

Mathematical Toolkit for Isabelle/UTP

Simon Foster

Pedro Ribeiro

Frank Zeyda

September 6, 2018

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	3
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	4
2.4.3	List Update	7
2.4.4	Drop While and Take While	7
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	13
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	16
3	Infinite Sequences	18
4	Finite Sets: extra functions and properties	24
5	Countable Sets: Extra functions and properties	29
5.1	Extra syntax	29
5.2	Countable set functions	29

6	Map Type: extra functions and properties	35
6.1	Functional Relations	36
6.2	Graphing Maps	36
6.3	Map Application	38
6.4	Map Membership	38
6.5	Preimage	38
6.6	Minus operation for maps	39
6.7	Map Bind	39
6.8	Range Restriction	39
6.9	Map Inverse and Identity	40
6.10	Merging of compatible maps	47
6.11	Conversion between lists and maps	48
6.12	Map Comprehension	49
6.13	Sorted lists from maps	49
6.14	Extra map lemmas	50
7	Alternative List Lexicographic Order	51
8	Partial Functions	51
8.1	Partial function type and operations	51
8.2	Algebraic laws	54
8.3	Lambda abstraction	55
8.4	Membership, application, and update	55
8.5	Domain laws	56
8.6	Range laws	57
8.7	Domain restriction laws	58
8.8	Range restriction laws	59
8.9	Graph laws	59
8.10	Entries	59
8.11	Summation	60
9	Finite Functions	61
9.1	Finite function type and operations	61
9.2	Algebraic laws	63
9.3	Membership, application, and update	63
9.4	Domain laws	65
9.5	Range laws	65
9.6	Domain restriction laws	66
9.7	Range restriction laws	66
9.8	Graph laws	66
10	Infinity Supplement	67
10.1	Type class <i>infinite</i>	67
10.2	Infinity Theorems	67
10.3	Instantiations	69

11 Positive Subtypes	69
11.1 Type Definition	69
11.2 Operators	70
11.3 Instantiations	70
11.4 Theorems	71
11.5 Transfer to Reals	72
12 Recall Undeclarations	73
12.1 ML File Import	73
12.2 Outer Commands	73
13 Meta-theory for UTP Toolkit	73

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
begin
```

2.1 Useful Abbreviations

abbreviation *list-sum* $xs \equiv foldr (+) xs 0$

2.2 Folds

context *abel-semigroup*
begin

lemma *foldr-snoc*: $\text{foldr } (*) (xs @ [x]) k = (\text{foldr } (*) xs k) * x$
by (*induct xs, simp-all add: commute left-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-el :: 'a list \Rightarrow nat \Rightarrow 'a option $(-\langle \cdot \rangle)_l [90, 0] 91)$

where

$\llbracket \langle i \rangle_l = \text{None}$

$\mid (x \# xs) \langle i \rangle_l = (\text{case } i \text{ of } 0 \Rightarrow \text{Some } x \mid \text{Suc } j \Rightarrow xs \langle j \rangle_l)$

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

apply (*induct xs arbitrary: i*)

apply *simp*

apply (*case-tac i*)

apply *simp-all*

done

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

apply (*induct xs arbitrary: i*)

apply *simp*

apply (*case-tac i*)

apply *simp-all*

done

2.4 Extra List Theorems

2.4.1 Map

lemma *map-nth-Cons-atLeastLessThan*:

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

proof –

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

by *auto*

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

by *simp*

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

by *simp*

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

by (*metis Suc-diff-1 le-0-eq length-upt list.simps(8) list.size(3) map-Suc-upt not-less upt-0*)

finally show *?thesis* ..

qed

2.4.2 Sorted Lists

lemma *sorted-last* [*simp*]: $\llbracket x \in \text{set } xs; \text{sorted } xs \rrbracket \Longrightarrow x \leq \text{last } xs$

by (*induct xs, auto*)

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: $(\forall i < \text{length } xs - \text{Suc } 0. xs ! i < xs ! \text{Suc } i)$
 using less-diff-conv by (auto simp add: *is-sorted-list-of-set-def*)
 with hyps(1) **have** srtD: 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 <i>is-sorted-list-of-set A (x # xs)</i> *is-sorted-list-of-set-def*)
 then **show** ?thesis
 by (metis <i>distinct xs</i> 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*:

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

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

finite A \implies *is-sorted-list-of-set* A (*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*:

finite A \implies *sorted-list-of-set*(A) = (THE xs. *sorted*(xs) \wedge *distinct*(xs) \wedge *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]*:

finite A \implies *sorted-list-of-set-alt*(A) = *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 rel \Rightarrow 'a set \Rightarrow 'a list \Rightarrow bool **where**

is-sorted-list-of-set-by R A xs = ((\forall i < length(xs) - 1. (xs[i], xs[i + 1]) \in R) \wedge *set*(xs) = A)

definition *sorted-list-of-set-by* :: 'a rel \Rightarrow 'a set \Rightarrow 'a list **where**

sorted-list-of-set-by R A = (THE xs. *is-sorted-list-of-set-by* R A xs)

definition *fin-set-lexord* :: 'a rel \Rightarrow 'a set rel **where**

fin-set-lexord R = {(A, B). *finite* A \wedge *finite* B \wedge

(\exists xs ys. *is-sorted-list-of-set-by* R A xs \wedge *is-sorted-list-of-set-by* R B ys
 \wedge (xs, ys) \in *lexord* R)}

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

[R \subseteq S; *is-sorted-list-of-set-by* R A xs] \implies *is-sorted-list-of-set-by* S A xs

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

lemma *lexord-mono'*:

[(\bigwedge x y. f x y \longrightarrow g x y); (xs, ys) \in *lexord* {(x, y). f x y}] \implies (xs, ys) \in *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) \implies (xs, ys) \in *fin-set-lexord* {(x, y). f x y} \longrightarrow (xs, ys) \in *fin-set-lexord* {(x, y). g x y}

proof

assume

fin: (xs, ys) \in *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 lists A. distinct(xs)}

lemma *tl-element*:

```

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

```

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*:

```

  ⌊ sorted I; x ∈ set I; x < n ⌋ ⇒ x ∈ set (takeWhile (λx. 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*: $\llbracket \text{sorted } xs; i \geq \text{length } (\text{takeWhile } (\lambda x. x \leq n) \text{ } xs); i < \text{length } xs \rrbracket \Rightarrow n \leq xs ! i$

```

  apply (induct xs arbitrary: i, auto)
  apply (rename-tac x xs i)
  apply (case-tac x ≤ 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*:

