text{cset-count } A$ 'rcset *A*) then *inv-into* (rcset *A*) (cset-count *A*) *i* else (*SOME* *x*. $x \in_c A$)) .

lemma *range-cset2seq*:
 $A \neq \{\}_c \implies \text{range } (\text{Rep-seq } (\text{cset2seq } A)) = \text{rcset } A$
 by (*force intro: someI2 simp add: cset2seq.rep-eq cset-count-inj-seq bot-cset.rep-eq cin.rep-eq*)

lemma *infinite-cset-count-surj*: *infinite* (rcset *A*) \implies *surj* (cset-count *A*)
 using *bij-betw-imp-surj cset-count-infinite-bij* **by** *auto*

lemma *cset2seq-inj*:
inj-on cset2seq {*A*. $A \neq \{\}_c$ }
apply (*rule inj-onI*)
apply (*simp*)
apply (*metis range-cset2seq rcset-inject*)
done

lift-definition *nat-seq2set* :: *nat seq* \Rightarrow *nat set* **is**
 $\lambda f.$ *prod-encode* ' {(*x*, *f x*) | *x*. *True*} .

lemma *inj-nat-seq2set*: *inj nat-seq2set*
proof (*rule injI, transfer*)
 fix *f g*
 assume *prod-encode* ' {(*x*, *f x*) | *x*. *True*} = *prod-encode* ' {(*x*, *g x*) | *x*. *True*}
 hence {(*x*, *f x*) | *x*. *True*} = {(*x*, *g x*) | *x*. *True*}
 by (*simp add: inj-image-eq-iff[OF inj-prod-encode]*)
 thus *f = g*
 by (*auto simp add: set-eq-iff*)
qed

lift-definition *bit-seq-of-nat-set* :: *nat set* \Rightarrow *bit seq*
is $\lambda A i.$ if ($i \in A$) then 1 else 0 .

lemma *bit-seq-of-nat-set-inj*: *inj bit-seq-of-nat-set*
apply (*rule injI*)
apply (*transfer, auto*)
apply (*metis bit.distinct(1)*)
apply (*meson zero-neq-one*)
done

lemma *bit-seq-of-nat-cset-bij*: *bij bit-seq-of-nat-set*
apply (*rule bijI*)
apply (*fact bit-seq-of-nat-set-inj*)
apply (*auto simp add: image-def*)

```

apply (transfer)
apply (rename-tac x)
apply (rule-tac x={i. x i = 1} in exI)
apply (auto)
done

```

This function is a partial injection from countable sets of natural sets to natural sets. When used with the Schroeder-Bernstein theorem, it can be used to conjure a total bijection between these two types.

definition *nat-set-cset-collapse* :: *nat set cset* \Rightarrow *nat set* **where**
nat-set-cset-collapse = *inv bit-seq-of-nat-set* \circ *seq-inj* \circ *cset2seq* \circ (λ A. (*bit-seq-of-nat-set* 'c A))

lemma *nat-set-cset-collapse-inj*: *inj-on nat-set-cset-collapse* {A. A \neq {}_c}

proof –

```

have ('c) bit-seq-of-nat-set ' {A. A  $\neq$  {}c}  $\subseteq$  {A. A  $\neq$  {}c}
by (auto simp add:cimage.rep-eq)
thus ?thesis
apply (simp add: nat-set-cset-collapse-def)
apply (rule comp-inj-on)
apply (meson bit-seq-of-nat-set-inj cset.inj-map injD inj-onI)
apply (rule comp-inj-on)
apply (metis cset2seq-inj subset-inj-on)
apply (rule comp-inj-on)
apply (rule subset-inj-on)
apply (rule seq-inj)
apply (simp)
apply (meson UNIV-I bij-imp-bij-inv bij-is-inj bit-seq-of-nat-cset-bij subsetI subset-inj-on)
done

```

qed

lemma *inj-csingle*:

```

inj csingle
by (auto intro: injI simp add: cinsert-def bot-cset.rep-eq)

```

lemma *range-csingle*:

```

range csingle  $\subseteq$  {A. A  $\neq$  {}c}
by (auto)

```

lift-definition *csets* :: 'a set \Rightarrow 'a cset set **is**

λ A. {B. B \subseteq A \wedge countable B} **by** auto

lemma *csets-finite*: *finite* A \implies *finite* (*csets* A)

by (auto simp add: csets-def)

lemma *csets-infinite*: *infinite* A \implies *infinite* (*csets* A)

by (auto simp add: csets-def, metis csets.abs-eq csets.rep-eq finite-countable-subset finite-imageI)

lemma *csets-UNIV*:

```

csets (UNIV :: 'a set) = (UNIV :: 'a cset set)
by (auto simp add: csets-def, metis image-iff rcset rcset-inverse)

```

lemma *infinite-nempty-cset*:

```

assumes infinite (UNIV :: 'a set)
shows infinite ({A. A  $\neq$  {}c} :: 'a cset set)

```

proof –

```

have infinite (UNIV :: 'a cset set)
  by (metis assms csets-UNIV csets-infinite)
hence infinite ((UNIV :: 'a cset set) - {{}_c})
  by (rule infinite-remove)
thus ?thesis
  by (auto)
qed

```

lemma *nat-set-cset-partial-bij*:

```

obtains f :: nat set cset  $\Rightarrow$  nat set where bij-betw f {A. A  $\neq$  {{}_c}} UNIV
  using Schroeder-Bernstein[OF nat-set-cset-collapse-inj, of UNIV csingle, simplified, OF inj-csingle
range-csingle]
  by (auto)

```

lemma *nat-set-cset-bij*:

```

obtains f :: nat set cset  $\Rightarrow$  nat set where bij f
proof -
  obtain g :: nat set cset  $\Rightarrow$  nat set where bij-betw g {A. A  $\neq$  {{}_c}} UNIV
    using nat-set-cset-partial-bij by blast
  moreover obtain h :: nat set cset  $\Rightarrow$  nat set cset where bij-betw h UNIV {A. A  $\neq$  {{}_c}}
  proof -
    have infinite (UNIV :: nat set cset set)
      by (metis Finite-Set.finite-set csets-UNIV csets-infinite infinite-UNIV-char-0)
    then obtain h' :: nat set cset  $\Rightarrow$  nat set cset where bij-betw h' UNIV (UNIV - {{}_c})
      using infinite-imp-bij-betw[of UNIV :: nat set cset set {{}_c}] by auto
    moreover have (UNIV :: nat set cset set) - {{}_c} = {A. A  $\neq$  {{}_c}}
      by (auto)
    ultimately show ?thesis
      using that by (auto)
  qed
  ultimately have bij (g  $\circ$  h)
    using bij-betw-trans by blast
  with that show ?thesis
    by (auto)
qed

```

definition *nat-set-cset-bij* = (SOME f :: nat set cset \Rightarrow nat set. bij f)

lemma *bij-nat-set-cset-bij*:

```

bij nat-set-cset-bij
by (metis nat-set-cset-bij nat-set-cset-bij-def someI-ex)

```

lemma *inj-on-image-csets*:

```

inj-on f A  $\implies$  inj-on (('_c) f) (csets A)
by (fastforce simp add: inj-on-def cimage-def cin-def csets-def)

```

lemma *image-csets-surj*:

```

[[ inj-on f A; f ' A = B ]]  $\implies$  ('_c) f ' csets A = csets B
apply (auto simp add: cimage-def csets-def image-mono map-fun-def)
apply (simp add: image-comp)
apply (auto simp add: image-Collect)
apply (erule subset-imageE)
apply (auto)
apply (metis countable-image rcset-inverse rcset-to-rcset subset-inj-on the-inv-into-onto)
done

```

```

lemma bij-betw-image-csets:
  bij-betw f A B  $\implies$  bij-betw (( $'_c$ ) f) (csets A) (csets B)
  by (simp add: bij-betw-def inj-on-image-csets image-csets-surj)
end

```

6 Extra Relational Definitions and Theorems

```

theory Relation-Extra
  imports HOL-Library.FuncSet
begin

```

We set up some nice syntax for heterogeneous relations at the type level

```

syntax
  -rel-type :: type  $\Rightarrow$  type  $\Rightarrow$  type (infixr  $\leftrightarrow$  0)

```

```

translations
  (type) 'a  $\leftrightarrow$  'b == (type) ('a  $\times$  'b) set

```

6.1 Relational Function Operations

```

definition rel-apply :: ('a  $\leftrightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b ( $'(-)'_r$  [999,0] 999) where
rel-apply R x = (if x  $\in$  Domain(R) then THE y. (x, y)  $\in$  R else undefined)

```

```

definition rel-domres :: 'a set  $\Rightarrow$  ('a  $\leftrightarrow$  'b)  $\Rightarrow$  'a  $\leftrightarrow$  'b (infixr  $\triangleleft_r$  85) where
rel-domres A R = {(k, v)  $\in$  R. k  $\in$  A}

```

```

definition rel-override :: ('a  $\leftrightarrow$  'b)  $\Rightarrow$  ('a  $\leftrightarrow$  'b)  $\Rightarrow$  'a  $\leftrightarrow$  'b (infixl  $+_r$  65) where
rel-override R S = ( $-$  Domain S)  $\triangleleft_r$  R  $\cup$  S

```

```

definition rel-update :: ('a  $\leftrightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  'a  $\leftrightarrow$  'b where
rel-update R k v = rel-override R {(k, v)}

```

6.2 Domain Restriction

```

lemma Domain-rel-domres [simp]: Domain (A  $\triangleleft_r$  R) = A  $\cap$  Domain(R)
  by (auto simp add: rel-domres-def)

```

```

lemma rel-domres-empty [simp]: {}  $\triangleleft_r$  R = {}
  by (simp add: rel-domres-def)

```

```

lemma rel-domres-UNIV [simp]: UNIV  $\triangleleft_r$  R = R
  by (simp add: rel-domres-def)

```

```

lemma rel-domres-nil [simp]: A  $\triangleleft_r$  {} = {}
  by (simp add: rel-domres-def)

```

```

lemma rel-domres-inter [simp]: A  $\triangleleft_r$  B  $\triangleleft_r$  R = (A  $\cap$  B)  $\triangleleft_r$  R
  by (auto simp add: rel-domres-def)

```

6.3 Relational Override

```

interpretation rel-override-monoid: monoid-add ( $+_r$ ) {}
  by (unfold-locales, simp-all add: rel-override-def, auto simp add: rel-domres-def)

```

lemma *Domain-rel-override* [simp]: $\text{Domain } (R +_r S) = \text{Domain}(R) \cup \text{Domain}(S)$
by (auto simp add: rel-override-def Domain-Un-eq)

lemma *Range-rel-override*: $\text{Range}(R +_r S) \subseteq \text{Range}(R) \cup \text{Range}(S)$
by (auto simp add: rel-override-def rel-domres-def)

6.4 Functional Relations

definition *functional* :: $('a \leftrightarrow 'b) \Rightarrow \text{bool}$ **where**
functional $g = \text{inj-on fst } g$

lemma *functional-algebraic*: $\text{functional } R \longleftrightarrow R^{-1} \circ R \subseteq \text{Id}$
apply (auto simp add: functional-def subset-iff relcomp-unfold)
using inj-on-eq-iff **apply** fastforce
apply (metis inj-onI surjective-pairing)
done

lemma *functional-determine*: $\llbracket \text{functional } R; (x, y) \in R; (x, z) \in R \rrbracket \Longrightarrow y = z$
by (auto simp add: functional-algebraic subset-iff relcomp-unfold)

lemma *functional-apply*:
assumes *functional* R $(x, y) \in R$
shows $R(x)_r = y$
by (metis (no-types, lifting) Domain.intros DomainE assms(1) assms(2) functional-determine rel-apply-def theI-unique)

lemma *functional-elem*:
assumes *functional* R $x \in \text{Domain}(R)$
shows $(x, R(x)_r) \in R$
using assms(1) assms(2) *functional-apply* **by** fastforce

lemma *functional-empty* [simp]: *functional* $\{\}$
by (simp add: functional-def)

lemma *functional-override* [intro]: $\llbracket \text{functional } R; \text{functional } S \rrbracket \Longrightarrow \text{functional } (R +_r S)$
by (auto simp add: functional-algebraic rel-override-def rel-domres-def)

definition *functional-list* :: $('a \times 'b) \text{ list} \Rightarrow \text{bool}$ **where**
functional-list $xs = (\forall x y z. \text{ListMem } (x, y) xs \wedge \text{ListMem } (x, z) xs \longrightarrow y = z)$

lemma *functional-insert* [simp]: $\text{functional } (\text{insert } (x, y) g) \longleftrightarrow (g \text{ `` } \{x\} \subseteq \{y\} \wedge \text{functional } g)$
by (auto simp add: functional-def inj-on-def image-def)

lemma *functional-list-nil* [simp]: *functional-list* $[]$
by (simp add: functional-list-def ListMem-iff)

lemma *functional-list*: $\text{functional-list } xs \longleftrightarrow \text{functional } (\text{set } xs)$
apply (induct xs)
apply (simp add: functional-def)
apply (simp add: functional-def functional-list-def ListMem-iff)
apply (safe)
apply (force)
apply (force)
apply (force)
apply (force)
apply (force)

```

    apply (force)
  apply (force)
  apply (force)
done

```

definition *fun-rel* :: ('a \Rightarrow 'b) \Rightarrow ('a \leftrightarrow 'b) **where**
fun-rel f = {(x, y). y = f x}

lemma *functional-fun-rel*: *functional* (fun-rel f)
by (simp add: fun-rel-def functional-def)
 (metis (mono-tags, lifting) Product-Type.Collect-case-prodD inj-onI prod.expand)

lemma *rel-apply-fun* [simp]: (fun-rel f)(x)_r = f x
by (simp add: fun-rel-def rel-apply-def)

6.5 Left-Total Relations

definition *left-totalr-on* :: 'a set \Rightarrow ('a \leftrightarrow 'b) \Rightarrow bool **where**
left-totalr-on A R \longleftrightarrow ($\forall x \in A. \exists y. (x, y) \in R$)

abbreviation *left-totalr* R \equiv *left-totalr-on* UNIV R

lemma *left-totalr-algebraic*: *left-totalr* R \longleftrightarrow Id \subseteq R \circ R⁻¹
by (auto simp add: left-totalr-on-def)

lemma *left-totalr-fun-rel*: *left-totalr* (fun-rel f)
by (simp add: left-totalr-on-def fun-rel-def)

