# Mathematical Toolkit for Isabelle/UTP

Simon Foster Pedro Ribeiro Frank Zeyda

March 2, 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

This document contains the description of our mathematical toolkit for Isabelle/UTP [2, 4, 5, 8], a mechanisation of Hoare and He's *Unifying Theories of Programming* [6, 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 [7];
- positive subtypes of existing types;
- trace algebras, which underlie generalised reactive processes in UTP [3];
- infinite sequences;
- injection universes;
- 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 Extra Lens Laws

**theory** *Lens-Extra*
  **imports**
    *Optics.Lenses*
    *HOL−Eisbach.Eisbach*
**begin**

**lemma** *comp-weak-lens*: $\llbracket$ *weak-lens* $x$; *weak-lens* $y$ $\rrbracket \Longrightarrow$ *weak-lens* $(x ;_L y)$

**by** (*unfold-locales*, *simp-all add*: *lens-comp-def*)

**lemma** *list-augment-last* [*simp*]:
*list-augment* (*xs* @ [*y*]) (*length xs*) *z* = *xs* @ [*z*]
**by** (*induct xs*, *simp-all*)

**lemma** *lens-get-put-quasi-commute*:
$[\![ \textit{vwb-lens } Y;\ X \subseteq_L Y ]\!] \Longrightarrow \textit{get}_Y\ (\textit{put}_X\ s\ v) = \textit{put}_{X\ /_L\ Y}\ (\textit{get}_Y\ s)\ v$
**proof** −
**assume** *a1*: *vwb-lens Y*
**assume** *a2*: $X \subseteq_L Y$
**have** $\bigwedge l\ la.\ \textit{put}_{l\ ;_L\ la} = (\lambda b\ c.\ \textit{put}_{la}\ (b::'b)\ (\textit{put}_l\ (\textit{get}_{la}\ b::'a)\ (c::'c)))$
**by** (*simp add*: *lens-comp-def*)
**then have** $\bigwedge l\ la\ b\ c.\ \textit{get}_l\ (\textit{put}_{la\ ;_L\ l}\ (b::'b)\ (c::'c)) = \textit{put}_{la}\ (\textit{get}_l\ b::'a)\ c \vee \neg\ \textit{weak-lens}\ l$
**by** *force*
**then show** *?thesis*
**using** *a2 a1* **by** (*metis lens-quotient-comp vwb-lens-wb wb-lens-def*)
**qed**

**lemma** *lens-put-of-quotient*:
$[\![ \textit{vwb-lens } Y;\ X \subseteq_L Y ]\!] \Longrightarrow \textit{put}_Y\ s\ (\textit{put}_{X\ /_L\ Y}\ v_2\ v_1) = \textit{put}_X\ (\textit{put}_Y\ s\ v_2)\ v_1$
**proof** −
**assume** *a1*: *vwb-lens Y*
**assume** *a2*: $X \subseteq_L Y$
**have** *f3*: $\bigwedge l\ b.\ \textit{put}_l\ (b::'b)\ (\textit{get}_l\ b::'a) = b \vee \neg\ \textit{vwb-lens}\ l$
**by** *force*
**have** *f4*: $\bigwedge b\ c.\ \textit{put}_{X\ /_L\ Y}\ (\textit{get}_Y\ b)\ c = \textit{get}_Y\ (\textit{put}_X\ b\ c)$
**using** *a2 a1* **by** (*simp add*: *lens-get-put-quasi-commute*)
**have** $\bigwedge b\ c\ a.\ \textit{put}_Y\ (\textit{put}_X\ b\ c)\ a = \textit{put}_Y\ b\ a$
**using** *a2 a1* **by** (*simp add*: *sublens-put-put*)
**then show** *?thesis*
**using** *f4 f3 a1* **by** (*metis mwb-lens.put-put mwb-lens-def vwb-lens-mwb weak-lens.put-get*)
**qed**

**lemma** *lens-override-idem* [*simp*]:
$\textit{vwb-lens } X \Longrightarrow S \oplus_L S \textit{ on } X = S$
**by** (*simp add*: *lens-override-def*)

**lemma** *lens-override-put-right-in*:
$[\![ \textit{vwb-lens } A;\ X \subseteq_L A ]\!] \Longrightarrow S_1 \oplus_L (\textit{put}_X\ S_2\ v) \textit{ on } A = \textit{put}_X\ (S_1 \oplus_L S_2 \textit{ on } A)\ v$
**by** (*simp add*: *lens-override-def lens-get-put-quasi-commute lens-put-of-quotient*)

**lemma** *lens-override-put-right-out*:
$[\![ \textit{vwb-lens } A;\ X \bowtie A ]\!] \Longrightarrow S_1 \oplus_L (\textit{put}_X\ S_2\ v) \textit{ on } A = (S_1 \oplus_L S_2 \textit{ on } A)$
**by** (*simp add*: *lens-override-def lens-indep.lens-put-irr2*)

**lemma** *bij-lens-intro*: $[\![ \textit{weak-lens } L;\ \bigwedge \sigma\ \varrho.\ \textit{put}_L\ \sigma\ (\textit{get}_L\ \varrho) = \varrho ]\!] \Longrightarrow \textit{bij-lens } L$
**using** *bij-lens.intro bij-lens-axioms.intro* **by** *blast*

## 2.1 Mapper Lenses

**definition** *lmap-lens* ::
$(('\alpha \Rightarrow '\beta) \Rightarrow ('\gamma \Rightarrow '\delta)) \Rightarrow$
$(('\beta \Rightarrow '\alpha) \Rightarrow '\delta \Rightarrow '\gamma) \Rightarrow$
$('\gamma \Rightarrow '\alpha) \Rightarrow$