```

  ⌊ a ∈ set xs; ¬ P a ⌋ ⇒ length (takeWhile P xs) < 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*:

```

  ⌊ sorted xs; distinct xs; (∃ a ∈ set xs. a ≥ n) ⌋ ⇒ xs ! length (takeWhile (λx. x < n) xs) ≥ 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 $\text{last } (s @ t) = \text{last } s$
by (*metis append-Nil2 assms length-0-conv*)

lemma *length-gt-zero-last-concat*:
assumes $\text{length } t > 0$
shows $\text{last } (s @ t) = \text{last } t$
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 $\text{prefix } (x \# xs) \text{ } ys$
obtains ys' **where** $ys = x \# ys'$ $\text{prefix } xs \text{ } ys'$
using *assms*
by (*metis append-Cons prefix-def*)

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

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

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

lemma *prefix-drop*:
 $\llbracket \text{drop } (\text{length } xs) \text{ } ys = zs; \text{prefix } xs \text{ } ys \rrbracket$
 $\implies ys = xs @ zs$
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 *Sublist.prefixE Sublist.prefixI* **by** *blast*

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* $\longleftrightarrow (\text{butlast } s) = (\text{butlast } t)$
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 $\text{length } s > 0$ $\text{length } z > 0$
 $\text{length } s = \text{length } z$ *strict-prefix* (*butlast* *s*) *t* *strict-prefix* (*butlast* *z*) *t*
shows $(\text{butlast } s) = (\text{butlast } z)$
using *assms* **by** (*auto simp add:strict-prefix-eq-exists*)

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

lemma *prefix-imp-length-not-gt*:

```

assumes prefix s t
shows  $\neg \text{length } t < \text{length } s$ 
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  $\text{length } s > 0$  strict-prefix (butlast s) t
shows  $\neg (\text{length } t < \text{length } s)$ 
using assms prefix-length-less by fastforce

lemma length-not-gt-iff-eq-length:
assumes  $\text{length } s > 0$  and strict-prefix (butlast s) t
shows  $(\neg (\text{length } s < \text{length } t)) = (\text{length } s = \text{length } t)$ 
proof -
have  $(\neg (\text{length } s < \text{length } t)) = ((\text{length } t < \text{length } s) \vee (\text{length } s = \text{length } t))$ 
by (metis not-less-iff-gr-or-eq)
also have  $\dots = (\text{length } s = \text{length } t)$ 
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  $(xs_1 @ ys_1, xs_2 @ ys_2) \in \text{lexord } R$   $\text{length}(xs_1) = \text{length}(xs_2)$ 
  shows  $(xs_1, xs_2) \in \text{lexord } R \vee (xs_1 = xs_2 \wedge (ys_1, ys_2) \in \text{lexord } R)$ 
using assms
proof (induct xs_2 arbitrary: xs_1)
  case (Cons x2 xs2') note hyps = this
  from hyps(3) obtain x1 xs1' where xs1:  $xs_1 = x_1 \# xs_1'$   $\text{length}(xs_1') = \text{length}(xs_2')$ 
    by (auto, metis Suc-length-conv)
  with hyps(2) have xcases:  $(x_1, x_2) \in R \vee (xs_1' @ ys_1, xs_2' @ ys_2) \in \text{lexord } R$ 
    by (auto)
  show ?case
  proof (cases  $(x_1, x_2) \in R$ )
    case True with xs1 show ?thesis
      by (auto)
  next
    case False
    with xcases have  $(xs_1' @ ys_1, xs_2' @ ys_2) \in \text{lexord } R$ 
      by (auto)
    with hyps(1) xs1 have dichot:  $(xs_1', xs_2') \in \text{lexord } R \vee (xs_1' = xs_2' \wedge (ys_1, ys_2) \in \text{lexord } R)$ 
      by (auto)
    have  $x_1 = x_2$ 
      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  $(xs, ys) \in \text{lexord } R$   $\text{length } xs = \text{length } ys$ 

```

```

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' = length xs
by (metis Suc-length-conv)
show ?case
proof (cases (x, y) ∈ 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') ∈ lexord R
by auto
with hyps(1,3) ys obtain i where  $(xs!i, ys!i) \in R \wedge i < \text{length } xs \wedge (\forall j < i. xs!j = ys!j)$ 
by force
with xy ys show ?thesis
apply (rule-tac x=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  $\text{length } xs > i \text{ length } ys > i \wedge (xs!i, ys!i) \in R \wedge \forall j < i. xs!j = ys!j$ 
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  $xs = x' \# xs' \wedge ys = y' \# ys'$ 
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* $\hat{\ } 100$) **where**
dist-concat ls1 ls2 = (uncurry (@) ' (ls1 \times ls2))

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

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

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

2.6 List Domain and Range

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

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

2.7 Extracting List Elements

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

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

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

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

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

lemma *seq-extract-split*:
 assumes $i \leq \text{length } xs$
 shows $\{0..<i\} \upharpoonright_l xs @ \{i..<\text{length } xs\} \upharpoonright_l xs = xs$
 using assms
proof (induct 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 nths-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 nths-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 nths-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-nths*)

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 $m < n \wedge n \leq \text{length } xs$

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 upt-eq-Cons-conv* **by** *blast*

moreover have $\text{map } (\text{nth } (x \# xs)) [m..<n] = \text{map } (\text{nth } xs) [m..<n-1]$

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 $xs: xs = ys @ zs$
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=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**
 $\text{seq-filter } xs \ A = \text{filter } (\lambda x. x \in A) \ xs$

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*]: $xs \upharpoonright_l \text{UNIV} = xs$
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 $length\ s < length\ t$ **and** *strict-prefix (butlast s) t* **and** $length\ s > 0$
shows $length\ (tl\ (t - (butlast\ s))) > 0$
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 *length t = length s and strict-prefix (butlast s) t*

shows $t - (\text{butlast } s) = [\text{last } t]$

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 *length s > 0 strict-prefix (butlast s) t length s < length t*

shows $\text{last } (tl \ (t - (\text{butlast } s))) = (\text{last } t)$

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 length s > 0 and length t > length s*

shows $tl \ (t - (\text{butlast } s)) \neq []$

using *assms length-tl-list-minus-butlast-gt-zero by fastforce*

lemma *tl-list-minus-butlast-empty*:

assumes *strict-prefix (butlast s) t and length s > 0 and length t = length s*

shows $tl \ (t - (\text{butlast } s)) = []$

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 $(t @ s) - (t @ (\text{butlast } u)) = s - (\text{butlast } u)$

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 length s = length t*

shows $tl \ (t - (\text{butlast } s)) = []$

using *assms by (metis list.sel*(3) *list-minus-butlast-eq-butlast-list)*

lemma *prefix-length-tl-minus*:

assumes *strict-prefix s t*

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

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)*

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

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

lemma nth-sinit:
   $i < n \implies \text{sinit } n \text{ xs } ! i = \text{xs } !_s i$ 
  apply (auto simp add: ssubstr-def)
  apply (transfer, auto)
  done

lemma sinit-append-split:
  assumes  $i < j$ 
  shows  $\text{sinit } j \text{ xs} = \text{sinit } i \text{ xs} @ \text{ssubstr } i j \text{ xs}$ 
proof -
  have  $[0.. $i$ ] @ [i.. $j$ ] = [0.. $j$ ]$ 
  by (metis asms 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-asym-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-asym-lemma2*:
 assumes *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 j* *rule*: *linorder-cases*)
 case *less* with *assms* **show** ?thesis
 by (*auto* *dest*: *sinit-linear-asym-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-asym-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* :: *nat*
 obtain *i* :: *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* :: $'a \text{ rel} \implies ('a \text{ seq}) \text{ rel}$ where
 $\text{seq-lexord } R = \{(xs, ys). (\exists i. (\text{sinit } i \text{ } xs, \text{sinit } i \text{ } ys) \in \text{lexord } R)\}$