6.6 Relation Sets

definition *rel-typed* :: 'a set \Rightarrow 'b set \Rightarrow ('a \leftrightarrow 'b) set (**infixr** \leftrightarrow_r 55) **where**
rel-typed A B = {R. Domain(R) \subseteq A \wedge Range(R) \subseteq B}

lemma *rel-typed-intro*: $\llbracket \text{Domain}(R) \subseteq A; \text{Range}(R) \subseteq B \rrbracket \Longrightarrow R \in A \leftrightarrow_r B$
by (simp add: rel-typed-def)

definition *rel-pfun* :: 'a set \Rightarrow 'b set \Rightarrow ('a \leftrightarrow 'b) set (**infixr** \rightarrow_r 55) **where**
rel-pfun A B = {R. R \in A \leftrightarrow_r B \wedge functional R}

lemma *rel-pfun-intro*: $\llbracket R \in A \leftrightarrow_r B; \text{functional } R \rrbracket \Longrightarrow R \in A \rightarrow_r B$
by (simp add: rel-pfun-def)

definition *rel-tfun* :: 'a set \Rightarrow 'b set \Rightarrow ('a \leftrightarrow 'b) set (**infixr** \rightarrow_r 55) **where**
rel-tfun A B = {R. R \in A \rightarrow_r B \wedge left-totalr R}

definition *rel-ffun* :: 'a set \Rightarrow 'b set \Rightarrow ('a \leftrightarrow 'b) set **where**
rel-ffun A B = {R. R \in A \rightarrow_r B \wedge finite(Domain R)}

6.7 Closure Properties

These can be seen as typing rules for relational functions

named-theorems *rclos*

lemma *rel-ffun-is-pfun* [rclos]: R \in rel-ffun A B \Longrightarrow R \in A \rightarrow_r B
by (simp add: rel-ffun-def)

lemma *rel-tfun-is-pfun* [rclos]: $R \in A \rightarrow_r B \implies R \in A \rightarrow_r B$
by (*simp add: rel-tfun-def*)

lemma *rel-pfun-is-typed* [rclos]: $R \in A \rightarrow_r B \implies R \in A \leftrightarrow_r B$
by (*simp add: rel-pfun-def*)

lemma *rel-ffun-empty* [rclos]: $\{\} \in \text{rel-ffun } A \ B$
by (*simp add: rel-ffun-def rel-pfun-def rel-typed-def*)

lemma *rel-pfun-apply* [rclos]: $\llbracket x \in \text{Domain}(R); R \in A \rightarrow_r B \rrbracket \implies R(x)_r \in B$
using *functional-apply* **by** (*fastforce simp add: rel-pfun-def rel-typed-def*)

lemma *rel-tfun-apply* [rclos]: $\llbracket x \in A; R \in A \rightarrow_r B \rrbracket \implies R(x)_r \in B$
by (*metis (no-types, lifting) Domain-iff iso-tuple-UNIV-I left-totalr-on-def mem-Collect-eq rel-pfun-apply rel-tfun-def*)

lemma *rel-typed-insert* [rclos]: $\llbracket R \in A \leftrightarrow_r B; x \in A; y \in B \rrbracket \implies \text{insert } (x, y) \ R \in A \leftrightarrow_r B$
by (*simp add: rel-typed-def*)

lemma *rel-pfun-insert* [rclos]: $\llbracket R \in A \rightarrow_r B; x \in A; y \in B; x \notin \text{Domain}(R) \rrbracket \implies \text{insert } (x, y) \ R \in A \rightarrow_r B$
by (*auto intro: rclos simp add: rel-pfun-def*)

lemma *rel-pfun-override* [rclos]: $\llbracket R \in A \rightarrow_r B; S \in A \rightarrow_r B \rrbracket \implies (R +_r S) \in A \rightarrow_r B$
apply (*rule rel-pfun-intro*)
apply (*rule rel-typed-intro*)
apply (*auto simp add: rel-pfun-def rel-typed-def*)
apply (*metis (no-types, hide-lams) Range.simps Range-Un-eq Range-rel-override Un-iff rev-subsetD*)
done

end

7 Map Type: extra functions and properties

theory *Map-Extra*
imports
Relation-Extra
HOL-Library.Countable-Set
HOL-Library.Monad-Syntax
begin

7.1 Graphing Maps

definition *map-graph* :: $('a \rightarrow 'b) \Rightarrow ('a \leftrightarrow 'b)$ **where**
map-graph $f = \{(x, y) \mid x \ y. \ f \ x = \text{Some } y\}$

definition *graph-map* :: $('a \leftrightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **where**
graph-map $g = (\lambda x. \text{if } (x \in \text{fst } 'g) \text{ then } \text{Some } (\text{SOME } y. (x, y) \in g) \text{ else } \text{None})$

definition *graph-map'* :: $('a \leftrightarrow 'b) \rightarrow ('a \rightarrow 'b)$ **where**
graph-map' $R = (\text{if } (\text{functional } R) \text{ then } \text{Some } (\text{graph-map } R) \text{ else } \text{None})$

lemma *map-graph-mem-equiv*: $(x, y) \in \text{map-graph } f \longleftrightarrow f(x) = \text{Some } y$
by (*simp add: map-graph-def*)

lemma *map-graph-functional* [simp]: *functional (map-graph f)*
 by (simp add: functional-def map-graph-def inj-on-def)

lemma *map-graph-countable* [simp]: *countable (dom f) \implies countable (map-graph f)*
 apply (auto simp add: map-graph-def countable-def)
 apply (rename-tac f')
 apply (rule-tac x=f' \circ fst in exI)
 apply (auto simp add: inj-on-def dom-def)
 apply fastforce
 done

lemma *map-graph-inv* [simp]: *graph-map (map-graph f) = f*
 by (auto intro!: ext simp add: map-graph-def graph-map-def image-def)

lemma *graph-map-empty* [simp]: *graph-map {} = Map.empty*
 by (simp add: graph-map-def)

lemma *graph-map-insert* [simp]: $\llbracket \text{functional } g; g' \llbracket \{x\} \subseteq \{y\} \rrbracket \implies \text{graph-map (insert (x,y) g)} = (\text{graph-map } g)(x \mapsto y)$
 by (rule ext, auto simp add: graph-map-def)

lemma *dom-map-graph*: *dom f = Domain(map-graph f)*
 by (simp add: map-graph-def dom-def image-def)

lemma *ran-map-graph*: *ran f = Range(map-graph f)*
 by (simp add: map-graph-def ran-def image-def)

lemma *rel-apply-map-graph*:
 $x \in \text{dom}(f) \implies (\text{map-graph } f)(x)_r = \text{the } (f \ x)$
 by (auto simp add: rel-apply-def map-graph-def)

lemma *ran-map-add-subset*:
 $\text{ran } (x ++ y) \subseteq (\text{ran } x) \cup (\text{ran } y)$
 by (auto simp add: ran-def)

lemma *finite-dom-graph*: *finite (dom f) \implies finite (map-graph f)*
 by (metis dom-map-graph finite-imageD fst-eq-Domain functional-def map-graph-functional)

lemma *finite-dom-ran* [simp]: *finite (dom f) \implies finite (ran f)*
 by (metis finite-Range finite-dom-graph ran-map-graph)

lemma *graph-map-inv* [simp]: *functional g \implies map-graph (graph-map g) = g*
 apply (auto simp add: map-graph-def graph-map-def functional-def)
 apply (metis (lifting, no-types) image-iff option.distinct(1) option.inject someI surjective-pairing)
 apply (simp add: inj-on-def)
 apply (metis fst-conv snd-conv some-equality)
 apply (metis (lifting) fst-conv image-iff)
 done

lemma *graph-map-dom*: *dom (graph-map R) = fst ' R*
 by (simp add: graph-map-def dom-def)

lemma *graph-map-countable-dom*: *countable R \implies countable (dom (graph-map R))*
 by (simp add: graph-map-dom)

```

lemma countable-ran:
  assumes countable (dom f)
  shows countable (ran f)
proof -
  have countable (map-graph f)
    by (simp add: assms)
  then have countable (Range(map-graph f))
    by (simp add: Range-snd)
  thus ?thesis
    by (simp add: ran-map-graph)
qed

```

```

lemma map-graph-inv' [simp]:
  graph-map' (map-graph f) = Some f
  by (simp add: graph-map'-def)

```

```

lemma map-graph-inj:
  inj map-graph
  by (metis injI map-graph-inv)

```

```

lemma map-eq-graph: f = g  $\longleftrightarrow$  map-graph f = map-graph g
  by (auto simp add: inj-eq map-graph-inj)

```

```

lemma map-le-graph: f  $\subseteq_m$  g  $\longleftrightarrow$  map-graph f  $\subseteq$  map-graph g
  by (force simp add: map-le-def map-graph-def)

```

```

lemma map-graph-comp: map-graph (g  $\circ_m$  f) = (map-graph f) O (map-graph g)
  apply (auto simp add: map-comp-def map-graph-def relcomp-unfold)
  apply (rename-tac a b)
  apply (case-tac f a, auto)
  done

```

7.2 Map Application

```

definition map-apply :: ('a  $\rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b (-'(-)'m [999,0] 999) where
  map-apply = ( $\lambda$  f x. the (f x))

```

7.3 Map Membership

```

fun map-member :: 'a  $\times$  'b  $\Rightarrow$  ('a  $\rightarrow$  'b)  $\Rightarrow$  bool (infix  $\in_m$  50) where
  (k, v)  $\in_m$  m  $\longleftrightarrow$  m(k) = Some(v)

```

```

lemma map-ext:
   $\llbracket \bigwedge x y. (x, y) \in_m A \longleftrightarrow (x, y) \in_m B \rrbracket \implies A = B$ 
  by (rule ext, auto, metis not-Some-eq)

```

```

lemma map-member-alt-def:
  (x, y)  $\in_m$  A  $\longleftrightarrow$  (x  $\in$  dom A  $\wedge$  A(x)m = y)
  by (auto simp add: map-apply-def)

```

```

lemma map-le-member:
  f  $\subseteq_m$  g  $\longleftrightarrow$  ( $\forall x y. (x, y) \in_m f \longrightarrow (x, y) \in_m g$ )
  by (force simp add: map-le-def)

```

7.4 Preimage

definition *preimage* :: ('a \rightarrow 'b) \Rightarrow 'b set \Rightarrow 'a set **where**
preimage f B = {x \in dom(f). the(f(x)) \in B}

lemma *preimage-range*: *preimage* f (ran f) = dom f
by (auto simp add: *preimage-def* *ran-def*)

lemma *dom-preimage*: dom (m \circ_m f) = *preimage* f (dom m)
apply (auto simp add: *dom-def* *preimage-def*)
apply (meson map-comp-Some-iff)
apply (metis map-comp-def option.case-eq-if option.distinct(1))
done

lemma *countable-preimage*:
 $\llbracket \text{countable } A; \text{inj-on } f \text{ (preimage } f \text{ } A) \rrbracket \implies \text{countable (preimage } f \text{ } A)$
apply (auto simp add: *countable-def*)
apply (rename-tac g)
apply (rule-tac x=g \circ the \circ f **in** exI)
apply (rule inj-onI)
apply (drule inj-onD)
apply (auto simp add: *preimage-def* *inj-onD*)
done

7.5 Minus operation for maps

definition *map-minus* :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) (**infixl** -- 100)
where *map-minus* f g = (λx . if (f x = g x) then None else f x)

lemma *map-minus-apply* [*simp*]: $y \in \text{dom}(f -- g) \implies (f -- g)(y)_m = f(y)_m$
by (auto simp add: *map-minus-def* *dom-def* *map-apply-def*)

lemma *map-member-plus*:
 $(x, y) \in_m f ++ g \iff ((x \notin \text{dom}(g) \wedge (x, y) \in_m f) \vee (x, y) \in_m g)$
by (auto simp add: *map-add-Some-iff*)

lemma *map-member-minus*:
 $(x, y) \in_m f -- g \iff (x, y) \in_m f \wedge (\neg (x, y) \in_m g)$
by (auto simp add: *map-minus-def*)

lemma *map-minus-plus-commute*:
 $\text{dom}(g) \cap \text{dom}(h) = \{\} \implies (f -- g) ++ h = (f ++ h) -- g$
apply (rule map-ext)
apply (auto simp add: *map-member-plus* *map-member-minus* *simp* del: *map-member.simps*)
apply (auto simp add: *map-member-alt-def*)
done

lemma *map-graph-minus*: *map-graph* (f -- g) = *map-graph* f - *map-graph* g
by (auto simp add: *map-minus-def* *map-graph-def*, (meson option.distinct(1))+)

lemma *map-minus-common-subset*:
 $\llbracket h \subseteq_m f; h \subseteq_m g \rrbracket \implies (f -- h = g -- h) = (f = g)$
by (auto simp add: *map-eq-graph* *map-graph-minus* *map-le-graph*)

7.6 Map Bind

Create some extra intro/elim rules to help dealing with proof about option bind.

lemma *option-bindSomeE* [elim!]:

$\llbracket X >>= F = \text{Some}(v); \bigwedge x. \llbracket X = \text{Some}(x); F(x) = \text{Some}(v) \rrbracket \implies P \rrbracket \implies P$
 by (case-tac X, auto)

lemma *option-bindSomeI* [intro]:

$\llbracket X = \text{Some}(x); F(x) = \text{Some}(y) \rrbracket \implies X >>= F = \text{Some}(y)$
 by (simp)

lemma *ifSomeE* [elim]: $\llbracket (\text{if } c \text{ then } \text{Some}(x) \text{ else } \text{None}) = \text{Some}(y); \llbracket c; x = y \rrbracket \implies P \rrbracket \implies P$

by (case-tac c, auto)

7.7 Range Restriction

A range restriction operator; only domain restriction is provided in HOL.

definition *ran-restrict-map* :: $('a \rightarrow 'b) \Rightarrow 'b \text{ set} \Rightarrow 'a \rightarrow 'b$ (-|- [111,110] 110) **where**
ran-restrict-map f B = $(\lambda x. \text{do } \{ v \leftarrow f(x); \text{if } (v \in B) \text{ then } \text{Some}(v) \text{ else } \text{None} \})$

lemma *ran-restrict-empty* [simp]: $f \upharpoonright_{\{\}} = \text{Map.empty}$

by (simp add: ran-restrict-map-def)

lemma *ran-restrict-ran* [simp]: $f \upharpoonright_{\text{ran}(f)} = f$

apply (auto simp add: ran-restrict-map-def ran-def)

apply (rule ext)

apply (case-tac f(x), auto)

done

lemma *ran-ran-restrict* [simp]: $\text{ran}(f \upharpoonright_B) = \text{ran}(f) \cap B$

by (auto intro!: option-bindSomeI simp add: ran-restrict-map-def ran-def)

lemma *dom-ran-restrict*: $\text{dom}(f \upharpoonright_B) \subseteq \text{dom}(f)$

by (auto simp add: ran-restrict-map-def dom-def)

lemma *ran-restrict-finite-dom* [intro]:

$\text{finite}(\text{dom}(f)) \implies \text{finite}(\text{dom}(f \upharpoonright_B))$

by (metis finite-subset dom-ran-restrict)

lemma *dom-Some* [simp]: $\text{dom} (\text{Some} \circ f) = \text{UNIV}$

by (auto)

lemma *dom-left-map-add* [simp]: $x \in \text{dom } g \implies (f ++ g) x = g x$

by (auto simp add: map-add-def dom-def)

lemma *dom-right-map-add* [simp]: $\llbracket x \notin \text{dom } g; x \in \text{dom } f \rrbracket \implies (f ++ g) x = f x$

by (auto simp add: map-add-def dom-def)

lemma *map-add-restrict*:

$f ++ g = (f \upharpoonright' (- \text{dom } g)) ++ g$

by (rule ext, auto simp add: map-add-def restrict-map-def)

7.8 Map Inverse and Identity

definition *map-inv* :: $('a \rightarrow 'b) \Rightarrow ('b \rightarrow 'a)$ **where**

$\text{map-inv } f \equiv \lambda y. \text{ if } (y \in \text{ran } f) \text{ then Some } (\text{SOME } x. f x = \text{Some } y) \text{ else None}$

definition $\text{map-id-on} :: 'a \text{ set} \Rightarrow ('a \rightarrow 'a) \text{ where}$
 $\text{map-id-on } xs \equiv \lambda x. \text{ if } (x \in xs) \text{ then Some } x \text{ else None}$

lemma $\text{map-id-on-in } [\text{simp}]$:
 $x \in xs \implies \text{map-id-on } xs \ x = \text{Some } x$
by ($\text{simp add:map-id-on-def}$)

lemma $\text{map-id-on-out } [\text{simp}]$:
 $x \notin xs \implies \text{map-id-on } xs \ x = \text{None}$
by ($\text{simp add:map-id-on-def}$)

lemma $\text{map-id-dom } [\text{simp}]$: $\text{dom } (\text{map-id-on } xs) = xs$
by ($\text{simp add:dom-def map-id-on-def}$)

lemma $\text{map-id-ran } [\text{simp}]$: $\text{ran } (\text{map-id-on } xs) = xs$
by ($\text{force simp add:ran-def map-id-on-def}$)

lemma $\text{map-id-on-UNIV} [\text{simp}]$: $\text{map-id-on UNIV} = \text{Some}$
by ($\text{simp add:map-id-on-def}$)

lemma $\text{map-id-on-inj } [\text{simp}]$:
 $\text{inj-on } (\text{map-id-on } xs) \ xs$
by ($\text{simp add:inj-on-def}$)

lemma $\text{map-inv-empty } [\text{simp}]$: $\text{map-inv Map.empty} = \text{Map.empty}$
by ($\text{simp add:map-inv-def}$)

lemma $\text{map-inv-id } [\text{simp}]$:
 $\text{map-inv } (\text{map-id-on } xs) = \text{map-id-on } xs$
by ($\text{force simp add:map-inv-def map-id-on-def ran-def}$)

lemma $\text{map-inv-Some } [\text{simp}]$: $\text{map-inv Some} = \text{Some}$
by ($\text{simp add:map-inv-def ran-def}$)

lemma $\text{map-inv-f-f } [\text{simp}]$:
 $\llbracket \text{inj-on } f \ (\text{dom } f); f x = \text{Some } y \rrbracket \implies \text{map-inv } f \ y = \text{Some } x$
apply ($\text{auto simp add:map-inv-def}$)
apply ($\text{rule some-equality}$)
apply ($\text{auto simp add:inj-on-def dom-def ran-def}$)
done

lemma $\text{dom-map-inv } [\text{simp}]$:
 $\text{dom } (\text{map-inv } f) = \text{ran } f$
by ($\text{auto simp add:map-inv-def}$)

lemma $\text{ran-map-inv } [\text{simp}]$:
 $\text{inj-on } f \ (\text{dom } f) \implies \text{ran } (\text{map-inv } f) = \text{dom } f$
apply ($\text{auto simp add:map-inv-def ran-def}$)
apply (rename-tac a b)
apply ($\text{rule-tac } x=a \text{ in } exI$)
apply (force intro:someI)
apply (rename-tac x y)
apply ($\text{rule-tac } x=y \text{ in } exI$)

```

apply (auto)
apply (rule some-equality, simp-all)
apply (auto simp add:inj-on-def dom-def)
done

lemma dom-image-ran:  $f \text{ ' dom } f = \text{Some ' ran } f$ 
  by (auto simp add:dom-def ran-def image-def)

lemma inj-map-inv [intro]:
   $\text{inj-on } f \text{ (dom } f) \implies \text{inj-on (map-inv } f) \text{ (ran } f)$ 
  apply (auto simp add:map-inv-def inj-on-def dom-def ran-def)
  apply (rename-tac x y u v)
  apply (frule-tac  $P = \lambda xa. f \text{ xa} = \text{Some } x$  in some-equality)
  apply (auto)
  apply (metis (mono-tags) option.sel someI)
done

lemma inj-map-bij:  $\text{inj-on } f \text{ (dom } f) \implies \text{bij-betw } f \text{ (dom } f) \text{ (Some ' ran } f)$ 
  by (auto simp add:inj-on-def dom-def ran-def image-def bij-betw-def)

lemma map-inv-map-inv [simp]:
  assumes  $\text{inj-on } f \text{ (dom } f)$ 
  shows  $\text{map-inv (map-inv } f) = f$ 
proof –

  from assms have  $\text{inj-on (map-inv } f) \text{ (ran } f)$ 
  by auto

  thus ?thesis
  apply (rule-tac ext)
  apply (rename-tac x)
  apply (case-tac  $\exists y. \text{map-inv } f \text{ y} = \text{Some } x$ )
  apply (auto)[1]
  apply (simp add:map-inv-def)
  apply (rename-tac x y)
  apply (case-tac  $y \in \text{ran } f$ , simp-all)
  apply (auto)
  apply (rule someI2-ex)
  apply (simp add:ran-def)
  apply (simp)
  apply (metis assms dom-image-ran dom-map-inv image-iff map-add-dom-app-simps(2) map-add-dom-app-simps(3)
    ran-map-inv)
  done
qed

lemma map-self-adjoin-complete [intro]:
  assumes  $\text{dom } f \cap \text{ran } f = \{\}$   $\text{inj-on } f \text{ (dom } f)$ 
  shows  $\text{inj-on (map-inv } f \text{ ++ } f) \text{ (dom } f \cup \text{ran } f)$ 
  apply (rule inj-onI)
  apply (insert assms)
  apply (rename-tac x y)
  apply (case-tac  $x \in \text{dom } f$ )
  apply (simp)
  apply (case-tac  $y \in \text{dom } f$ )
  apply (simp add:inj-on-def)

```

```

apply (case-tac  $y \in \text{ran } f$ )
apply (subgoal-tac  $y \in \text{dom } (\text{map-inv } f)$ )
apply (simp)
apply (metis Int-iff domD empty-iff ranI ran-map-inv)
apply (simp)
apply (simp)
apply (simp)
apply (case-tac  $y \in \text{dom } f$ )
apply (simp)
apply (case-tac  $y \in \text{ran } f$ )
apply (subgoal-tac  $y \in \text{dom } (\text{map-inv } f)$ )
apply (simp)
apply (metis Int-iff empty-iff)
apply (simp)
apply (metis Int-iff domD empty-iff ranI ran-map-inv)
apply (simp)
apply (metis (lifting) inj-map-inv inj-on-contrad)
done

```

```

lemma inj-completed-map [intro]:
   $\llbracket \text{dom } f = \text{ran } f; \text{inj-on } f (\text{dom } f) \rrbracket \implies \text{inj } (\text{Some } ++ f)$ 
apply (drule inj-map-bij)
apply (auto simp add:bij-betw-def)
apply (auto simp add:inj-on-def)[1]
apply (rename-tac  $x \ y$ )
apply (case-tac  $x \in \text{dom } f$ )
apply (simp)
apply (case-tac  $y \in \text{dom } f$ )
apply (simp)
apply (simp add:ran-def)
apply (case-tac  $y \in \text{dom } f$ )
apply (auto intro:ranI)
done

```

```

lemma bij-completed-map [intro]:
   $\llbracket \text{dom } f = \text{ran } f; \text{inj-on } f (\text{dom } f) \rrbracket \implies$ 
   $\text{bij-betw } (\text{Some } ++ f) \text{ UNIV } (\text{range } \text{Some})$ 
apply (auto simp add:bij-betw-def)
apply (rename-tac  $x$ )
apply (case-tac  $x \in \text{dom } f$ )
apply (simp)
apply (metis domD rangeI)
apply (simp)
apply (simp add:image-def)
apply (metis (full-types) dom-image-ran dom-left-map-add image-iff map-add-dom-app-simps(3))
done

```

```

lemma bij-map-Some:
   $\text{bij-betw } f \ a \ (\text{Some } 'b) \implies \text{bij-betw } (f \circ \text{the}) \ a \ b$ 
apply (simp add:bij-betw-def)
apply (safe)
apply (metis (hide-lams, no-types) comp-inj-on-iff f-the-inv-into-f inj-on-inverseI option.sel)
apply (metis (hide-lams, no-types) comp-apply image-iff option.sel)
apply (metis imageI image-comp option.sel)
done

```

```

lemma ran-map-add [simp]:
   $m'(dom\ m \cap dom\ n) = n'(dom\ m \cap dom\ n) \implies$ 
   $ran(m ++ n) = ran\ n \cup ran\ m$ 
  apply (auto simp add:ran-def)
  apply (metis map-add-find-right)
  apply (rename-tac x a)
  apply (case-tac a ∈ dom n)
  apply (subgoal-tac ∃ b. n b = Some x)
  apply (auto)
  apply (rename-tac x a b y)
  apply (rule-tac x=b in exI)
  apply (simp)
  apply (metis (hide-lams, no-types) IntI domI image-iff)
apply (metis (full-types) map-add-None map-add-dom-app-simps(1) map-add-dom-app-simps(3) not-None-eq)
done

```

```

lemma ran-maplets [simp]:
   $\llbracket length\ xs = length\ ys; distinct\ xs \rrbracket \implies ran\ [xs \mapsto] ys = set\ ys$ 
  by (induct rule:list-induct2, simp-all)

```

```

lemma inj-map-add:
   $\llbracket inj-on\ f\ (dom\ f); inj-on\ g\ (dom\ g); ran\ f \cap ran\ g = \{\} \rrbracket \implies$ 
   $inj-on\ (f ++ g)\ (dom\ f \cup dom\ g)$ 
  apply (auto simp add:inj-on-def)
  apply (metis (full-types) disjoint-iff-not-equal domI dom-left-map-add map-add-dom-app-simps(3) ranI)
  apply (metis domI)
  apply (metis disjoint-iff-not-equal ranI)
  apply (metis disjoint-iff-not-equal domIff map-add-Some-iff ranI)
  apply (metis domI)
done

```

```

lemma map-inv-add [simp]:
  assumes  $inj-on\ f\ (dom\ f)\ inj-on\ g\ (dom\ g)$ 
   $dom\ f \cap dom\ g = \{\} \quad ran\ f \cap ran\ g = \{\}$ 
  shows  $map-inv\ (f ++ g) = map-inv\ f ++ map-inv\ g$ 
proof (rule ext)

```

```

  from assms have  $minj: inj-on\ (f ++ g)\ (dom\ (f ++ g))$ 
  by (simp, metis inj-map-add sup-commute)

```

```

  fix  $x$ 
  have  $x \in ran\ g \implies map-inv\ (f ++ g)\ x = (map-inv\ f ++ map-inv\ g)\ x$ 
proof –
  assume  $ran:x \in ran\ g$ 
  then obtain  $y$  where  $dom:g\ y = Some\ x\ y \in dom\ g$ 
  by (auto simp add:ran-def)

```

```

  hence  $(f ++ g)\ y = Some\ x$ 
  by simp

```

```

  with assms minj ran dom show  $map-inv\ (f ++ g)\ x = (map-inv\ f ++ map-inv\ g)\ x$ 
  by simp

```

```

qed

```



```

moreover have  $\llbracket x \notin \text{ran } g; x \in \text{ran } f \rrbracket \implies \text{map-inv } (f ++ g) x = (\text{map-inv } f ++ \text{map-inv } g) x$ 
proof –
  assume  $\text{ran}:x \notin \text{ran } g \ x \in \text{ran } f$ 
  with assms obtain y where  $\text{dom}:f y = \text{Some } x \ y \in \text{dom } f \ y \notin \text{dom } g$ 
    by (auto simp add:ran-def)

  with ran have  $(f ++ g) y = \text{Some } x$ 
    by (simp)

  with assms minj ran dom show  $\text{map-inv } (f ++ g) x = (\text{map-inv } f ++ \text{map-inv } g) x$ 
    by simp
qed

moreover from assms minj have  $\llbracket x \notin \text{ran } g; x \notin \text{ran } f \rrbracket \implies \text{map-inv } (f ++ g) x = (\text{map-inv } f$ 
 $++ \text{map-inv } g) x$ 
  apply (auto simp add:map-inv-def ran-def map-add-def)
  apply (metis dom-left-map-add map-add-def map-add-dom-app-simps(3))
  done

ultimately show  $\text{map-inv } (f ++ g) x = (\text{map-inv } f ++ \text{map-inv } g) x$ 
  apply (case-tac x \in \text{ran } g)
  apply (simp)
  apply (case-tac x \in \text{ran } f)
  apply (simp-all)
  done
qed

lemma map-add-lookup [simp]:
   $x \notin \text{dom } f \implies ([x \mapsto y] ++ f) x = \text{Some } y$ 
  by (simp add:map-add-def dom-def)

lemma map-add-Some:  $\text{Some } ++ f = \text{map-id-on } (- \text{dom } f) ++ f$ 
  apply (rule ext)
  apply (rename-tac x)
  apply (case-tac x \in \text{dom } f)
  apply (simp-all)
  done

lemma distinct-map-dom:
   $x \notin \text{set } xs \implies x \notin \text{dom } [xs \mapsto] ys$ 
  by (simp add:dom-def)

lemma distinct-map-ran:
   $\llbracket \text{distinct } xs; y \notin \text{set } ys; \text{length } xs = \text{length } ys \rrbracket \implies$ 
 $y \notin \text{ran } ([xs \mapsto] ys)$ 
  apply (simp add:map-upds-def)
  apply (subgoal-tac distinct (map fst (rev (zip xs ys))))
  apply (simp add:ran-distinct)
  apply (metis (hide-lams, no-types) image-iff set-zip-rightD surjective-pairing)
  apply (simp add:zip-rev[THEN sym])
  done

lemma maplets-lookup[rule-format,dest]:
   $\llbracket \text{length } xs = \text{length } ys; \text{distinct } xs \rrbracket \implies$ 