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*:
 $\text{irrefl } R \implies \text{irrefl } (\text{seq-lexord } R)$
 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: (sinit m xs, sinit m ys) ∈ lexord R (sinit n ys, sinit n zs) ∈ lexord R
  hence inz: m > 0
  using gr0I by force
  from las(1) obtain i where sinitm: (sinit m xs!i, sinit m ys!i) ∈ R i < m ∨ j < i. sinit m xs!j =
sinit m ys!j
  using lexord-eq-length by force
  from las(2) obtain j where sinitn: (sinit n ys!j, sinit n zs!j) ∈ R j < n ∨ k < j. sinit n ys!k = sinit
n zs!k
  using lexord-eq-length by force
  show  $\exists i. (sinit\ i\ xs, sinit\ i\ zs) \in lexord\ R$ 
  proof (cases i ≤ j)
    case True note lt = this
    with sinitm sinitn have  $(sinit\ n\ xs!i, sinit\ n\ zs!i) \in R$ 
    by (metis assms le-eq-less-or-eq le-less-trans nth-sinit transD)
    moreover from lt sinitm sinitn have  $\forall j < i. sinit\ m\ xs!j = sinit\ m\ zs!j$ 
    by (metis less-le-trans less-trans nth-sinit)
    ultimately have  $(sinit\ n\ xs, sinit\ n\ zs) \in lexord\ R$  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  $(sinit\ n\ xs!j, sinit\ n\ zs!j) \in R$ 
  by (metis less-trans nth-sinit)
  moreover from ge sinitm sinitn have  $\forall k < j. sinit\ m\ xs!k = sinit\ m\ zs!k$ 
  by (metis dual-order.strict-trans nth-sinit)
  ultimately have  $(sinit\ n\ xs, sinit\ n\ zs) \in lexord\ R$  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:*

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

lemma *seq-lexord-antisym:*

$\llbracket asym\ R; (a, b) \in seq-lexord\ R \rrbracket \implies (b, a) \notin seq-lexord\ R$
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  $(x,y) \in \text{seq-lexord } R \vee x = y \vee (y,x) \in \text{seq-lexord } R$ 

```

```

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  $\longleftrightarrow (xs, ys) \in \text{seq-lexord } \{(xs, ys). xs < ys\}$ 

```

```

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 seq
  show  $xs \leq xs$  by (simp add: less-eq-seq-def)
next
  fix xs ys zs :: 'a 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 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 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 seq
  have (xs, ys)  $\in$  seq-lexord  $\{(u, v). u < v\} \vee$  xs = ys  $\vee$  (ys, xs)  $\in$  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]:
  ( $\bigwedge x y. f x y \longrightarrow g x y$ )  $\implies$  (xs, ys)  $\in$  seq-lexord  $\{(x, y). f x y\} \longrightarrow$  (xs, ys)  $\in$  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 # insort-rel R x ys)

inductive sorted-rel :: 'a rel  $\Rightarrow$  'a list  $\Rightarrow$  bool where
Nil-rel [iff]: sorted-rel R [] |
Cons-rel:  $\forall y \in \text{set } xs. (x = y \vee (x, y) \in R) \implies \text{sorted-rel } R xs \implies \text{sorted-rel } R (x \# xs)$ 

definition list-of-set :: 'a rel  $\Rightarrow$  'a set  $\Rightarrow$  'a list where
list-of-set R = folding.F (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 ( $\langle \cdot \rangle_f$ )
notation fminus (infixl  $-_f$  65)

```

```

syntax
  -FinFset :: args => 'a fset    ( $\{\cdot\}$ )

```

```

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

```

```

term fBall

```

```

syntax
  -fBall :: pttrn => 'a fset => bool => bool (( $\exists \forall$   $-| \in | \cdot / -$ )  $[0, 0, 10]$  10)
  -fBex  :: pttrn => 'a fset => bool => bool (( $\exists \exists$   $-| \in | \cdot / -$ )  $[0, 0, 10]$  10)

```

```

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

```

```

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  $\{\}$  =  $\{\}$ 
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]:  $\llbracket \text{sorted } xs; \text{distinct } xs \rrbracket \Longrightarrow \text{flist } (\text{finset } xs) = xs$ 
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 < fcard vs  $\Longrightarrow$  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 *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*: $\llbracket x \in \textit{flists } A; x \in \textit{flists } B \rrbracket \Longrightarrow A = B$
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* \subseteq | *t*) then *Abs-fset* ($\bigcup x \in A. \langle x \rangle_f$) else *t*)

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

lemma *finite-UN-subsets*:
 $\llbracket \forall a \in A. B a \subseteq b; \textit{finite } b \rrbracket \Longrightarrow \textit{finite } (\bigcup a \in A. B a)$
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* ($\text{if } (\forall a \in A. a \sqsubseteq t) \text{ then } (\bigcup x \in A. \langle x \rangle_f) \text{ else } \langle t \rangle_f \in \{x. \text{finite } x\}$))
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* ($\text{if } (A = \{\}) \text{ then } \langle t \rangle_f \text{ else } (\bigcap x \in A. \langle x \rangle_f) \in \{x. \text{finite } x\}$))
apply (*metis Abs-fset-inverse fglb-def*)
apply (*auto*)
apply (*metis finite-INT finite-fset*)
done

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

fset (*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]*:

$xs \neq \{\} \implies \langle \bigcap_f xs \rangle_f = (\bigcap x \in \langle xs \rangle_f. \langle x \rangle_f)$
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 \sqsubseteq *FinPow* *xs*
by (*auto simp add: fmember-def*)

lemma *FUnion-FinPow [simp]*:

$\bigcup_f (\text{FinPow } x) = x$
by (*auto simp add: fmember-def less-eq-fset-def*)

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

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

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

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* :: $'a \text{ cset} \Rightarrow 'a \text{ cset} \Rightarrow \text{bool}$ ($(-/ \subseteq_c -)$ [51, 51] 50)
where $A \subseteq_c B \equiv A \leq B$

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

5.2 Countable set functions

setup-lifting *type-definition-cset*

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

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

definition *cBex* :: $'a \text{ cset} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **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 :: $\text{pttrn} \Rightarrow 'a \text{ cset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($((\exists \forall -\in_c - / -)$ [0, 0, 10] 10)
-cBex :: $\text{pttrn} \Rightarrow 'a \text{ cset} \Rightarrow \text{bool} \Rightarrow \text{bool}$ ($((\exists \exists -\in_c - / -)$ [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

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

syntax (*xsymbols*)

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

translations

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

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'*
@{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*:

$\text{countable}(A) \Longrightarrow \text{countable } \{B. B \subseteq A \wedge \text{finite}(B)\}$
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**
 $\lambda A \ f. \text{if } (A = \{\}) \text{ then } \{\} \text{ else } \text{INTER } A \ f$
by (auto)

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

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

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

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

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

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

proof –

fix A

have $\{B :: 'a \text{ cset}. B \subseteq_c A \wedge cfinite B\} = \text{acset } \{B :: 'a \text{ set}. B \subseteq \text{rcset } A \wedge finite B\}$

apply $(\text{auto simp add: } cfinite.rep\text{-eq } cin\text{-def } less\text{-eq-cset-def } countable\text{-finite})$

using $image\text{-iff}$ **apply** $fastforce$

done

moreover have $countable \{B :: 'a \text{ set}. B \subseteq \text{rcset } A \wedge finite B\}$

by $(\text{auto intro: } countable\text{-finite-power})$

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

by $simp$

qed

definition $CCollect :: ('a \Rightarrow \text{bool option}) \Rightarrow 'a \text{ cset option}$ **where**

$CCollect p = (\text{if } (None \notin \text{range } p) \text{ then } Some (cset\text{-Collect } (the \circ p)) \text{ else } None)$

definition $cset\text{-mapM} :: 'a \text{ option} \text{ cset} \Rightarrow 'a \text{ cset option}$ **where**

$cset\text{-mapM } A = (\text{if } (None \in_c A) \text{ then } None \text{ else } Some (the ' _c A))$

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

$cset\text{-mapM } (cimage \text{ Some } A) = \text{Some } A$

apply $(\text{auto simp add: } cset\text{-mapM-def})$

apply $(metis \text{ cimage-cinsert cinsertI1 option.sel set-cinsert})$

done

definition $CCollect\text{-ext} :: ('a \Rightarrow 'b \text{ option}) \Rightarrow ('a \Rightarrow \text{bool option}) \Rightarrow 'b \text{ cset option}$ **where**

$CCollect\text{-ext } f p = \text{do } \{ xs \leftarrow CCollect p; cset\text{-mapM } (f ' _c xs) \}$

lemma $the\text{-Some-image [simp]:}$

$the ' \text{ Some } ' xs = xs$

by $(\text{auto simp add: image-iff})$

lemma $CCollect\text{-ext-Some [simp]:}$

$CCollect\text{-ext } \text{Some } xs = CCollect xs$

apply $(\text{case-tac } CCollect xs)$

apply $(\text{auto simp add: } CCollect\text{-ext-def})$

done

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

lift-definition $fset\text{-cset} :: 'a \text{ fset} \Rightarrow 'a \text{ cset}$ **is** id

using $uncountable\text{-infinite}$ **by** $auto$

definition $cset\text{-count} :: 'a \text{ cset} \Rightarrow 'a \Rightarrow \text{nat}$ **where**

$cset\text{-count } A =$

$(\text{if } (finite (\text{rcset } A)))$

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 (λ A i. if (i \in cset-count A \wedge rcset A) then inv-into (rcset A) (cset-count A) i else (SOME x. x \in_c A)) .

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

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

lemma *inj-nat-seq2set*: *inj nat-seq2set*
proof (rule *injI*, *transfer*)
 fix f g
 assume *prod-encode* $\{(x, f x) \mid x. \text{True}\} = \text{prod-encode } \{(x, g x) \mid x. \text{True}\}$
 hence $\{(x, f x) \mid x. \text{True}\} = \{(x, g x) \mid x. \text{True}\}$
 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 $\lambda A i. \text{if } (i \in A) \text{ then } 1 \text{ else } 0.$

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 ($\lambda A. (\text{bit-seq-of-nat-set } 'c A)$)

lemma *nat-set-cset-collapse-inj*: *inj-on nat-set-cset-collapse* {A. A \neq {}}
proof –
 have $('c) \text{ bit-seq-of-nat-set } \{(A. A \neq \{\}_c) \subseteq \{(A. A \neq \{\}_c)\}$
 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
 $\lambda A. \{B. B \subseteq A \wedge \text{countable } B\}$  by auto

lemma csets-finite: finite A  $\Longrightarrow$  finite (csets A)
  by (auto simp add: csets-def)

lemma csets-infinite: infinite A  $\Longrightarrow$  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 ( $\{A. A \neq \{\}_c\} :: 'a \text{ cset set}$ )
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*:
 $\llbracket \text{inj-on } f \text{ } A; f \text{ ' } A = B \rrbracket \implies (\text{' }_c) f \text{ ' } csets A = csets B$
 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 Map Type: extra functions and properties

```

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

```

6.1 Functional Relations

definition *functional* :: ('a * 'b) set \Rightarrow bool **where**
functional g = inj-on fst g

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

lemma *functional-insert* [simp]: *functional* (insert (x,y) g) \longleftrightarrow (g "{x} \subseteq {y}" \wedge *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*: *functional-list* xs \longleftrightarrow *functional* (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)
done

6.2 Graphing Maps

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

definition *graph-map* :: ('a * 'b) set \Rightarrow ('a \rightarrow 'b) **where**
graph-map g = (λ x. if (x \in fst 'g) then Some (SOME y. (x,y) \in g) else None)

definition *graph-map'* :: ('a \times 'b) set \rightarrow ('a \rightarrow 'b) **where**
graph-map' R = (if (functional R) then Some (graph-map R) else None)

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

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

lemma *dom-map-graph*: $\text{dom } f = \text{Domain}(\text{map-graph } f)$
 by (simp add: map-graph-def dom-def image-def)

lemma *ran-map-graph*: $\text{ran } f = \text{Range}(\text{map-graph } f)$
 by (simp add: map-graph-def ran-def image-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*: $\text{finite } (\text{dom } f) \implies \text{finite } (\text{map-graph } f)$
 by (metis dom-map-graph finite-imageD fst-eq-Domain functional-def map-graph-functional)

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

lemma *graph-map-inv* [simp]: $\text{functional } g \implies \text{map-graph } (\text{graph-map } g) = g$
 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*: $\text{dom } (\text{graph-map } R) = \text{fst} \circ R$
 by (simp add: graph-map-def dom-def)

lemma *graph-map-countable-dom*: $\text{countable } R \implies \text{countable } (\text{dom } (\text{graph-map } R))$
 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]:
 $\text{graph-map}' (\text{map-graph } f) = \text{Some } f$
 by (simp add: graph-map'-def)

lemma *map-graph-inj*:

inj map-graph
by (*metis injI map-graph-inv*)

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

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

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

6.3 Map Application

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

6.4 Map Membership

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

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*:
 $(x, y) \in_m A \longleftrightarrow (x \in \text{dom } A \wedge A(x)_m = y)$
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*)