```

$\forall y. [xs \mapsto] ys] x = \text{Some } y \longrightarrow y \in \text{set } ys$
by (induct rule:list-induct2, auto)

lemma maplets-distinct-inj [intro]:
 $\llbracket \text{length } xs = \text{length } ys; \text{distinct } xs; \text{distinct } ys; \text{set } xs \cap \text{set } ys = \{\} \rrbracket \Longrightarrow$
 $\text{inj-on } [xs \mapsto] ys] (\text{set } xs)$
apply (induct rule:list-induct2)
apply (simp-all)
apply (rule conjI)
apply (rule inj-onI)
apply (rename-tac x xs y ys xa ya)
apply (case-tac xa = x)
apply (simp)
apply (case-tac xa = y)
apply (simp)
apply (simp)
apply (case-tac ya = x)
apply (simp)
apply (simp add:inj-on-def)
apply (auto)
apply (rename-tac x xs y ys xa)
apply (case-tac xa = y)
apply (simp)
apply (metis maplets-lookup)
done

lemma map-inv-maplet[simp]: $\text{map-inv } [x \mapsto y] = [y \mapsto x]$
by (auto simp add:map-inv-def)

lemma map-inv-maplets [simp]:
 $\llbracket \text{length } xs = \text{length } ys; \text{distinct } xs; \text{distinct } ys; \text{set } xs \cap \text{set } ys = \{\} \rrbracket \Longrightarrow$
 $\text{map-inv } [xs \mapsto] ys] = [ys \mapsto] xs]$
apply (induct rule:list-induct2)
apply (simp-all)
apply (rename-tac x xs y ys)
apply (subgoal-tac map-inv ([xs \mapsto] ys] ++ [x \mapsto y]) = map-inv [xs \mapsto] ys] ++ map-inv [x \mapsto y])
apply (simp)
apply (rule map-inv-add)
apply (auto)
done

lemma maplets-lookup-nth [rule-format,simp]:
 $\llbracket \text{length } xs = \text{length } ys; \text{distinct } xs \rrbracket \Longrightarrow$
 $\forall i < \text{length } ys. [xs \mapsto] ys] (xs ! i) = \text{Some } (ys ! i)$
apply (induct rule:list-induct2)
apply (auto)
apply (rename-tac x xs y ys i)
apply (case-tac i)
apply (simp-all)
apply (metis nth-mem)
apply (rename-tac x xs y ys i)
apply (case-tac i)
apply (auto)
done

```

theorem inv-map-inv:
   $\llbracket \text{inj-on } f \text{ (dom } f); \text{ran } f = \text{dom } f \rrbracket$ 
 $\implies \text{inv } (\text{the} \circ (\text{Some} ++ f)) = \text{the} \circ \text{map-inv } (\text{Some} ++ f)$ 
  apply (rule ext)
  apply (simp add: map-add-Some)
  apply (simp add: inv-def)
  apply (rename-tac x)
  apply (case-tac  $\exists y. f y = \text{Some } x$ )
  apply (erule exE)
  apply (rename-tac x y)
  apply (subgoal-tac  $x \in \text{ran } f$ )
  apply (subgoal-tac  $y \in \text{dom } f$ )
  apply (simp)
  apply (rule some-equality)
  apply (simp)
  apply (metis (hide-lams, mono-tags) domD domI dom-left-map-add inj-on-contrad map-add-Some
map-add-dom-app-simps(3) option.sel)
  apply (simp add: dom-def)
  apply (metis ranI)
  apply (simp)
  apply (rename-tac x)
  apply (subgoal-tac  $x \notin \text{ran } f$ )
  apply (simp)
  apply (rule some-equality)
  apply (simp)
  apply (metis domD dom-left-map-add map-add-Some map-add-dom-app-simps(3) option.sel)
  apply (metis dom-image-ran image-iff)
done

```

```

lemma map-comp-dom:  $\text{dom } (g \circ_m f) \subseteq \text{dom } f$ 
  by (metis (lifting, full-types) Collect-mono dom-def map-comp-simps(1))

```

```

lemma map-comp-assoc:  $f \circ_m (g \circ_m h) = f \circ_m g \circ_m h$ 

```

```

proof

```

```

  fix x

```

```

  show  $(f \circ_m (g \circ_m h)) x = (f \circ_m g \circ_m h) x$ 

```

```

  proof (cases h x)

```

```

    case None thus ?thesis

```

```

      by (auto simp add: map-comp-def)

```

```

  next

```

```

    case (Some y) thus ?thesis

```

```

      by (auto simp add: map-comp-def)

```

```

  qed

```

```

qed

```

```

lemma map-comp-runit [simp]:  $f \circ_m \text{Some} = f$ 

```

```

  by (simp add: map-comp-def)

```

```

lemma map-comp-lunit [simp]:  $\text{Some} \circ_m f = f$ 

```

```

proof

```

```

  fix x

```

```

  show  $(\text{Some} \circ_m f) x = f x$ 

```

```

  proof (cases f x)

```

```

    case None thus ?thesis

```

```

      by (simp add: map-comp-def)

```

```

next
  case (Some y) thus ?thesis
    by (simp add: map-comp-def)
qed
qed

```

```

lemma map-comp-apply [simp]: (f ◦m g) x = g(x) >>= f
  by (auto simp add: map-comp-def option.case-eq-if)

```

7.9 Merging of compatible maps

definition *comp-map* :: ('a → 'b) ⇒ ('a → 'b) ⇒ bool (infixl \parallel_m 60) **where**
comp-map f g = (∀ x ∈ dom(f) ∩ dom(g). the(f(x)) = the(g(x)))

```

lemma comp-map-unit: Map.empty  $\parallel_m$  f
  by (simp add: comp-map-def)

```

```

lemma comp-map-refl: f  $\parallel_m$  f
  by (simp add: comp-map-def)

```

```

lemma comp-map-sym: f  $\parallel_m$  g ⇒ g  $\parallel_m$  f
  by (simp add: comp-map-def)

```

definition *merge* :: ('a → 'b) set ⇒ 'a → 'b **where**
merge fs =
 (λ x. if (∃ f ∈ fs. x ∈ dom(f)) then (THE y. ∀ f ∈ fs. x ∈ dom(f) → f(x) = y) else None)

```

lemma merge-empty: merge {} = Map.empty
  by (simp add: merge-def)

```

```

lemma merge-singleton: merge {f} = f
  apply (auto intro!: ext simp add: merge-def)
  using option.collapse apply fastforce
  done

```

7.10 Conversion between lists and maps

definition *map-of-list* :: 'a list ⇒ (nat → 'a) **where**
map-of-list xs = (λ i. if (i < length xs) then Some (xs[i]) else None)

```

lemma map-of-list-nil [simp]: map-of-list [] = Map.empty
  by (simp add: map-of-list-def)

```

```

lemma dom-map-of-list [simp]: dom (map-of-list xs) = {0.. $\text{length } xs$ }
  by (auto simp add: map-of-list-def dom-def)

```

```

lemma ran-map-of-list [simp]: ran (map-of-list xs) = set xs
  apply (simp add: ran-def map-of-list-def)
  apply (safe)
  apply (force)
  apply (meson in-set-conv-nth)
  done

```

definition *list-of-map* :: (nat → 'a) ⇒ 'a list **where**
list-of-map f = (if (f = Map.empty) then [] else map (the ∘ f) [0 ..< Suc(GREATEST x. x ∈ dom f)])

lemma *list-of-map-empty* [simp]: *list-of-map Map.empty* = []
by (*simp add: list-of-map-def*)

definition *list-of-map'* :: (nat \rightarrow 'a) \rightarrow 'a list **where**
list-of-map' f = (if (\exists n. dom *f* = {0..*n*}) then *Some (list-of-map f)* else *None*)

lemma *map-of-list-inv* [simp]: *list-of-map (map-of-list xs)* = *xs*

proof (*cases xs* = [])

case *True* **thus** ?thesis **by** (*simp*)

next

case *False*

moreover **hence** (*GREATEST x. x \in dom (map-of-list xs) = length xs - 1*)

by (*auto intro: Greatest-equality*)

moreover **from** *False* **have** *map-of-list xs \neq Map.empty*

by (*metis ran-empty ran-map-of-list set-empty*)

ultimately **show** ?thesis

by (*auto simp add: list-of-map-def map-of-list-def nth-equalityI*)

qed

7.11 Map Comprehension

Map comprehension simply converts a relation built through set comprehension into a map.

syntax

-Mapcompr :: 'a \Rightarrow 'b \Rightarrow idts \Rightarrow bool \Rightarrow 'a \rightarrow 'b ((1[- \mapsto - | / - / -]))

translations

-Mapcompr F G xs P == *CONST graph-map {(F, G) | xs. P}*

lemma *map-compr-eta*:

$[x \mapsto y \mid x y. (x, y) \in_m f] = f$

apply (*rule ext*)

apply (*auto simp add: graph-map-def*)

apply (*metis (mono-tags, lifting) Domain.DomainI fst-eq-Domain mem-Collect-eq old.prod.case option.distinct(1) option.expand option.sel*)

done

lemma *map-compr-simple*:

$[x \mapsto F x y \mid x y. (x, y) \in_m f] = (\lambda x. \text{do } \{ y \leftarrow f(x); \text{Some}(F x y) \})$

apply (*rule ext*)

apply (*auto simp add: graph-map-def image-Collect*)

done

lemma *map-compr-dom-simple* [simp]:

dom $[x \mapsto f x \mid x. P x] = \{x. P x\}$

by (*force simp add: graph-map-dom image-Collect*)

lemma *map-compr-ran-simple* [simp]:

ran $[x \mapsto f x \mid x. P x] = \{f x \mid x. P x\}$

apply (*auto simp add: graph-map-def ran-def*)

apply (*metis (mono-tags, lifting) fst-eqD image-eqI mem-Collect-eq someI*)

done

lemma *map-compr-eval-simple* [simp]:

$[x \mapsto f x \mid x. P x] x = (\text{if } (P x) \text{ then } \text{Some } (f x) \text{ else } \text{None})$

by (*auto simp add: graph-map-def image-Collect*)

7.12 Sorted lists from maps

definition *sorted-list-of-map* :: $('a::\text{linorder} \rightarrow 'b) \Rightarrow ('a \times 'b) \text{ list}$ **where**
sorted-list-of-map $f = \text{map } (\lambda k. (k, \text{the } (f k))) (\text{sorted-list-of-set}(\text{dom}(f)))$

lemma *sorted-list-of-map-empty* [simp]:
sorted-list-of-map $\text{Map.empty} = []$
by (simp add: sorted-list-of-map-def)

lemma *sorted-list-of-map-inv*:
assumes $\text{finite}(\text{dom}(f))$
shows $\text{map-of } (\text{sorted-list-of-map } f) = f$
proof –
obtain A **where** $\text{finite } A \wedge A = \text{dom}(f)$
by (simp add: assms)
thus ?thesis
proof (induct A rule: finite-induct)
case empty **thus** ?thesis
by (simp add: sorted-list-of-map-def, metis dom-empty empty-iff map-le-antisym map-le-def)
next
case (insert $x A$) **thus** ?thesis
by (simp add: assms sorted-list-of-map-def map-of-map-keys)
qed
qed

declare *map-member.simps* [simp del]

7.13 Extra map lemmas

lemma *map-eqI*:
 $\llbracket \text{dom } f = \text{dom } g; \forall x \in \text{dom}(f). \text{the}(f x) = \text{the}(g x) \rrbracket \Longrightarrow f = g$
by (metis domIff map-le-antisym map-le-def option.expand)

lemma *map-restrict-dom* [simp]: $f \restriction \text{dom } f = f$
by (simp add: map-eqI)

lemma *map-restrict-dom-compl*: $f \restriction (- \text{dom } f) = \text{Map.empty}$
by (metis dom-eq-empty-conv dom-restrict inf-compl-bot)

lemma *restrict-map-neg-disj*:
 $\text{dom}(f) \cap A = \{\} \Longrightarrow f \restriction (- A) = f$
by (auto simp add: restrict-map-def, rule ext, auto, metis disjoint-iff-not-equal domIff)

lemma *map-plus-restrict-dist*: $(f ++ g) \restriction A = (f \restriction A) ++ (g \restriction A)$
by (auto simp add: restrict-map-def map-add-def)

lemma *map-plus-eq-left*:
assumes $f ++ h = g ++ h$
shows $(f \restriction (- \text{dom } h)) = (g \restriction (- \text{dom } h))$
proof –
have $h \restriction (- \text{dom } h) = \text{Map.empty}$
by (metis Compl-disjoint dom-eq-empty-conv dom-restrict)
then have $f2: f \restriction (- \text{dom } h) = (f ++ h) \restriction (- \text{dom } h)$
by (simp add: map-plus-restrict-dist)
have $h \restriction (- \text{dom } h) = \text{Map.empty}$
by (metis (no-types) Compl-disjoint dom-eq-empty-conv dom-restrict)

```

then show ?thesis
  using f2 assms by (simp add: map-plus-restrict-dist)
qed

lemma map-add-split:
  dom(f) = A ∪ B ⟹ (f |A) ++ (f |B) = f
  by (rule ext, auto simp add: map-add-def restrict-map-def option.case-eq-if)

lemma map-le-via-restrict:
  f ⊆m g ⟷ g |dom(f) = f
  by (auto simp add: map-le-def restrict-map-def dom-def fun-eq-iff)

lemma map-add-cancel:
  f ⊆m g ⟹ f ++ (g -- f) = g
  by (auto simp add: map-le-def map-add-def map-minus-def fun-eq-iff option.case-eq-if)
  (metis domIff)

lemma map-le-iff-add: f ⊆m g ⟷ (∃ h. dom(f) ∩ dom(h) = {} ∧ f ++ h = g)
  apply (auto)
  apply (rule-tac x=g -- f in exI)
  apply (metis (no-types, lifting) Int-emptyI domIff map-add-cancel map-le-def map-minus-def)
  apply (simp add: map-add-comm)
  done

lemma map-add-comm-weak: (∀ k ∈ dom m1 ∩ dom m2. m1(k) = m2(k)) ⟹ m1 ++ m2 = m2 ++ m1
  by (auto simp add: map-add-def option.case-eq-if fun-eq-iff)
  (metis IntI domI option.inject)

end

```

8 Alternative List Lexicographic Order

```

theory List-Lexord-Alt
  imports Main
begin

```

Since we can't instantiate the order class twice for lists, and we want prefix as the default order for the UTP we here add syntax for the lexicographic order relation.

```

definition list-lex-less :: 'a::linorder list ⇒ 'a list ⇒ bool (infix <l 50)
where xs <l ys ⟷ (xs, ys) ∈ lexord {(u, v). u < v}

```

```

lemma list-lex-less-neq [simp]: x <l y ⟹ x ≠ y
  apply (simp add: list-lex-less-def)
  apply (meson case-prodD less-irrefl lexord-irreflexive mem-Collect-eq)
done

```

```

lemma not-less-Nil [simp]: ¬ x <l []
  by (simp add: list-lex-less-def)

```

```

lemma Nil-less-Cons [simp]: [] <l a # x
  by (simp add: list-lex-less-def)

```

```

lemma Cons-less-Cons [simp]: a # x <l b # y ⟷ a < b ∨ a = b ∧ x <l y
  by (simp add: list-lex-less-def)

```

end

9 Partial Functions

```
theory Partial-Fun
imports Optics.Lenses Map-Extra
begin
```

I'm not completely satisfied with partial functions as provided by Map.thy, since they don't have a unique type and so we can't instantiate classes, make use of adhoc-overloading etc. Consequently I've created a new type and derived the laws.