6.5 Preimage

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

lemma *preimage-range*: $\text{preimage } f (\text{ran } f) = \text{dom } f$
by (*auto simp add: preimage-def ran-def*)

lemma *dom-preimage*: $\text{dom } (m \circ_m f) = \text{preimage } f (\text{dom } m)$
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*:
 $\llbracket \text{countable } A; \text{inj-on } f (\text{preimage } f A) \rrbracket \implies \text{countable } (\text{preimage } f A)$
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

```

6.6 Minus operation for maps

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

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

lemma *map-member-minus*:
 $(x, y) \in_m f -- g \longleftrightarrow (x, y) \in_m f \wedge (\neg (x, y) \in_m g)$
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*: $\text{map-graph } (f -- g) = \text{map-graph } f - \text{map-graph } g$
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)

6.7 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)

6.8 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**
 $\text{ran-restrict-map } f\ B = (\lambda x.\ \text{do } \{ v <- f(x); \text{if } (v \in B) \text{ then } \text{Some}(v) \text{ else } \text{None} \})$

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]: $x \in \text{dom } g \implies (f \mathrel{++} g) x = g x$
 by (auto simp add: map-add-def dom-def)

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

lemma *map-add-restrict*:
 $f \mathrel{++} g = (f \upharpoonright_{(- \text{dom } g)}) \mathrel{++} g$
 by (rule ext, auto simp add: map-add-def restrict-map-def)

6.9 Map Inverse and Identity

definition *map-inv* :: $('a \rightarrow 'b) \Rightarrow ('b \rightarrow 'a)$ **where**
 $\text{map-inv } f \equiv \lambda y. \text{ if } (y \in \text{ran } f) \text{ then } \text{Some } (\text{SOME } x. f x = y) \text{ else } \text{None}$

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

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

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

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

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


```

lemma map-id-on-UNIV[simp]: map-id-on UNIV = Some
  by (simp add:map-id-on-def)

lemma map-id-on-inj [simp]:
  inj-on (map-id-on xs) xs
  by (simp add:inj-on-def)

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

lemma map-inv-id [simp]:
  map-inv (map-id-on xs) = map-id-on xs
  by (force simp add:map-inv-def map-id-on-def ran-def)

lemma map-inv-Some [simp]: map-inv Some = Some
  by (simp add:map-inv-def ran-def)

lemma map-inv-f-f [simp]:
  [| inj-on f (dom f); f x = Some y |] ==> map-inv f y = Some x
  apply (auto simp add:map-inv-def)
  apply (rule some-equality)
  apply (auto simp add:inj-on-def dom-def ran-def)
  done

lemma dom-map-inv [simp]:
  dom (map-inv f) = ran f
  by (auto simp add:map-inv-def)

lemma ran-map-inv [simp]:
  inj-on f (dom f) ==> ran (map-inv f) = dom f
  apply (auto simp add:map-inv-def ran-def)
  apply (rename-tac a b)
  apply (rule-tac x=a in exI)
  apply (force intro:someI)
  apply (rename-tac x y)
  apply (rule-tac x=y 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 ` dom f = Some ` ran f
  by (auto simp add:dom-def ran-def image-def)

lemma inj-map-inv [intro]:
  inj-on f (dom f) ==> inj-on (map-inv f) (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=λ xa. f xa = Some x in some-equality)
  apply (auto)
  apply (metis (mono-tags) option.sel someI)
  done

lemma inj-map-bij: inj-on f (dom f) ==> bij-betw f (dom f) (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 inj-on f (dom f)
  shows map-inv (map-inv f) = f
proof -

  from assms have inj-on (map-inv f) (ran f)
    by auto

  thus ?thesis
  apply (rule-tac ext)
  apply (rename-tac x)
  apply (case-tac  $\exists y. \text{map-inv } f \ y = \text{Some } x$ )
  apply (auto)[1]
  apply (simp add:map-inv-def)
  apply (rename-tac x y)
  apply (case-tac  $y \in \text{ran } f, \text{simp-all}$ )
  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 dom f  $\cap$  ran f = {} inj-on f (dom f)
  shows inj-on (map-inv f ++ f) (dom f  $\cup$  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]:
   $\llbracket \text{dom } f = \text{ran } f; \text{inj-on } f (\text{dom } f) \rrbracket \implies$ 
  bij-betw (Some ++ f) UNIV (range 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:
  bij-betw f a (Some ' b)  $\implies \text{bij-betw } (\text{the } \circ f) a b$ 
  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]:
   $m'(\text{dom } m \cap \text{dom } n) = n'(\text{dom } m \cap \text{dom } n) \implies$ 
   $\text{ran}(m ++ n) = \text{ran } n \cup \text{ran } m$ 
  apply (auto simp add:ran-def)
  apply (metis map-add-find-right)
  apply (rename-tac x a)
  apply (case-tac a  $\in \text{dom } n$ )
  apply (subgoal-tac  $\exists b. n b = \text{Some } x$ )
  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]:
   $\llbracket \text{length } xs = \text{length } ys; \text{distinct } xs \rrbracket \implies \text{ran } [xs \mapsto] ys = \text{set } ys$ 

```

by (induct rule:list-induct2, simp-all)

lemma inj-map-add:

[[inj-on f (dom f); inj-on g (dom g); ran f \cap ran g = {}]] \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 = {} 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 [[x \notin ran g; x \in ran f]] \implies map-inv (f ++ g) x = (map-inv f ++ map-inv g) x

proof –

assume ran:x \notin ran g x \in ran f

with assms **obtain** y **where** dom:f y = Some x y \in dom f y \notin dom g

by (auto simp add:ran-def)

with ran **have** (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 from assms minj **have** [[x \notin ran g; x \notin ran f]] \implies map-inv (f ++ g) x = (map-inv f ++ 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 map-inv (f ++ g) x = (map-inv f ++ map-inv g) x
  apply (case-tac x ∈ ran g)
  apply (simp)
  apply (case-tac x ∈ ran f)
  apply (simp-all)
done
qed

```

```

lemma map-add-lookup [simp]:
  x ∉ dom f ⟹ ([x ↦ y] ++ f) x = Some y
  by (simp add:map-add-def dom-def)

```

```

lemma map-add-Some: Some ++ f = map-id-on (− dom f) ++ f
  apply (rule ext)
  apply (rename-tac x)
  apply (case-tac x ∈ dom f)
  apply (simp-all)
done

```

```

lemma distinct-map-dom:
  x ∉ set xs ⟹ x ∉ dom [xs [↦] ys]
  by (simp add:dom-def)

```

```

lemma distinct-map-ran:
  [| distinct xs; y ∉ set ys; length xs = length ys |] ⟹
  y ∉ ran ([xs [↦] 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]:
  [| length xs = length ys; distinct xs |] ⟹
  ∀ y. [xs [↦] ys] x = Some y ⟶ y ∈ set ys
  by (induct rule:list-induct2, auto)

```

```

lemma maplets-distinct-inj [intro]:
  [| length xs = length ys; distinct xs; distinct ys; set xs ∩ set ys = {} |] ⟹
  inj-on [xs [↦] ys] (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]: map-inv [x ↦ y] = [y ↦ x]
by (auto simp add:map-inv-def)

```

```

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

```

```

lemma maplets-lookup-nth [rule-format, simp]:
   $\llbracket \text{length } xs = \text{length } ys; \text{distinct } xs \rrbracket \implies$ 
   $\forall i < \text{length } ys. [xs \mapsto ys] (xs ! i) = \text{Some } (ys ! i)$ 
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 ∈ ran f)
apply (subgoal-tac y ∈ 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  $\notin$  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 (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]: $(f \circ_m g) x = g(x) >>= f$
by (auto simp add: map-comp-def option.case-eq-if)

6.10 Merging of compatible maps

definition comp-map :: $('a \multimap 'b) \Rightarrow ('a \multimap 'b) \Rightarrow \text{bool}$ (infixl \parallel_m 60) **where**
 $\text{comp-map } f \ g = (\forall x \in \text{dom}(f) \cap \text{dom}(g). \text{the}(f(x)) = \text{the}(g(x)))$

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

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

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

definition *merge* :: $('a \multimap 'b) \text{ set} \Rightarrow 'a \multimap 'b$ **where**
merge *fs* =
 $(\lambda x. \text{if } (\exists f \in fs. x \in \text{dom}(f)) \text{ then } (THE y. \forall f \in fs. x \in \text{dom}(f) \longrightarrow f(x) = y) \text{ else None})$

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

6.11 Conversion between lists and maps

definition *map-of-list* :: $'a \text{ list} \Rightarrow (\text{nat} \multimap 'a)$ **where**
map-of-list *xs* = $(\lambda i. \text{if } (i < \text{length } xs) \text{ then } \text{Some } (xs[i]) \text{ else None})$

lemma *map-of-list-nil* [*simp*]: *map-of-list* \square = *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* :: $(\text{nat} \multimap 'a) \Rightarrow 'a \text{ list}$ **where**
list-of-map *f* = $(\text{if } (f = \text{Map.empty}) \text{ then } \square \text{ else } \text{map } (the \circ f) [0..< \text{Suc}(\text{GREATEST } x. x \in \text{dom } f)])$

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

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

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** $(\text{GREATEST } x. x \in \text{dom } (\text{map-of-list } xs)) = \text{length } xs - 1$

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

6.12 Map Comprehension

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

syntax

$\text{-Mapcompr} :: 'a \Rightarrow 'b \Rightarrow \text{idts} \Rightarrow \text{bool} \Rightarrow 'a \rightarrow 'b \quad ((1[- \mapsto - \mid /-./ -]))$

translations

$\text{-Mapcompr } F \ G \ xs \ P == \text{CONST graph-map } \{(F, G) \mid xs. P\}$

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*]:

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

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

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

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

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*)

6.13 Sorted lists from maps

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

lemma *sorted-list-of-map-empty* [*simp*]:

$\text{sorted-list-of-map } \text{Map.empty} = []$

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

lemma *sorted-list-of-map-inv*:

assumes *finite*($\text{dom}(f)$)

shows $\text{map-of } (\text{sorted-list-of-map } f) = f$

proof –

obtain *A* **where** *finite* *A* $A = \text{dom}(f)$

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]

```

6.14 Extra map lemmas

lemma *map-eqI*:

```

   $\llbracket \text{dom } f = \text{dom } g; \forall x \in \text{dom}(f). \text{the}(f\ x) = \text{the}(g\ x) \rrbracket \implies f = g$ 
  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*: $f \restriction (\neg \text{dom } f) = \text{Map.empty}$

```

  by (metis dom-eq-empty-conv dom-restrict inf-compl-bot)

```

lemma *restrict-map-neg-disj*:

```

   $\text{dom}(f) \cap A = \{\} \implies f \restriction (\neg A) = f$ 
  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  $(f \restriction (\neg \text{dom } h)) = (g \restriction (\neg \text{dom } h))$ 

```

proof –

```

  have  $h \restriction (\neg \text{dom } h) = \text{Map.empty}$ 
  by (metis Compl-disjoint dom-eq-empty-conv dom-restrict)
  then have  $f2: f \restriction (\neg \text{dom } h) = (f ++ h) \restriction (\neg \text{dom } h)$ 
  by (simp add: map-plus-restrict-dist)
  have  $h \restriction (\neg \text{dom } h) = \text{Map.empty}$ 
  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*:

```

   $\text{dom}(f) = A \cup B \implies (f \restriction A) ++ (f \restriction B) = f$ 
  by (rule ext, auto simp add: map-add-def restrict-map-def option.case-eq-if)

```

lemma *map-le-via-restrict*:

```

   $f \subseteq_m g \iff g \restriction \text{dom}(f) = f$ 
  by (auto simp add: map-le-def restrict-map-def dom-def fun-eq-iff)

```

lemma *map-add-cancel*:

```

   $f \subseteq_m g \implies f ++ (g - f) = g$ 
  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*: $f \subseteq_m g \iff (\exists h. \text{dom}(f) \cap \text{dom}(h) = \{\} \wedge f ++ h = g)$

```

apply (auto)
apply (rule-tac  $x=g \dashv\dashv f$  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*: $(\forall k \in \text{dom } m1 \cap \text{dom } m2. m1(k) = m2(k)) \implies m1 ++ m2 = m2 ++ m1$

```

by (auto simp add: map-add-def option.case-eq-if fun-eq-iff)
    (metis IntI domI option.inject)

```

end

7 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* :: $'a::\text{linorder}$ *list* $\Rightarrow 'a$ *list* $\Rightarrow \text{bool}$ (**infix** $<_l$ 50)

where $xs <_l ys \iff (xs, ys) \in \text{lexord } \{(u, v). u < v\}$

lemma *list-lex-less-neq* [*simp*]: $x <_l y \implies x \neq y$

apply (*simp* *add*: *list-lex-less-def*)

apply (*meson* *case-prodD* *less-irrefl* *lexord-irreflexive* *mem-Collect-eq*)

done

lemma *not-less-Nil* [*simp*]: $\neg x <_l []$

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*]: $a \# x <_l b \# y \iff a < b \vee a = b \wedge x <_l y$

by (*simp* *add*: *list-lex-less-def*)

end

8 Partial Functions

theory *Partial-Fun*

imports *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.