9.1 Partial function type and operations

```
typedef ('a, 'b) pfun = UNIV :: ('a  $\rightarrow$  'b) set ..
```

```
setup-lifting type-definition-pfun
```

```
lift-definition pfun-app :: ('a, 'b) pfun  $\Rightarrow$  'a  $\Rightarrow$  'b (-'(-) p [999,0] 999) is
 $\lambda f x.$  if  $(x \in \text{dom } f)$  then the  $(f x)$  else undefined .
```

```
lift-definition pfun-upd :: ('a, 'b) pfun  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  ('a, 'b) pfun
is  $\lambda f k v.$   $f(k := \text{Some } v)$  .
```

```
lift-definition pdom :: ('a, 'b) pfun  $\Rightarrow$  'a set is dom .
```

```
lift-definition pran :: ('a, 'b) pfun  $\Rightarrow$  'b set is ran .
```

```
lift-definition pfun-comp :: ('b, 'c) pfun  $\Rightarrow$  ('a, 'b) pfun  $\Rightarrow$  ('a, 'c) pfun (infixl  $\circ_p$  55) is map-comp .
```

```
lift-definition pfun-member :: 'a  $\times$  'b  $\Rightarrow$  ('a, 'b) pfun  $\Rightarrow$  bool (infix  $\in_p$  50) is ( $\in_m$ ) .
```

```
lift-definition pId-on :: 'a set  $\Rightarrow$  ('a, 'a) pfun is  $\lambda A x.$  if  $(x \in A)$  then Some  $x$  else None .
```

```
abbreviation pId :: ('a, 'a) pfun where
pId  $\equiv$  pId-on UNIV
```

```
lift-definition plambda :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) pfun
is  $\lambda P f x.$  if  $(P x)$  then Some  $(f x)$  else None .
```

```
lift-definition pdom-res :: 'a set  $\Rightarrow$  ('a, 'b) pfun  $\Rightarrow$  ('a, 'b) pfun (infixr  $\triangleleft_p$  85)
is  $\lambda A f.$  restrict-map  $f A$  .
```

```
lift-definition pran-res :: ('a, 'b) pfun  $\Rightarrow$  'b set  $\Rightarrow$  ('a, 'b) pfun (infixl  $\triangleright_p$  85)
is ran-restrict-map .
```

```
lift-definition pfun-graph :: ('a, 'b) pfun  $\Rightarrow$  ('a  $\times$  'b) set is map-graph .
```

```
lift-definition graph-pfun :: ('a  $\times$  'b) set  $\Rightarrow$  ('a, 'b) pfun is graph-map .
```

```
lift-definition pfun-entries :: 'k set  $\Rightarrow$  ('k  $\Rightarrow$  'v)  $\Rightarrow$  ('k, 'v) pfun is
 $\lambda d f x.$  if  $(x \in d)$  then Some  $(f x)$  else None .
```

```
definition pcard :: ('a, 'b) pfun  $\Rightarrow$  nat
where pcard  $f = \text{card } (\text{pdom } f)$ 
```



```

instantiation pfun :: (type, type) zero
begin
lift-definition zero-pfun :: ('a', 'b') pfun is Map.empty .
instance ..
end

abbreviation pempty :: ('a', 'b') pfun ( $\{\}_p$ )
where pempty  $\equiv$  0

instantiation pfun :: (type, type) plus
begin
lift-definition plus-pfun :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun is (++) .
instance ..
end

instantiation pfun :: (type, type) minus
begin
lift-definition minus-pfun :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun is (--) .
instance ..
end

instance pfun :: (type, type) monoid-add
  by (intro-classes, (transfer, auto)+)

instantiation pfun :: (type, type) inf
begin
lift-definition inf-pfun :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun is
 $\lambda f g x. \text{if } (x \in \text{dom}(f) \cap \text{dom}(g) \wedge f(x) = g(x)) \text{ then } f(x) \text{ else } \text{None}$  .
instance ..
end

abbreviation pfun-inter :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun (infixl  $\cap_p$  80)
where pfun-inter  $\equiv$  inf

instantiation pfun :: (type, type) order
begin
  lift-definition less-eq-pfun :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  bool is
 $\lambda f g. f \subseteq_m g$  .
  lift-definition less-pfun :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  bool is
 $\lambda f g. f \subseteq_m g \wedge f \neq g$  .
instance
  by (intro-classes, (transfer, auto intro: map-le-trans simp add: map-le-antisym)+)
end

abbreviation pfun-subset :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  bool (infix  $\subset_p$  50)
where pfun-subset  $\equiv$  less

abbreviation pfun-subset-eq :: ('a', 'b') pfun  $\Rightarrow$  ('a', 'b') pfun  $\Rightarrow$  bool (infix  $\subseteq_p$  50)
where pfun-subset-eq  $\equiv$  less-eq

instance pfun :: (type, type) semilattice-inf
  by (intro-classes, (transfer, auto simp add: map-le-def dom-def)+)

lemma pfun-subset-eq-least [simp]:

```

$\{\}_p \subseteq_p f$
by (*transfer*, *auto*)

syntax

$-PfunUpd :: [('a, 'b) pfun, maplets] => ('a, 'b) pfun (-'(-)_p [900,0]900)$
 $-Pfun :: maplets => ('a, 'b) pfun ((1\{-\}_p))$
 $-plam :: pptrn \Rightarrow logic \Rightarrow logic \Rightarrow logic (\lambda - | - . - [0,0,10] 10)$

translations

$-PfunUpd m (-Maplets xy ms) == -PfunUpd (-PfunUpd m xy) ms$
 $-PfunUpd m (-maplet x y) == CONST pfun-upd m x y$
 $-Pfun ms ==> -PfunUpd (CONST pempty) ms$
 $-Pfun (-Maplets ms1 ms2) <= -PfunUpd (-Pfun ms1) ms2$
 $-Pfun ms <= -PfunUpd (CONST pempty) ms$
 $\lambda x | P . e ==> CONST plambda (\lambda x. P) (\lambda x. e)$
 $\lambda x | P . e <= CONST plambda (\lambda x. P) (\lambda y. e)$
 $\lambda y | P . e <= CONST plambda (\lambda x. P) (\lambda y. e)$
 $\lambda y | f v y . e <= CONST plambda (f v) (\lambda y. e)$

9.2 Algebraic laws

lemma *pfun-comp-assoc*: $f \circ_p (g \circ_p h) = (f \circ_p g) \circ_p h$
by (*transfer*, *simp add: map-comp-assoc*)

lemma *pfun-comp-left-id* [*simp*]: $pId \circ_p f = f$
by (*transfer*, *auto*)

lemma *pfun-comp-right-id* [*simp*]: $f \circ_p pId = f$
by (*transfer*, *auto*)

lemma *pfun-override-dist-comp*:
 $(f + g) \circ_p h = (f \circ_p h) + (g \circ_p h)$
apply (*transfer*)
apply (*rule ext*)
apply (*auto simp add: map-add-def*)
apply (*rename-tac f g h x*)
apply (*case-tac h x*)
apply (*auto*)
apply (*rename-tac f g h x y*)
apply (*case-tac g y*)
apply (*auto*)
done

lemma *pfun-minus-unit* [*simp*]:
fixes $f :: ('a, 'b) pfun$
shows $f - 0 = f$
by (*transfer*, *simp add: map-minus-def*)

lemma *pfun-minus-zero* [*simp*]:
fixes $f :: ('a, 'b) pfun$
shows $0 - f = 0$
by (*transfer*, *simp add: map-minus-def*)

lemma *pfun-minus-self* [*simp*]:
fixes $f :: ('a, 'b) pfun$
shows $f - f = 0$

by (transfer, simp add: map-minus-def)

lemma pfun-plus-commute:

$\text{pdom}(f) \cap \text{pdom}(g) = \{\} \implies f + g = g + f$
 by (transfer, metis map-add-comm)

lemma pfun-plus-commute-weak:

$(\forall k \in \text{pdom}(f) \cap \text{pdom}(g). f(k)_p = g(k)_p) \implies f + g = g + f$
 by (transfer, simp, metis IntD1 IntD2 domD map-add-comm-weak option.sel)

lemma pfun-minus-plus-commute:

$\text{pdom}(g) \cap \text{pdom}(h) = \{\} \implies (f - g) + h = (f + h) - g$
 by (transfer, simp add: map-minus-plus-commute)

lemma pfun-plus-minus:

$f \subseteq_p g \implies (g - f) + f = g$
 by (transfer, rule ext, auto simp add: map-le-def map-minus-def map-add-def option.case-eq-if)

lemma pfun-minus-common-subset:

$\llbracket h \subseteq_p f; h \subseteq_p g \rrbracket \implies (f - h = g - h) = (f = g)$
 by (transfer, simp add: map-minus-common-subset)

lemma pfun-minus-plus:

$\text{pdom}(f) \cap \text{pdom}(g) = \{\} \implies (f + g) - g = f$
 by (transfer, simp add: map-add-def map-minus-def option.case-eq-if, rule ext, auto)
 (metis Int-commute domIff insert-disjoint(1) insert-dom)

lemma pfun-plus-pos: $x + y = \{\}_p \implies x = \{\}_p$

by (transfer, simp)

lemma pfun-le-plus: $\text{pdom } x \cap \text{pdom } y = \{\} \implies x \leq x + y$

by (transfer, auto simp add: map-le-iff-add)

9.3 Lambda abstraction

lemma plambda-app [simp]: $(\lambda x \mid P x . f x)(v)_p = (\text{if } (P v) \text{ then } (f v) \text{ else undefined})$

by (transfer, auto)

lemma plambda-eta [simp]: $(\lambda x \mid x \in \text{pdom}(f). f(x)_p) = f$

by (transfer; auto simp add: domIff)

lemma plambda-id [simp]: $(\lambda x \mid P x . x) = \text{pId-on } \{x. P x\}$

by (transfer, simp)

9.4 Membership, application, and update

lemma pfun-ext: $\llbracket \bigwedge x y. (x, y) \in_p f \longleftrightarrow (x, y) \in_p g \rrbracket \implies f = g$

by (transfer, simp add: map-ext)

lemma pfun-member-alt-def:

$(x, y) \in_p f \longleftrightarrow (x \in \text{pdom } f \wedge f(x)_p = y)$
 by (transfer, auto simp add: map-member-alt-def map-apply-def)

lemma pfun-member-plus:

$(x, y) \in_p f + g \longleftrightarrow ((x \notin \text{pdom}(g) \wedge (x, y) \in_p f) \vee (x, y) \in_p g)$
 by (transfer, simp add: map-member-plus)

lemma *pfun-member-minus*:

$(x, y) \in_p f - g \longleftrightarrow (x, y) \in_p f \wedge (\neg (x, y) \in_p g)$
by (*transfer*, *simp add: map-member-minus*)

lemma *pfun-app-upd-1* [*simp*]: $x = y \implies (f(x \mapsto v)_p)(y)_p = v$
by (*transfer*, *simp*)

lemma *pfun-app-upd-2* [*simp*]: $x \neq y \implies (f(x \mapsto v)_p)(y)_p = f(y)_p$
by (*transfer*, *simp*)

lemma *pfun-graph-apply* [*simp*]: *rel-apply* (*pfun-graph* *f*) $x = f(x)_p$
by (*transfer*, *auto simp add: rel-apply-def map-graph-def*)

lemma *pfun-upd-ext* [*simp*]: $x \in \text{pdom}(f) \implies f(x \mapsto f(x)_p)_p = f$
by (*transfer*, *simp add: domIff*)

lemma *pfun-app-add* [*simp*]: $x \in \text{pdom}(g) \implies (f + g)(x)_p = g(x)_p$
by (*transfer*, *auto*)

lemma *pfun-upd-add* [*simp*]: $f + g(x \mapsto v)_p = (f + g)(x \mapsto v)_p$
by (*transfer*, *simp*)

lemma *pfun-upd-twice* [*simp*]: $f(x \mapsto u, x \mapsto v)_p = f(x \mapsto v)_p$
by (*transfer*, *simp*)

lemma *pfun-upd-comm*:

assumes $x \neq y$
shows $f(y \mapsto u, x \mapsto v)_p = f(x \mapsto v, y \mapsto u)_p$
using *assms* **by** (*transfer*, *auto*)

lemma *pfun-upd-comm-linorder* [*simp*]:

fixes $x\ y :: 'a :: \text{linorder}$
assumes $x < y$
shows $f(y \mapsto u, x \mapsto v)_p = f(x \mapsto v, y \mapsto u)_p$
using *assms* **by** (*transfer*, *auto*)

lemma *pfun-app-minus* [*simp*]: $x \notin \text{pdom } g \implies (f - g)(x)_p = f(x)_p$
by (*transfer*, *auto simp add: map-minus-def*)

lemma *pfun-app-empty* [*simp*]: $\{\}_p(x)_p = \text{undefined}$
by (*transfer*, *simp*)

lemma *pfun-app-not-in-dom*:

$x \notin \text{pdom}(f) \implies f(x)_p = \text{undefined}$
by (*transfer*, *simp*)

lemma *pfun-upd-minus* [*simp*]:

$x \notin \text{pdom } g \implies (f - g)(x \mapsto v)_p = (f(x \mapsto v)_p - g)$
by (*transfer*, *auto simp add: map-minus-def*)

lemma *pdom-member-minus-iff* [*simp*]:

$x \notin \text{pdom } g \implies x \in \text{pdom}(f - g) \longleftrightarrow x \in \text{pdom}(f)$
by (*transfer*, *simp add: domIff map-minus-def*)

lemma *psubseteq-pfun-upd1* [*intro*]:
 $\llbracket f \subseteq_p g; x \notin \text{pdom}(g) \rrbracket \implies f \subseteq_p g(x \mapsto v)_p$
by (*transfer*, *auto simp add: map-le-def dom-def*)

lemma *psubseteq-pfun-upd2* [*intro*]:
 $\llbracket f \subseteq_p g; x \notin \text{pdom}(f) \rrbracket \implies f \subseteq_p g(x \mapsto v)_p$
by (*transfer*, *auto simp add: map-le-def dom-def*)

lemma *psubseteq-pfun-upd3* [*intro*]:
 $\llbracket f \subseteq_p g; g(x)_p = v \rrbracket \implies f \subseteq_p g(x \mapsto v)_p$
by (*transfer*, *auto simp add: map-le-def dom-def*)

lemma *psubseteq-dom-subset*:
 $f \subseteq_p g \implies \text{pdom}(f) \subseteq \text{pdom}(g)$
by (*transfer*, *auto simp add: map-le-def dom-def*)

lemma *psubseteq-ran-subset*:
 $f \subseteq_p g \implies \text{pran}(f) \subseteq \text{pran}(g)$
by (*transfer*, *auto simp add: map-le-def dom-def ran-def, fastforce*)

9.5 Domain laws

lemma *pdom-zero* [*simp*]: $\text{pdom } 0 = \{\}$
by (*transfer*, *simp*)

lemma *pdom-pId-on* [*simp*]: $\text{pdom } (\text{pId-on } A) = A$
by (*transfer*, *auto*)

lemma *pdom-plus* [*simp*]: $\text{pdom } (f + g) = \text{pdom } f \cup \text{pdom } g$
by (*transfer*, *auto*)

lemma *pdom-minus* [*simp*]: $g \leq f \implies \text{pdom } (f - g) = \text{pdom } f - \text{pdom } g$
apply (*transfer*, *auto simp add: map-minus-def*)
apply (*meson option.distinct*(1))
apply (*metis domIff map-le-def option.simps*(3))
done

lemma *pdom-inter*: $\text{pdom } (f \cap_p g) \subseteq \text{pdom } f \cap \text{pdom } g$
by (*transfer*, *auto simp add: dom-def*)

lemma *pdom-comp* [*simp*]: $\text{pdom } (g \circ_p f) = \text{pdom } (f \triangleright_p \text{pdom } g)$
by (*transfer*, *auto simp add: ran-restrict-map-def*)

lemma *pdom-upd* [*simp*]: $\text{pdom } (f(k \mapsto v)_p) = \text{insert } k (\text{pdom } f)$
by (*transfer*, *simp*)

lemma *pdom-plamda* [*simp*]: $\text{pdom } (\lambda x \mid P x . f x) = \{x. P x\}$
by (*transfer*, *auto*)

lemma *pdom-pdom-res* [*simp*]: $\text{pdom } (A \triangleleft_p f) = A \cap \text{pdom}(f)$
by (*transfer*, *auto*)

lemma *pdom-graph-pfun* [*simp*]: $\text{pdom } (\text{graph-pfun } R) = \text{Domain } R$
by (*transfer*, *simp add: Domain-fst graph-map-dom*)

lemma *pdom-pran-res-finite* [*simp*]:

finite (*pdom* *f*) \implies *finite* (*pdom* (*f* \triangleright_p *A*))
 by (*transfer*, *auto*)

lemma *pdom-pfun-graph-finite* [*simp*]:
finite (*pdom* *f*) \implies *finite* (*pfun-graph* *f*)
 by (*transfer*, *simp* add: *finite-dom-graph*)

9.6 Range laws

lemma *pran-zero* [*simp*]: *pran* 0 = {}
 by (*transfer*, *simp*)

lemma *pran-pId-on* [*simp*]: *pran* (*pId-on* *A*) = *A*
 by (*transfer*, *auto* *simp* add: *ran-def*)

lemma *pran-upd* [*simp*]: *pran* (*f* (*k* \mapsto *v*)_{*p*}) = *insert* *v* (*pran* ((- {*k*}) \triangleleft_p *f*))
 by (*transfer*, *auto* *simp* add: *ran-def restrict-map-def*)

lemma *pran-plamda* [*simp*]: *pran* ($\lambda x \mid P x . f x$) = {*f* *x* \mid *x*. *P* *x*}
 by (*transfer*, *auto* *simp* add: *ran-def*)

lemma *pran-pran-res* [*simp*]: *pran* (*f* \triangleright_p *A*) = *pran*(*f*) \cap *A*
 by (*transfer*, *auto*)

lemma *pran-comp* [*simp*]: *pran* (*g* \circ_p *f*) = *pran* (*pran* *f* \triangleleft_p *g*)
 by (*transfer*, *auto* *simp* add: *ran-def restrict-map-def*)

lemma *pran-finite* [*simp*]: *finite* (*pdom* *f*) \implies *finite* (*pran* *f*)
 by (*transfer*, *auto*)

9.7 Domain restriction laws

lemma *pdom-res-zero* [*simp*]: *A* \triangleleft_p {}_{*p*} = {}_{*p*}
 by (*transfer*, *auto*)

lemma *pdom-res-empty* [*simp*]:
 ({} \triangleleft_p *f*) = {}_{*p*}
 by (*transfer*, *auto*)

lemma *pdom-res-pdom* [*simp*]:
pdom(*f*) \triangleleft_p *f* = *f*
 by (*transfer*, *auto*)

lemma *pdom-res-UNIV* [*simp*]: *UNIV* \triangleleft_p *f* = *f*
 by (*transfer*, *auto*)

lemma *pdom-res-alt-def*: *A* \triangleleft_p *f* = *f* \circ_p *pId-on* *A*
 by (*transfer*, *rule ext*, *auto* *simp* add: *restrict-map-def*)

lemma *pdom-res-upd-in* [*simp*]:
k \in *A* \implies *A* \triangleleft_p *f* (*k* \mapsto *v*)_{*p*} = (*A* \triangleleft_p *f*) (*k* \mapsto *v*)_{*p*}
 by (*transfer*, *auto*)

lemma *pdom-res-upd-out* [*simp*]:
k \notin *A* \implies *A* \triangleleft_p *f* (*k* \mapsto *v*)_{*p*} = *A* \triangleleft_p *f*
 by (*transfer*, *auto*)

lemma *pfun-pdom-antires-upd* [simp]:
 $k \in A \implies ((- A) \triangleleft_p m)(k \mapsto v)_p = ((- (A - \{k\})) \triangleleft_p m)(k \mapsto v)_p$
 by (transfer, simp)

lemma *pdom-antires-insert-notin* [simp]:
 $k \notin \text{pdom}(f) \implies (- \text{insert } k A) \triangleleft_p f = (- A) \triangleleft_p f$
 by (transfer, auto simp add: restrict-map-def)

lemma *pdom-res-override* [simp]: $A \triangleleft_p (f + g) = (A \triangleleft_p f) + (A \triangleleft_p g)$
 by (simp add: pdom-res-alt-def pfun-override-dist-comp)

lemma *pdom-res-minus* [simp]: $A \triangleleft_p (f - g) = (A \triangleleft_p f) - g$
 by (transfer, auto simp add: map-minus-def restrict-map-def)

lemma *pdom-res-swap*: $(A \triangleleft_p f) \triangleright_p B = A \triangleleft_p (f \triangleright_p B)$
 by (transfer, auto simp add: restrict-map-def ran-restrict-map-def)

lemma *pdom-res-twice* [simp]: $A \triangleleft_p (B \triangleleft_p f) = (A \cap B) \triangleleft_p f$
 by (transfer, auto simp add: Int-commute)

lemma *pdom-res-comp* [simp]: $A \triangleleft_p (g \circ_p f) = g \circ_p (A \triangleleft_p f)$
 by (simp add: pdom-res-alt-def pfun-comp-assoc)

lemma *pdom-res-apply* [simp]:
 $x \in A \implies (A \triangleleft_p f)(x)_p = f(x)_p$
 by (transfer, auto)

9.8 Range restriction laws

lemma *pran-res-zero* [simp]: $\{\}_p \triangleright_p A = \{\}_p$
 by (transfer, auto simp add: ran-restrict-map-def)

lemma *pran-res-upd-1* [simp]: $v \in A \implies f(x \mapsto v)_p \triangleright_p A = (f \triangleright_p A)(x \mapsto v)_p$
 by (transfer, auto simp add: ran-restrict-map-def)

lemma *pran-res-upd-2* [simp]: $v \notin A \implies f(x \mapsto v)_p \triangleright_p A = ((- \{x\}) \triangleleft_p f) \triangleright_p A$
 by (transfer, auto simp add: ran-restrict-map-def)

lemma *pran-res-alt-def*: $f \triangleright_p A = \text{pId-on } A \circ_p f$
 by (transfer, rule ext, auto simp add: ran-restrict-map-def)

lemma *pran-res-override*: $(f + g) \triangleright_p A \subseteq_p (f \triangleright_p A) + (g \triangleright_p A)$
 apply (transfer, auto simp add: map-add-def ran-restrict-map-def map-le-def)
 apply (rename-tac f g A a y x)
 apply (case-tac g a)
 apply (auto)
 done

9.9 Graph laws

lemma *pfun-graph-inv*: $\text{graph-pfun } (\text{pfun-graph } f) = f$
 by (transfer, simp)

lemma *pfun-graph-zero*: $\text{pfun-graph } 0 = \{\}$
 by (transfer, simp add: map-graph-def)

lemma *pfun-graph-pId-on*: $\text{pfun-graph } (pId\text{-on } A) = Id\text{-on } A$
by (*transfer*, *auto simp add: map-graph-def*)

lemma *pfun-graph-minus*: $\text{pfun-graph } (f - g) = \text{pfun-graph } f - \text{pfun-graph } g$
by (*transfer*, *simp add: map-graph-minus*)

lemma *pfun-graph-inter*: $\text{pfun-graph } (f \cap_p g) = \text{pfun-graph } f \cap \text{pfun-graph } g$
apply (*transfer*, *auto simp add: map-graph-def*)
apply (*metis option.discI*)
done

9.10 Entries

lemma *pfun-entries-empty* [*simp*]: $\text{pfun-entries } \{\} f = \{\}_p$
by (*transfer*, *simp*)

lemma *pfun-entries-apply-1* [*simp*]:
 $x \in d \implies (\text{pfun-entries } d f)(x)_p = f x$
by (*transfer*, *auto*)

lemma *pfun-entries-apply-2* [*simp*]:
 $x \notin d \implies (\text{pfun-entries } d f)(x)_p = \text{undefined}$
by (*transfer*, *auto*)

9.11 Summation

definition *pfun-sum* :: $(k, v::\text{comm-monoid-add}) \text{ pfun} \Rightarrow v$ **where**
 $\text{pfun-sum } f = \text{sum } (\text{pfun-app } f) (\text{pdom } f)$

lemma *pfun-sum-empty* [*simp*]: $\text{pfun-sum } \{\}_p = 0$
by (*simp add: pfun-sum-def*)

lemma *pfun-sum-upd-1*:
assumes $\text{finite}(\text{pdom}(m))$ $k \notin \text{pdom}(m)$
shows $\text{pfun-sum } (m(k \mapsto v))_p = \text{pfun-sum } m + v$
by (*simp-all add: pfun-sum-def assms, metis add.commute assms(2) pfun-app-upd-2 sum.cong*)

lemma *pfun-sums-upd-2*:
assumes $\text{finite}(\text{pdom}(m))$
shows $\text{pfun-sum } (m(k \mapsto v))_p = \text{pfun-sum } ((-\{k\}) \triangleleft_p m) + v$

proof (*cases* $k \notin \text{pdom}(m)$)
case *True*
then show *?thesis*
by (*simp add: pfun-sum-upd-1 assms*)
next
case *False*
then show *?thesis*
using *assms pfun-sum-upd-1* [*of* $((-\{k\}) \triangleleft_p m) k v]$
by (*simp add: pfun-sum-upd-1*)

qed

lemma *pfun-sum-dom-res-insert* [*simp*]:
assumes $x \in \text{pdom } f$ $x \notin A$ $\text{finite } A$
shows $\text{pfun-sum } ((\text{insert } x A) \triangleleft_p f) = f(x)_p + \text{pfun-sum } (A \triangleleft_p f)$
using *assms* **by** (*simp add: pfun-sum-def*)


```

lemma pfun-sum-pdom-res:
  fixes  $f :: ('a, 'b :: ab\text{-group-add})\ pfun$ 
  assumes  $finite(pdom\ f)$ 
  shows  $pfun\text{-sum}\ (A \triangleleft_p f) = pfun\text{-sum}\ f - (pfun\text{-sum}\ ((- A) \triangleleft_p f))$ 
proof -
  have  $1:A \cap pdom(f) = pdom(f) - (pdom(f) - A)$ 
    by (auto)
  show ?thesis
    apply (simp add: pfun-sum-def)
    apply (subst 1)
    apply (subst sum-diff)
    apply (auto simp add: sum-diff Diff-subset Int-commute boolean-algebra-class.diff-eq assms)
  done
qed

```

```

lemma pfun-sum-pdom-antires [simp]:
  fixes  $f :: ('a, 'b :: ab\text{-group-add})\ pfun$ 
  assumes  $finite(pdom\ f)$ 
  shows  $pfun\text{-sum}\ ((- A) \triangleleft_p f) = pfun\text{-sum}\ f - pfun\text{-sum}\ (A \triangleleft_p f)$ 
  by (subst pfun-sum-pdom-res, simp-all add: assms)

```

9.12 Partial Function Lens

definition *pfun-lens* :: $'a \Rightarrow ('b \Rightarrow ('a, 'b)\ pfun)$ **where**
 $[lens\text{-defs}]$: $pfun\text{-lens}\ i = (\mid lens\text{-get} = \lambda s. s(i)_p, lens\text{-put} = \lambda s\ v. s(i \mapsto v)_p \mid)$

```

lemma pfun-lens-mwb [simp]:  $mwb\text{-lens}\ (pfun\text{-lens}\ i)$ 
  by (unfold-locales, simp-all add: pfun-lens-def)

```

```

lemma pfun-lens-src:  $\mathcal{S}_{pfun\text{-lens}\ i} = \{f. i \in pdom(f)\}$ 
  by (auto simp add: lens-defs lens-source-def, transfer, force)

```

Hide implementation details for partial functions

```

lifting-update pfun.lifting
lifting-forget pfun.lifting

```

end

10 Finite Functions

```

theory Finite-Fun
imports Map-Extra Partial-Fun FSet-Extra
begin

```

10.1 Finite function type and operations

```

typedef  $('a, 'b)\ ffun = \{f :: ('a, 'b)\ pfun. finite(pdom(f))\}$ 
  morphisms pfun-of Abs-pfun
  by (rule-tac x={}_p in exI, auto)

```

```

setup-lifting type-definition-ffun

```

```

lift-definition ffun-app ::  $('a, 'b)\ ffun \Rightarrow 'a \Rightarrow 'b\ (-'(-)_f\ [999,0]\ 999)$  is pfun-app .

```

lift-definition *ffun-upd* :: ('a, 'b) ffun \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) ffun **is** *pfun-upd* **by** *simp*

lift-definition *fdom* :: ('a, 'b) ffun \Rightarrow 'a set **is** *pdom* .

lift-definition *fran* :: ('a, 'b) ffun \Rightarrow 'b set **is** *pran* .

lift-definition *ffun-comp* :: ('b, 'c) ffun \Rightarrow ('a, 'b) ffun \Rightarrow ('a, 'c) ffun (**infixl** \circ_f 55) **is** *pfun-comp* **by** *auto*

lift-definition *ffun-member* :: 'a \times 'b \Rightarrow ('a, 'b) ffun \Rightarrow bool (**infix** \in_f 50) **is** (\in_p) .

lift-definition *fdom-res* :: 'a set \Rightarrow ('a, 'b) ffun \Rightarrow ('a, 'b) ffun (**infixl** \triangleleft_f 85)
is *pdom-res* **by** *simp*

lift-definition *fran-res* :: ('a, 'b) ffun \Rightarrow 'b set \Rightarrow ('a, 'b) ffun (**infixl** \triangleright_f 85)
is *pran-res* **by** *simp*

lift-definition *ffun-graph* :: ('a, 'b) ffun \Rightarrow ('a \times 'b) set **is** *pfun-graph* .