8.1 Partial function type and operations

typedef $('a, 'b)$ *pfun* = *UNIV* :: $('a \multimap 'b)$ *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 (**infixl** \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
 $\lambda f g x. \text{if } (x \in \text{dom}(f) \cap \text{dom}(g) \wedge f(x) = g(x)) \text{ then } f(x) \text{ else None} .$ 
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]  $\Rightarrow$  ('a, 'b) pfun (-'(-')p [900,0]900)
-Pfun     :: maplets  $\Rightarrow$  ('a, 'b) pfun ((1{-})p)
-plam     :: pttrn  $\Rightarrow$  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                      $\Rightarrow$  -PfunUpd (CONST pempty) ms
-Pfun (-Maplets ms1 ms2)    <= -PfunUpd (-Pfun ms1) ms2
-Pfun ms                     <= -PfunUpd (CONST pempty) ms
 $\lambda x \mid P . e$                $\Rightarrow$  CONST plambda ( $\lambda x. P$ ) ( $\lambda x. e$ )
 $\lambda x \mid P . e$               <= CONST plambda ( $\lambda x. P$ ) ( $\lambda y. e$ )
 $\lambda y \mid P . e$               <= CONST plambda ( $\lambda x. P$ ) ( $\lambda y. e$ )

```

$\lambda y \mid f v y . e \quad \leq \text{CONST } plambda (f v) (\lambda y . e)$

8.2 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 $f :: ('a, 'b) \text{ pfun}$
shows $f - 0 = f$
by (*transfer*, *simp add: map-minus-def*)

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

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

lemma *pfun-plus-commute*:
 $pdom(f) \cap pdom(g) = \{\} \implies f + g = g + f$
by (*transfer*, *metis map-add-comm*)

lemma *pfun-plus-commute-weak*:
 $(\forall k \in pdom(f) \cap pdom(g). f(k)_p = g(k)_p) \implies f + g = g + f$
by (*transfer*, *simp*, *metis IntD1 IntD2 domD map-add-comm-weak option.sel*)

lemma *pfun-minus-plus-commute*:
 $pdom(g) \cap pdom(h) = \{\} \implies (f - g) + h = (f + h) - g$
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*:

$pdom(f) \cap pdom(g) = \{\} \implies (f + g) - g = f$
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*: $pdom\ x \cap pdom\ y = \{\} \implies x \leq x + y$

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

8.3 Lambda abstraction

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

by (*transfer*, *auto*)

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

by (*transfer*; *auto simp add: domIff*)

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

by (*transfer*, *simp*)

8.4 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*:

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

lemma *pfun-member-plus*:

$(x, y) \in_p f + g \longleftrightarrow ((x \notin pdom(g) \wedge (x, y) \in_p f) \vee (x, y) \in_p g)$
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-app-add* [*simp*]: $x \in pdom(g) \implies (f + g)(x)_p = g(x)_p$

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*]:
 $x \notin \text{pdom } g \implies x \in \text{pdom}(f - g) \iff x \in \text{pdom}(f)$
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*)

8.5 Domain laws

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

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

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

lemma *pdom-minus* [simp]: $g \leq f \implies pdom (f - g) = pdom f - pdom g$
 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*: $pdom (f \cap_p g) \subseteq pdom f \cap pdom g$
 by (transfer, auto simp add: dom-def)

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

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

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

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

lemma *pdom-graph-pfun* [simp]: $pdom (graph-pfun R) = Domain R$
 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)

8.6 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]: $\text{pran } (g \circ_p f) = \text{pran } (\text{pran } f \triangleleft_p g)$
 by (transfer, auto simp add: ran-def restrict-map-def)

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

8.7 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]:
 $\text{pdom}(f) \triangleleft_p f = f$
 by (transfer, auto)

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

lemma *pdom-res-alt-def*: $A \triangleleft_p f = f \circ_p \text{pId-on } A$
 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)

8.8 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*: $f \triangleright_p A = pId\text{-}on\ A \circ_p f$
 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

8.9 Graph laws

lemma *pfun-graph-inv*: $graph\text{-}pfun\ (pfun\text{-}graph\ f) = f$
 by (transfer, simp)

lemma *pfun-graph-zero*: $pfun\text{-}graph\ 0 = \{\}$
 by (transfer, simp add: map-graph-def)

lemma *pfun-graph-pId-on*: $pfun\text{-}graph\ (pId\text{-}on\ A) = Id\text{-}on\ A$
 by (transfer, auto simp add: map-graph-def)

lemma *pfun-graph-minus*: $pfun\text{-}graph\ (f - g) = pfun\text{-}graph\ f - pfun\text{-}graph\ g$
 by (transfer, simp add: map-graph-minus)

lemma *pfun-graph-inter*: $pfun\text{-}graph\ (f \cap_p g) = pfun\text{-}graph\ f \cap pfun\text{-}graph\ g$
 apply (transfer, auto simp add: map-graph-def)
 apply (metis option.discI)+
 done

8.10 Entries

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

lemma *pfun-entries-apply-1* [simp]:
 $x \in d \implies (pfun\text{-}entries\ d\ f)(x)_p = f\ x$
 by (transfer, auto)

lemma *pfun-entries-apply-2* [simp]:
 $x \notin d \implies (pfun\text{-}entries\ d\ f)(x)_p = undefined$
 by (transfer, auto)

8.11 Summation

definition $\text{pfun-sum} :: ('k, 'v :: \text{comm-monoid-add}) \text{pfun} \Rightarrow 'v$ **where**
 $\text{pfun-sum } f = \text{sum } (\text{pfun-app } f) (\text{pdom } f)$

lemma pfun-sum-empty $[\text{simp}]$: $\text{pfun-sum } \{\}_p = 0$
by $(\text{simp add: pfun-sum-def})$

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 $(\text{simp-all add: pfun-sum-def assms,metis add.commute assms(2) pfun-app-upd-2 sum.cong})$

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 $(\text{cases } k \notin \text{pdom}(m))$

case True
then show $?thesis$
by $(\text{simp add: pfun-sum-upd-1 assms})$
next
case False
then show $?thesis$
using $\text{assms pfun-sum-upd-1 [of } ((- \{k\}) \triangleleft_p m) k v]$
by $(\text{simp add: pfun-sum-upd-1})$
qed

lemma $\text{pfun-sum-dom-res-insert}$ $[\text{simp}]$:
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 $\text{assms by (simp add: pfun-sum-def)}$

lemma pfun-sum-pdom-res :
fixes $f :: ('a, 'b :: \text{ab-group-add}) \text{pfun}$
assumes $\text{finite}(\text{pdom } f)$
shows $\text{pfun-sum } (A \triangleleft_p f) = \text{pfun-sum } f - (\text{pfun-sum } ((- A) \triangleleft_p f))$
proof $-$
have $1 : A \cap \text{pdom}(f) = \text{pdom}(f) - (\text{pdom}(f) - A)$
by (auto)
show $?thesis$
apply $(\text{simp add: pfun-sum-def})$
apply $(\text{subst } 1)$
apply (subst sum-diff)
apply $(\text{auto simp add: sum-diff Diff-subset Int-commute boolean-algebra-class.diff-eq assms})$
done
qed

lemma $\text{pfun-sum-pdom-antires}$ $[\text{simp}]$:
fixes $f :: ('a, 'b :: \text{ab-group-add}) \text{pfun}$
assumes $\text{finite}(\text{pdom } f)$
shows $\text{pfun-sum } ((- A) \triangleleft_p f) = \text{pfun-sum } f - \text{pfun-sum } (A \triangleleft_p f)$
by $(\text{subst pfun-sum-pdom-res, simp-all add: assms})$

Hide implementation details for partial functions

lifting-update pfun.lifting
lifting-forget pfun.lifting

end

9 Finite Functions

```
theory Finite-Fun
imports Map-Extra Partial-Fun FSet-Extra
begin
```

9.1 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 ffun-inter :: ('a, 'b) ffun  $\Rightarrow$  ('a, 'b) ffun  $\Rightarrow$  ('a, 'b) ffun (infixl  $\cap_f$  80)
where ffun-inter  $\equiv$  inf

instantiation ffun :: (type, type) order
begin
  lift-definition less-eq-ffun :: ('a, 'b) ffun  $\Rightarrow$  ('a, 'b) ffun  $\Rightarrow$  bool is
     $\lambda f g. f \subseteq_p g$  .
  lift-definition less-ffun :: ('a, 'b) ffun  $\Rightarrow$  ('a, 'b) ffun  $\Rightarrow$  bool is
     $\lambda f g. f < g$  .
instance
  by (intro-classes, (transfer, auto)+)
end

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

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

instance ffun :: (type, type) semilattice-inf
  by (intro-classes, (transfer, auto)+)

lemma ffun-subset-eq-least [simp]:
   $\{\}_f \subseteq_f f$ 
  by (transfer, auto)

syntax
  -FfunUpd :: [('a, 'b) ffun, maplets]  $\Rightarrow$  ('a, 'b) ffun (-'(-)f [900,0]900)
  -Ffun :: maplets  $\Rightarrow$  ('a, 'b) ffun ((1{-}f) )

translations
  -FfunUpd m (-Maplets xy ms) == -FfunUpd (-FfunUpd m xy) ms
  -FfunUpd m (-maplet x y) == CONST ffun-upd m x y
  -Ffun ms  $\Rightarrow$  -FfunUpd (CONST fempty) ms
  -Ffun (-Maplets ms1 ms2)  $\leq$  -FfunUpd (-Ffun ms1) ms2
  -Ffun ms  $\leq$  -FfunUpd (CONST fempty) ms