lift-definition *graph-ffun* :: ('a \times 'b) set \Rightarrow ('a, 'b) ffun **is**
 $\lambda R.$ if (finite (Domain R)) then *graph-pfun* R else *pempty*
by (*simp add: finite-Domain*)

instantiation *ffun* :: (type, type) zero
begin
lift-definition *zero-ffun* :: ('a, 'b) ffun **is** 0 **by** *simp*
instance ..
end

abbreviation *fempty* :: ('a, 'b) ffun ($\{\}_f$)
where *fempty* \equiv 0

instantiation *ffun* :: (type, type) plus
begin
lift-definition *plus-ffun* :: ('a, 'b) ffun \Rightarrow ('a, 'b) ffun \Rightarrow ('a, 'b) ffun **is** (+) **by** *simp*
instance ..
end

instantiation *ffun* :: (type, type) minus
begin
lift-definition *minus-ffun* :: ('a, 'b) ffun \Rightarrow ('a, 'b) ffun \Rightarrow ('a, 'b) ffun **is** (-)
by (*metis finite-Diff finite-Domain pdom-graph-pfun pdom-pfun-graph-finite pfun-graph-inv pfun-graph-minus*)
instance ..
end

instance *ffun* :: (type, type) monoid-add
by (*intro-classes, (transfer, simp add: add.assoc)* +)

instantiation *ffun* :: (type, type) inf
begin
lift-definition *inf-ffun* :: ('a, 'b) ffun \Rightarrow ('a, 'b) ffun \Rightarrow ('a, 'b) ffun **is** inf
by (*meson finite-Int infinite-super pdom-inter*)
instance ..
end

abbreviation $\text{ffun-inter} :: ('a, 'b) \text{ffun} \Rightarrow ('a, 'b) \text{ffun} \Rightarrow ('a, 'b) \text{ffun}$ (**infixl** \cap_f 80)
where $\text{ffun-inter} \equiv \text{inf}$

instantiation $\text{ffun} :: (\text{type}, \text{type}) \text{order}$

begin

lift-definition $\text{less-eq-ffun} :: ('a, 'b) \text{ffun} \Rightarrow ('a, 'b) \text{ffun} \Rightarrow \text{bool}$ **is**

$\lambda f g. f \subseteq_p g$.

lift-definition $\text{less-ffun} :: ('a, 'b) \text{ffun} \Rightarrow ('a, 'b) \text{ffun} \Rightarrow \text{bool}$ **is**

$\lambda f g. f < g$.

instance

by (*intro-classes*, (*transfer*, *auto*) $+$)

end

abbreviation $\text{ffun-subset} :: ('a, 'b) \text{ffun} \Rightarrow ('a, 'b) \text{ffun} \Rightarrow \text{bool}$ (**infix** \subset_f 50)

where $\text{ffun-subset} \equiv \text{less}$

abbreviation $\text{ffun-subset-eq} :: ('a, 'b) \text{ffun} \Rightarrow ('a, 'b) \text{ffun} \Rightarrow \text{bool}$ (**infix** \subseteq_f 50)

where $\text{ffun-subset-eq} \equiv \text{less-eq}$

instance $\text{ffun} :: (\text{type}, \text{type}) \text{semilattice-inf}$

by (*intro-classes*, (*transfer*, *auto*) $+$)

lemma $\text{ffun-subset-eq-least}$ [*simp*]:

$\{\}_f \subseteq_f f$

by (*transfer*, *auto*)

syntax

$\text{-FfunUpd} :: [('a, 'b) \text{ffun}, \text{maplets}] \Rightarrow ('a, 'b) \text{ffun}$ ($\text{-}(\text{'})_f$ [900,0]900)

$\text{-Ffun} :: \text{maplets} \Rightarrow ('a, 'b) \text{ffun}$ ($(\text{I}\{\text{-}\}_f)$)

translations

$\text{-FfunUpd } m \text{ (-Maplets } xy \text{ } ms) == \text{-FfunUpd } (\text{-FfunUpd } m \text{ } xy) \text{ } ms$

$\text{-FfunUpd } m \text{ (-maplet } x \text{ } y) == \text{CONST ffun-upd } m \text{ } x \text{ } y$

$\text{-Ffun } ms ==> \text{-FfunUpd } (\text{CONST fempty}) \text{ } ms$

$\text{-Ffun } (\text{-Maplets } ms1 \text{ } ms2) <= \text{-FfunUpd } (\text{-Ffun } ms1) \text{ } ms2$

$\text{-Ffun } ms <= \text{-FfunUpd } (\text{CONST fempty}) \text{ } ms$

10.2 Algebraic laws

lemma $\text{ffun-comp-assoc}: f \circ_f (g \circ_f h) = (f \circ_f g) \circ_f h$

by (*transfer*, *simp add: pfun-comp-assoc*)

lemma $\text{pfun-override-dist-comp}$:

$(f + g) \circ_f h = (f \circ_f h) + (g \circ_f h)$

by (*transfer*, *simp add: pfun-override-dist-comp*)

lemma ffun-minus-unit [*simp*]:

fixes $f :: ('a, 'b) \text{ffun}$

shows $f - 0 = f$

by (*transfer*, *simp*)

lemma ffun-minus-zero [*simp*]:

fixes $f :: ('a, 'b) \text{ffun}$

shows $0 - f = 0$

by (*transfer*, *simp*)

lemma *ffun-minus-self* [simp]:

fixes $f :: ('a, 'b) \text{ffun}$
 shows $f - f = 0$
 by (transfer, simp)

lemma *ffun-plus-commute*:

$\text{fdom}(f) \cap \text{fdom}(g) = \{\} \implies f + g = g + f$
 by (transfer, metis pfun-plus-commute)

lemma *ffun-minus-plus-commute*:

$\text{fdom}(g) \cap \text{fdom}(h) = \{\} \implies (f - g) + h = (f + h) - g$
 by (transfer, simp add: pfun-minus-plus-commute)

lemma *ffun-plus-minus*:

$f \subseteq_f g \implies (g - f) + f = g$
 by (transfer, simp add: pfun-plus-minus)

lemma *ffun-minus-common-subset*:

$\llbracket h \subseteq_f f; h \subseteq_f g \rrbracket \implies (f - h = g - h) = (f = g)$
 by (transfer, simp add: pfun-minus-common-subset)

lemma *ffun-minus-plus*:

$\text{fdom}(f) \cap \text{fdom}(g) = \{\} \implies (f + g) - g = f$
 by (transfer, simp add: pfun-minus-plus)

lemma *ffun-plus-pos*: $x + y = \{\}_f \implies x = \{\}_f$

by (transfer, simp add: pfun-plus-pos)

lemma *ffun-le-plus*: $\text{fdom } x \cap \text{fdom } y = \{\} \implies x \leq x + y$

by (transfer, simp add: pfun-le-plus)

10.3 Membership, application, and update

lemma *ffun-ext*: $\llbracket \bigwedge x y. (x, y) \in_f f \longleftrightarrow (x, y) \in_f g \rrbracket \implies f = g$

by (transfer, simp add: pfun-ext)

lemma *ffun-member-alt-def*:

$(x, y) \in_f f \longleftrightarrow (x \in \text{fdom } f \wedge f(x)_f = y)$
 by (transfer, simp add: pfun-member-alt-def)

lemma *ffun-member-plus*:

$(x, y) \in_f f + g \longleftrightarrow ((x \notin \text{fdom}(g) \wedge (x, y) \in_f f) \vee (x, y) \in_f g)$
 by (transfer, simp add: pfun-member-plus)

lemma *ffun-member-minus*:

$(x, y) \in_f f - g \longleftrightarrow (x, y) \in_f f \wedge \neg (x, y) \in_f g$
 by (transfer, simp add: pfun-member-minus)

lemma *ffun-app-upd-1* [simp]: $x = y \implies (f(x \mapsto v)_f)(y)_f = v$

by (transfer, simp)

lemma *ffun-app-upd-2* [simp]: $x \neq y \implies (f(x \mapsto v)_f)(y)_f = f(y)_f$

by (transfer, simp)

lemma *ffun-upd-ext* [simp]: $x \in \text{fdom}(f) \implies f(x \mapsto f(x)_f)_f = f$

by (transfer, simp)

lemma *ffun-app-add* [simp]: $x \in \text{fdom}(g) \implies (f + g)(x)_f = g(x)_f$
by (*transfer*, *simp*)

lemma *ffun-upd-add* [simp]: $f + g(x \mapsto v)_f = (f + g)(x \mapsto v)_f$
by (*transfer*, *simp*)

lemma *ffun-upd-twice* [simp]: $f(x \mapsto u, x \mapsto v)_f = f(x \mapsto v)_f$
by (*transfer*, *simp*)

lemma *ffun-upd-comm*:
assumes $x \neq y$
shows $f(y \mapsto u, x \mapsto v)_f = f(x \mapsto v, y \mapsto u)_f$
using *assms* **by** (*transfer*, *simp* *add*: *pfun-upd-comm*)

lemma *ffun-upd-comm-linorder* [simp]:
fixes $x\ y :: 'a :: \text{linorder}$
assumes $x < y$
shows $f(y \mapsto u, x \mapsto v)_f = f(x \mapsto v, y \mapsto u)_f$
using *assms* **by** (*transfer*, *auto*)

lemma *ffun-app-minus* [simp]: $x \notin \text{fdom } g \implies (f - g)(x)_f = f(x)_f$
by (*transfer*, *auto*)

lemma *ffun-upd-minus* [simp]:
 $x \notin \text{fdom } g \implies (f - g)(x \mapsto v)_f = (f(x \mapsto v)_f - g)$
by (*transfer*, *auto*)

lemma *fdom-member-minus-iff* [simp]:
 $x \notin \text{fdom } g \implies x \in \text{fdom}(f - g) \longleftrightarrow x \in \text{fdom}(f)$
by (*transfer*, *simp*)

lemma *fsubsetq-ffun-upd1* [intro]:
 $\llbracket f \subseteq_f g; x \notin \text{fdom}(g) \rrbracket \implies f \subseteq_f g(x \mapsto v)_f$
by (*transfer*, *auto*)

lemma *fsubsetq-ffun-upd2* [intro]:
 $\llbracket f \subseteq_f g; x \notin \text{fdom}(f) \rrbracket \implies f \subseteq_f g(x \mapsto v)_f$
by (*transfer*, *auto*)

lemma *psubsetq-pfun-upd3* [intro]:
 $\llbracket f \subseteq_f g; g(x)_f = v \rrbracket \implies f \subseteq_f g(x \mapsto v)_f$
by (*transfer*, *auto*)

lemma *fsubsetq-dom-subset*:
 $f \subseteq_f g \implies \text{fdom}(f) \subseteq \text{fdom}(g)$
by (*transfer*, *auto* *simp* *add*: *psubsetq-dom-subset*)

lemma *fsubsetq-ran-subset*:
 $f \subseteq_f g \implies \text{fran}(f) \subseteq \text{fran}(g)$
by (*transfer*, *simp* *add*: *psubsetq-ran-subset*)

10.4 Domain laws

lemma *fdom-finite* [simp]: $\text{finite}(\text{fdom}(f))$
by (*transfer*, *simp*)

lemma *fdom-zero* [*simp*]: $\text{fdom } 0 = \{\}$
by (*transfer*, *simp*)

lemma *fdom-plus* [*simp*]: $\text{fdom } (f + g) = \text{fdom } f \cup \text{fdom } g$
by (*transfer*, *auto*)

lemma *fdom-inter*: $\text{fdom } (f \cap_f g) \subseteq \text{fdom } f \cap \text{fdom } g$
by (*transfer*, *meson pdom-inter*)

lemma *fdom-comp* [*simp*]: $\text{fdom } (g \circ_f f) = \text{fdom } (f \triangleright_f \text{fdom } g)$
by (*transfer*, *auto*)

lemma *fdom-upd* [*simp*]: $\text{fdom } (f(k \mapsto v)_f) = \text{insert } k (\text{fdom } f)$
by (*transfer*, *simp*)

lemma *fdom-fdom-res* [*simp*]: $\text{fdom } (A \triangleleft_f f) = A \cap \text{fdom}(f)$
by (*transfer*, *auto*)

lemma *fdom-graph-ffun* [*simp*]:
 $\text{finite } (\text{Domain } R) \implies \text{fdom } (\text{graph-ffun } R) = \text{Domain } R$
by (*transfer*, *simp add: Domain-fst graph-map-dom*)

10.5 Range laws

lemma *fran-zero* [*simp*]: $\text{fran } 0 = \{\}$
by (*transfer*, *simp*)

lemma *fran-upd* [*simp*]: $\text{fran } (f(k \mapsto v)_f) = \text{insert } v (\text{fran } ((-\{k\}) \triangleleft_f f))$
by (*transfer*, *auto*)

lemma *fran-fran-res* [*simp*]: $\text{fran } (f \triangleright_f A) = \text{fran}(f) \cap A$
by (*transfer*, *auto*)

lemma *fran-comp* [*simp*]: $\text{fran } (g \circ_f f) = \text{fran } (\text{fran } f \triangleleft_f g)$
by (*transfer*, *auto*)

10.6 Domain restriction laws

lemma *fdom-res-zero* [*simp*]: $A \triangleleft_f \{\}_f = \{\}_f$
by (*transfer*, *auto*)

lemma *fdom-res-empty* [*simp*]:
 $(\{\} \triangleleft_f f) = \{\}_f$
by (*transfer*, *auto*)

lemma *fdom-res-fdom* [*simp*]:
 $\text{fdom}(f) \triangleleft_f f = f$
by (*transfer*, *auto*)

lemma *pdom-res-upd-in* [*simp*]:
 $k \in A \implies A \triangleleft_f f(k \mapsto v)_f = (A \triangleleft_f f)(k \mapsto v)_f$
by (*transfer*, *auto*)

lemma *pdom-res-upd-out* [*simp*]:
 $k \notin A \implies A \triangleleft_f f(k \mapsto v)_f = A \triangleleft_f f$

by (transfer, auto)

lemma *fdom-res-override* [simp]: $A \triangleleft_f (f + g) = (A \triangleleft_f f) + (A \triangleleft_f g)$
 by (metis *fdom-res.rep-eq pdom-res-override pfun-of-inject plus-ffun.rep-eq*)

lemma *fdom-res-minus* [simp]: $A \triangleleft_f (f - g) = (A \triangleleft_f f) - g$
 by (transfer, auto)

lemma *fdom-res-swap*: $(A \triangleleft_f f) \triangleright_f B = A \triangleleft_f (f \triangleright_f B)$
 by (transfer, simp add: *pdom-res-swap*)

lemma *fdom-res-twice* [simp]: $A \triangleleft_f (B \triangleleft_f f) = (A \cap B) \triangleleft_f f$
 by (transfer, auto)

lemma *fdom-res-comp* [simp]: $A \triangleleft_f (g \circ_f f) = g \circ_f (A \triangleleft_f f)$
 by (transfer, simp)

10.7 Range restriction laws

lemma *fran-res-zero* [simp]: $\{ \}_f \triangleright_f A = \{ \}_f$
 by (transfer, auto)

lemma *fran-res-upd-1* [simp]: $v \in A \implies f(x \mapsto v)_f \triangleright_f A = (f \triangleright_f A)(x \mapsto v)_f$
 by (transfer, auto)

lemma *fran-res-upd-2* [simp]: $v \notin A \implies f(x \mapsto v)_f \triangleright_f A = ((- \{x\}) \triangleleft_f f) \triangleright_f A$
 by (transfer, auto)

lemma *fran-res-override*: $(f + g) \triangleright_f A \subseteq_f (f \triangleright_f A) + (g \triangleright_f A)$
 by (transfer, simp add: *fran-res-override*)

10.8 Graph laws

lemma *ffun-graph-inv*: $\text{graph-ffun} (\text{ffun-graph } f) = f$
 by (transfer, auto simp add: *pfun-graph-inv finite-Domain*)

lemma *ffun-graph-zero*: $\text{ffun-graph } 0 = \{ \}$
 by (transfer, simp add: *pfun-graph-zero*)

lemma *ffun-graph-minus*: $\text{ffun-graph } (f - g) = \text{ffun-graph } f - \text{ffun-graph } g$
 by (transfer, simp add: *pfun-graph-minus*)

lemma *ffun-graph-inter*: $\text{ffun-graph } (f \cap_f g) = \text{ffun-graph } f \cap \text{ffun-graph } g$
 by (transfer, simp add: *pfun-graph-inter*)

10.9 Partial Function Lens

definition *ffun-lens* :: $'a \Rightarrow ('b \implies ('a, 'b) \text{ffun})$ **where**
[lens-defs]: $\text{ffun-lens } i = \langle \text{lens-get} = \lambda s. s(i)_f, \text{lens-put} = \lambda s v. s(i \mapsto v)_f \rangle$

lemma *ffun-lens-mwb* [simp]: $\text{mwb-lens} (\text{ffun-lens } i)$
 by (unfold-locale, simp-all add: *ffun-lens-def*)

lemma *ffun-lens-src*: $\mathcal{S}_{\text{ffun-lens } i} = \{f. i \in \text{fdom}(f)\}$
 by (auto simp add: *lens-defs lens-source-def, metis ffun-upd-ext*)

Hide implementation details for finite functions

lifting-update *ffun.lifting*

lifting-forget *ffun.lifting*

end

11 Infinity Supplement

theory *Infinity*

imports *HOL.Real*

HOL-Library.Infinite-Set

Optics.Two

begin

This theory introduces a type class *infinite* that guarantees that the underlying universe of the type is infinite. It also provides useful theorems to prove infinity of the universes for various HOL types.

11.1 Type class *infinite*

The type class postulates that the universe (carrier) of a type is infinite.

class *infinite* =

assumes *infinite-UNIV* [*simp*]: *infinite* (*UNIV* :: 'a set)

11.2 Infinity Theorems

Useful theorems to prove that a type's *UNIV* is infinite.

Note that *infinite-UNIV-nat* is already a simplification rule by default.

lemmas *infinite-UNIV-int* [*simp*]

theorem *infinite-UNIV-real* [*simp*]:

infinite (*UNIV* :: real set)

by (*rule infinite-UNIV-char-0*)

theorem *infinite-UNIV-fun1* [*simp*]:

infinite (*UNIV* :: 'a set) \implies

card (*UNIV* :: 'b set) \neq *Suc 0* \implies

infinite (*UNIV* :: ('a \Rightarrow 'b) set)

apply (*erule contrapos-nn*)

apply (*erule finite-fun-UNIVD1*)

apply (*assumption*)

done

theorem *infinite-UNIV-fun2* [*simp*]:

infinite (*UNIV* :: 'b set) \implies

infinite (*UNIV* :: ('a \Rightarrow 'b) set)

apply (*erule contrapos-nn*)

apply (*erule finite-fun-UNIVD2*)

done

theorem *infinite-UNIV-set* [*simp*]:

infinite (*UNIV* :: 'a set) \implies


```

infinite (UNIV :: 'a set set)
apply (erule contrapos-nn)
apply (simp add: Finite-Set.finite-set)
done

```

```

theorem infinite-UNIV-prod1 [simp]:
infinite (UNIV :: 'a set)  $\implies$ 
infinite (UNIV :: ('a  $\times$  'b) set)
apply (erule contrapos-nn)
apply (simp add: finite-prod)
done

```

```

theorem infinite-UNIV-prod2 [simp]:
infinite (UNIV :: 'b set)  $\implies$ 
infinite (UNIV :: ('a  $\times$  'b) set)
apply (erule contrapos-nn)
apply (simp add: finite-prod)
done

```

```

theorem infinite-UNIV-sum1 [simp]:
infinite (UNIV :: 'a set)  $\implies$ 
infinite (UNIV :: ('a + 'b) set)
apply (erule contrapos-nn)
apply (simp)
done

```

```

theorem infinite-UNIV-sum2 [simp]:
infinite (UNIV :: 'b set)  $\implies$ 
infinite (UNIV :: ('a + 'b) set)
apply (erule contrapos-nn)
apply (simp)
done

```

```

theorem infinite-UNIV-list [simp]:
infinite (UNIV :: 'a list set)
apply (rule infinite-UNIV-listI)
done

```

```

theorem infinite-UNIV-option [simp]:
infinite (UNIV :: 'a set)  $\implies$ 
infinite (UNIV :: 'a option set)
apply (erule contrapos-nn)
apply (simp)
done

```

```

theorem infinite-image [intro]:
infinite A  $\implies$  inj-on f A  $\implies$  infinite (f ` A)
apply (metis finite-imageD)
done

```

```

theorem infinite-transfer :
infinite B  $\implies$  B  $\subseteq$  f ` A  $\implies$  infinite A
using infinite-super
apply (blast)
done

```

11.3 Instantiations

The instantiations for product and sum types have stronger caveats than in principle needed. Namely, it would be sufficient for one type of a product or sum to be infinite. A corresponding rule, however, cannot be formulated using type classes. Generally, classes are not entirely adequate for the purpose of deriving the infinity of HOL types, which is perhaps why a class such as *infinite* was omitted from the Isabelle/HOL library.

```
instance nat :: infinite by (intro-classes, simp)
instance int :: infinite by (intro-classes, simp)
instance real :: infinite by (intro-classes, simp)
instance fun :: (type, infinite) infinite by (intro-classes, simp)
instance set :: (infinite) infinite by (intro-classes, simp)
instance prod :: (infinite, infinite) infinite by (intro-classes, simp)
instance sum :: (infinite, infinite) infinite by (intro-classes, simp)
instance list :: (type) infinite by (intro-classes, simp)
instance option :: (infinite) infinite by (intro-classes, simp)

subclass (in infinite) two by (intro-classes, auto)

end
```

12 Positive Subtypes

```
theory Positive
imports
  Infinity
  HOL-Library.Countable
begin
```

12.1 Type Definition

```
typedef (overloaded) 'a::{zero, linorder} pos = {x::'a. x ≥ 0}
  apply (rule-tac x = 0 in exI)
  apply (clarsimp)
  done
```

```
syntax
  -type-pos :: type ⇒ type (-+ [999] 999)
```

```
translations
  (type) 'a+ == (type) 'a pos
```

```
setup-lifting type-definition-pos
```

```
type-synonym preal = real pos
```

12.2 Operators

```
lift-definition mk-pos :: 'a::{zero, linorder} ⇒ 'a pos is
λ n. if (n ≥ 0) then n else 0 by auto
```

```
lift-definition real-of-pos :: real pos ⇒ real is id .
```

```
declare [[coercion real-of-pos]]
```

12.3 Instantiations

instantiation *pos* :: ({*zero*, *linorder*}) *zero*

begin

lift-definition *zero-pos* :: '*a pos*

 is 0 :: '*a* ..

instance ..

end

instantiation *pos* :: ({*zero*, *linorder*}) *linorder*

begin

lift-definition *less-eq-pos* :: '*a pos* \Rightarrow '*a pos* \Rightarrow *bool*

 is (\leq) :: '*a* \Rightarrow '*a* \Rightarrow *bool* .

lift-definition *less-pos* :: '*a pos* \Rightarrow '*a pos* \Rightarrow *bool*

 is ($<$) :: '*a* \Rightarrow '*a* \Rightarrow *bool* .

instance

apply (*intro-classes*; *transfer*)

apply (*auto*)

done

end

instance *pos* :: ({*zero*, *linorder*, *no-top*}) *no-top*

apply (*intro-classes*)

apply (*transfer*)

apply (*clarsimp*)

apply (*meson gt-ex less-imp-le order.strict-trans1*)

done

instance *pos* :: ({*zero*, *linorder*, *no-top*}) *infinite*

apply (*intro-classes*)

apply (*rule notI*)

apply (*subgoal-tac* $\forall x::'a pos. x \leq \text{Max UNIV}$)

using *gt-ex leD* **apply** (*blast*)

apply (*simp*)

done

instantiation *pos* :: (*linordered-semidom*) *linordered-semidom*

begin

lift-definition *one-pos* :: '*a pos*

 is 1 :: '*a* **by** (*simp*)

lift-definition *plus-pos* :: '*a pos* \Rightarrow '*a pos* \Rightarrow '*a pos*

 is (+) **by** (*simp*)

lift-definition *minus-pos* :: '*a pos* \Rightarrow '*a pos* \Rightarrow '*a pos*

 is $\lambda x y. \text{if } y \leq x \text{ then } x - y \text{ else } 0$

by (*simp add: add-le-imp-le-diff*)

lift-definition *times-pos* :: '*a pos* \Rightarrow '*a pos* \Rightarrow '*a pos*

 is *times* **by** (*simp*)

instance

apply (*intro-classes*; *transfer*; *simp?*)

apply (*simp add: add.assoc*)

apply (*simp add: add commute*)

apply (*safe*; *clarsimp?*) [1]

apply (*simp add: diff-diff-add*)

apply (*metis add-le-cancel-left le-add-diff-inverse*)

apply (*simp add: add commute add-le-imp-le-diff*)

apply (*metis add-increasing2 antisym linear*)

```

    apply (simp add: mult.assoc)
    apply (simp add: mult.commute)
    apply (simp add: comm-semiring-class.distrib)
    apply (simp add: mult-strict-left-mono)
    apply (safe; clarsimp?) [1]
    apply (simp add: right-diff-distrib')
    apply (simp add: mult-left-mono)
    using mult-left-le-imp-le apply (fastforce)
    apply (simp add: distrib-left)
  done
end

instantiation pos :: (linordered-field) semidom-divide
begin
  lift-definition divide-pos :: 'a pos  $\Rightarrow$  'a pos  $\Rightarrow$  'a pos
    is divide by (simp)
  instance
    apply (intro-classes; transfer)
    apply (simp-all)
  done
end

instantiation pos :: (linordered-field) inverse
begin
  lift-definition inverse-pos :: 'a pos  $\Rightarrow$  'a pos
    is inverse by (simp)
  instance ..
end

lemma pos-positive [simp]:  $0 \leq (x::'a::\{\text{zero}, \text{linorder}\} \text{ pos})$ 
  by (transfer, simp)

```

12.4 Theorems

```

lemma mk-pos-zero [simp]:  $\text{mk-pos } 0 = 0$ 
  by (transfer, simp)

lemma mk-pos-one [simp]:  $\text{mk-pos } 1 = 1$ 
  by (transfer, simp)

lemma mk-pos-leq:
   $\llbracket 0 \leq x; x \leq y \rrbracket \Longrightarrow \text{mk-pos } x \leq \text{mk-pos } y$ 
  by (transfer, auto)

lemma mk-pos-less:
   $\llbracket 0 \leq x; x < y \rrbracket \Longrightarrow \text{mk-pos } x < \text{mk-pos } y$ 
  by (transfer, auto)

lemma real-of-pos [simp]:  $x \geq 0 \Longrightarrow \text{real-of-pos } (\text{mk-pos } x) = x$ 
  by (transfer, simp)

lemma mk-pos-real-of-pos [simp]:  $\text{mk-pos } (\text{real-of-pos } x) = x$ 
  by (transfer, simp)

```

12.5 Transfer to Reals

named-theorems *pos-transfer*

lemma *real-of-pos-0* [*pos-transfer*]:

real-of-pos 0 = 0

by (*transfer*, *auto*)

lemma *real-of-pos-1* [*pos-transfer*]:

real-of-pos 1 = 1

by (*transfer*, *auto*)

lemma *real-op-pos-plus* [*pos-transfer*]:

real-of-pos (x + y) = real-of-pos x + real-of-pos y

by (*transfer*, *simp*)

lemma *real-op-pos-minus* [*pos-transfer*]:

x ≥ y ⇒ real-of-pos (x - y) = real-of-pos x - real-of-pos y

by (*transfer*, *simp*)

lemma *real-op-pos-mult* [*pos-transfer*]:

*real-of-pos (x * y) = real-of-pos x * real-of-pos y*

by (*transfer*, *simp*)

lemma *real-op-pos-div* [*pos-transfer*]:

real-of-pos (x / y) = real-of-pos x / real-of-pos y

by (*transfer*, *simp*)

lemma *real-of-pos-numeral* [*pos-transfer*]:

real-of-pos (numeral n) = numeral n

by (*induct n*, *simp-all only: numeral.simps pos-transfer*)

lemma *real-of-pos-eq-transfer* [*pos-transfer*]:

x = y ⟷ real-of-pos x = real-of-pos y

by (*transfer*, *auto*)

lemma *real-of-pos-less-eq-transfer* [*pos-transfer*]:

x ≤ y ⟷ real-of-pos x ≤ real-of-pos y

by (*transfer*, *auto*)

lemma *real-of-pos-less-transfer* [*pos-transfer*]:

x < y ⟷ real-of-pos x < real-of-pos y

by (*transfer*, *auto*)

end

13 Recall Undeclarations

theory *Total-Recall*

imports *Main*

keywords

purge-syntax :: *thy-decl* **and**

purge-notation :: *thy-decl* **and**

recall-syntax :: *thy-decl*

begin

13.1 ML File Import

ML-file *Total-Recall.ML*

13.2 Outer Commands

```
ML <
  val - =
    Outer-Syntax.command @ {command-keyword purge-syntax}
      purge raw syntax clauses
      ((Parse.syntax-mode -- Scan.repeat1 Parse.const-decl) >>
        (Toplevel.theory o (fn (mode, args) =>
          (TotalRecall.record-no-syntax mode args) o
          (Sign.del-syntax-cmd mode args))));

  val - =
    Outer-Syntax.local-theory @ {command-keyword purge-notation}
      purge concrete syntax for constants / fixed variables
      ((Parse.syntax-mode -- Parse.and-list1 (Parse.const -- Parse.mixfix)) >>
        (fn (mode, args) =>
          (Local-Theory.background-theory
            (TotalRecall.record-no-notation mode args)) o
            (Specification.notation-cmd false mode args)));

  val - =
    Outer-Syntax.command @ {command-keyword recall-syntax}
      recall undecarations of all purged items
      (Scan.succeed (Toplevel.theory TotalRecall.execute-all))
>
end
```

14 Meta-theory for UTP Toolkit

```
theory utp-toolkit
  imports
    HOL.Deriv
    HOL-Library.Adhoc-Overloading
    HOL-Library.Char-ord
    HOL-Library.Countable-Set
    HOL-Library.FSet
    HOL-Library.Monad-Syntax
    HOL-Library.Countable
    HOL-Library.Order-Continuity
    HOL-Library.Prefix-Order
    HOL-Library.Product-Order
    HOL-Library.Sublist
    HOL-Algebra.Complete-Lattice
    HOL-Algebra.Galois-Connection
    HOL-Eisbach.Eisbach
    Optics.Lenses
    Countable-Set-Extra
    FSet-Extra
    Relation-Extra
    Map-Extra
    List-Extra
    List-Lexord-Alt
```

Partial-Fun
Finite-Fun
Infinity
Positive
Total-Recall
begin end

References

- [1] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
- [2] A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, 2010.
- [3] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, LNCS 8963, pages 21–41. Springer, 2014.
- [4] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *ICTAC*, LNCS 9965. Springer, 2016.
- [5] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
- [6] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1998.
- [7] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *UTP*, LNCS 10134. Springer, 2016.