```

9.2 Algebraic laws

lemma *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 *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 $f :: ('a, 'b)$ *ffun*
 shows $f - 0 = f$
 by (*transfer*, *simp*)

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

lemma *ffun-minus-self* [*simp*]:
 fixes $f :: ('a, 'b)$ *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*)

9.3 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-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 *psubseteq-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 *fsubseteq-dom-subset*:
 $f \subseteq_f g \implies \text{fdom}(f) \subseteq \text{fdom}(g)$
 by (transfer, auto simp add: psubseteq-dom-subset)

lemma *fsubseteq-ran-subset*:
 $f \subseteq_f g \implies \text{fran}(f) \subseteq \text{fran}(g)$
 by (transfer, simp add: psubseteq-ran-subset)

9.4 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)

9.5 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)

9.6 Domain restriction laws

lemma *fdom-res-zero* [simp]: $A \triangleleft_f \{\} = \{\}_f$
 by (transfer, auto)

lemma *fdom-res-empty* [simp]:
 $(\{\} \triangleleft_f f) = \{\}_f$
 by (transfer, auto)

lemma *fdom-res-fdom* [simp]:
 $fdom(f) \triangleleft_f f = f$
 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)

9.7 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: pran-res-override)

9.8 Graph laws

lemma *ffun-graph-inv*: $graph_ffun (ffun_graph f) = f$
 by (transfer, auto simp add: pfun-graph-inv finite-Domain)

lemma *ffun-graph-zero*: $ffun_graph 0 = \{\}$

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)

Hide implementation details for finite functions

lifting-update *ffun.lifting*

lifting-forget *ffun.lifting*

end

10 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.

10.1 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)

10.2 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
 $\text{card } (\text{UNIV} :: 'b \text{ set}) \neq \text{Suc } 0 \implies$
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  $\implies B \subseteq f^{-1} A \implies$  infinite A
  using infinite-super
  apply (blast)
done

```

10.3 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

```

11 Positive Subtypes

```

theory Positive
imports
  Infinity
  HOL-Library.Countable
begin

```

11.1 Type Definition

```

typedef (overloaded) 'a::{zero, linorder} pos = {x::'a. x  $\geq$  0}
  apply (rule-tac x = 0 in exI)
  apply (clarsimp)
done

```

```

syntax
  -type-pos :: type  $\Rightarrow$  type (+ [999] 999)

```

```

translations
  (type) 'a+ == (type) 'a pos

```

```

setup-lifting type-definition-pos

```

```

type-synonym preal = real pos

```

11.2 Operators

lift-definition *mk-pos* :: '*a*::{*zero*, *linorder*} \Rightarrow '*a pos* **is**
 $\lambda n.$ if ($n \geq 0$) then *n* else 0 **by** *auto*

lift-definition *real-of-pos* :: *real pos* \Rightarrow *real* **is** *id* .

declare [[*coercion real-of-pos*]]

11.3 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 $\lambda x y.$ if $y \leq x$ then $x - y$ 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]:  $0 \leq (x::'a::\{zero, linorder\} pos)$ 
  by (transfer, simp)

```

11.4 Theorems

```

lemma mk-pos-zero [simp]:  $mk-pos\ 0 = 0$ 
  by (transfer, simp)

```

```

lemma mk-pos-one [simp]:  $mk-pos\ 1 = 1$ 
  by (transfer, simp)

```

```

lemma mk-pos-leq:
   $\llbracket 0 \leq x; x \leq y \rrbracket \Longrightarrow mk-pos\ x \leq mk-pos\ y$ 
  by (transfer, auto)

```

```

lemma mk-pos-less:
   $\llbracket 0 \leq x; x < y \rrbracket \Longrightarrow mk-pos\ x < mk-pos\ y$ 
  by (transfer, auto)

```

lemma *real-of-pos [simp]*: $x \geq 0 \implies \text{real-of-pos } (\text{mk-pos } x) = x$
by (*transfer, simp*)

lemma *mk-pos-real-of-pos [simp]*: $\text{mk-pos } (\text{real-of-pos } x) = x$
by (*transfer, simp*)

11.5 Transfer to Reals

named-theorems *pos-transfer*

lemma *real-of-pos-0 [pos-transfer]*:
 $\text{real-of-pos } 0 = 0$
by (*transfer, auto*)

lemma *real-of-pos-1 [pos-transfer]*:
 $\text{real-of-pos } 1 = 1$
by (*transfer, auto*)

lemma *real-op-pos-plus [pos-transfer]*:
 $\text{real-of-pos } (x + y) = \text{real-of-pos } x + \text{real-of-pos } y$
by (*transfer, simp*)

lemma *real-op-pos-minus [pos-transfer]*:
 $x \geq y \implies \text{real-of-pos } (x - y) = \text{real-of-pos } x - \text{real-of-pos } y$
by (*transfer, simp*)

lemma *real-op-pos-mult [pos-transfer]*:
 $\text{real-of-pos } (x * y) = \text{real-of-pos } x * \text{real-of-pos } y$
by (*transfer, simp*)

lemma *real-op-pos-div [pos-transfer]*:
 $\text{real-of-pos } (x / y) = \text{real-of-pos } x / \text{real-of-pos } y$
by (*transfer, simp*)

lemma *real-of-pos-numeral [pos-transfer]*:
 $\text{real-of-pos } (\text{numeral } n) = \text{numeral } n$
by (*induct n, simp-all only: numeral.simps pos-transfer*)

lemma *real-of-pos-eq-transfer [pos-transfer]*:
 $x = y \iff \text{real-of-pos } x = \text{real-of-pos } y$
by (*transfer, auto*)

lemma *real-of-pos-less-eq-transfer [pos-transfer]*:
 $x \leq y \iff \text{real-of-pos } x \leq \text{real-of-pos } y$
by (*transfer, auto*)

lemma *real-of-pos-less-transfer [pos-transfer]*:
 $x < y \iff \text{real-of-pos } x < \text{real-of-pos } y$
by (*transfer, auto*)

end

12 Recall Undeclarations

```
theory Total-Recall
imports Main
keywords
  purge-syntax :: thy-decl and
  purge-notation :: thy-decl and
  recall-syntax :: thy-decl
begin
```

12.1 ML File Import

ML-file *Total-Recall.ML*

12.2 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.mifix)) >>
        (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 undeclarations of all purged items
      (Scan.succeed (Toplevel.theory TotalRecall.execute-all))
)
end
```

13 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
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.