$('\beta \Longrightarrow '\alpha) \Rightarrow$
$('\delta \Longrightarrow '\gamma)$ **where**
[*lens-defs*]:
*lmap-lens f g h l* = (|
*lens-get* = *f* (*get*$_l$),
*lens-put* = *g o* (*put*$_l$) *o h* |)

The parse translation below yields a heterogeneous mapping lens for any record type. This achieved through the utility function above that constructs a functorial lens. This takes as input a heterogeneous mapping function that lifts a function on a record's extension type to an update on the entire record, and also the record's "more" function. The first input is given twice as it has different polymorphic types, being effectively a type functor construction which are not explicitly supported by HOL. We note that the *more-update* function does something similar to the extension lifting, but is not precisely suitable here since it only considers homogeneous functions, namely of type $'a \Rightarrow 'a$ rather than $'a \Rightarrow 'b$.

**syntax**
*-lmap* :: *id* ⇒ *logic* (*lmap*[-])

```
ML ‹
  fun lmap-tr [Free (name, -)] =
    let
      val extend = Free (name ^ .extend, dummyT);
      val truncate = Free (name ^ .truncate, dummyT);
      val more = Free (name ^ .more, dummyT);
      val map-ext = Abs (f, dummyT,
                    Abs (r, dummyT,
                      extend $ (truncate $ Bound 0) $ (Bound 1 $ (more $ (Bound 0)))))

    in
      Const (@{const-syntax lmap-lens}, dummyT) $ map-ext $ map-ext $ more
    end
  | lmap-tr - = raise Match;
›
```

**parse-translation** ‹[(@{*syntax-const -lmap*}, *K lmap-tr*)]›

### 2.2 Tactic

A simple tactic for simplifying lens expressions

**declare** *split-paired-all* [*alpha-splits*]

**method** *lens-simp* = (*simp add*: *alpha-splits lens-defs prod.case-eq-if*)

**end**

## 3 Lists: extra functions and properties

**theory** *List-Extra*
**imports**
*Main*
*HOL−Library.Sublist*
*HOL−Library.Monad-Syntax*
*HOL−Library.Prefix-Order*
**begin**

## 3.1 Useful Abbreviations

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

## 3.2 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* (-$\langle$-$\rangle_l$ [*90*, *0*] *91*)
**where**
  $[]\langle i\rangle_l$ = *None*
| $(x \# xs)\langle i\rangle_l$ = (*case i of 0* $\Rightarrow$ *Some x* | *Suc j* $\Rightarrow$ *xs* $\langle j\rangle_l$)

**lemma** *nth-el-appendl*[*simp*]: *i* < *length xs* $\Longrightarrow$ $(xs\ @\ ys)\langle i\rangle_l = xs\langle i\rangle_l$
  **apply** (*induct xs arbitrary*: *i*)
   **apply** *simp*
  **apply** (*case-tac i*)
   **apply** *simp-all*
  **done**

**lemma** *nth-el-appendr*[*simp*]: *length xs* $\leq$ *i* $\Longrightarrow$ $(xs\ @\ ys)\langle i\rangle_l = ys\langle i - length\ xs\rangle_l$
  **apply** (*induct xs arbitrary*: *i*)
   **apply** *simp*
  **apply** (*case-tac i*)
   **apply** *simp-all*
  **done**

## 3.3 Extra List Theorems

### 3.3.1 Map

**lemma** *map-nth-Cons-atLeastLessThan*:
  *map* (*nth* (*x* # *xs*)) [*Suc m*..<*n*] = *map* (*nth xs*) [*m*..<*n* − *1*]
**proof** −
  **have** *nth xs* = *nth* (*x* # *xs*) $\circ$ *Suc*
    **by** *auto*
  **hence** *map* (*nth xs*) [*m*..<*n* − *1*] = *map* (*nth* (*x* # *xs*) $\circ$ *Suc*) [*m*..<*n* − *1*]
    **by** *simp*
  **also have** ... = *map* (*nth* (*x* # *xs*)) (*map Suc* [*m*..<*n* − *1*])
    **by** *simp*
  **also have** ... = *map* (*nth* (*x* # *xs*)) [*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**

### 3.3.2 Sorted Lists

**lemma** *sorted-last* [*simp*]: $[\![$ *x* $\in$ *set xs*; *sorted xs* $]\!]$ $\Longrightarrow$ *x* $\leq$ *last xs*
  **apply** (*induct xs*)
   **apply** (*auto*)
   **apply** (*metis last-in-set sorted-Cons*)+
  **done**

**lemma** *sorted-map*: $[\![$ *sorted xs*; *mono f* $]\!]$ $\Longrightarrow$ *sorted* (*map f xs*)
  **by** (*simp add*: *monoD sorted-equals-nth-mono*)

**lemma** *sorted-distinct* [*intro*]: ⟦ *sorted* (*xs*); *distinct*(*xs*) ⟧ ⟹ (∀ *i*<*length xs* − *1*. *xs*!*i* < *xs*!(*i* + *1*))
  **apply** (*induct xs*)
  **apply** (*auto*)
 **apply** (*metis Suc-mono distinct.simps*(*2*) *length-Cons lessI less-SucI less-le nth-Cons-Suc nth-eq-iff-index-eq sorted-equals-nth-mono*)
**done**

Is the given list a permutation of the given set?

**definition** *is-sorted-list-of-set* :: (*'a*::*ord*) *set* ⇒ *'a list* ⇒ *bool* **where**
*is-sorted-list-of-set A xs* = ((∀ *i*<*length*(*xs*) − *1*. *xs*!*i* < *xs*!(*i* + *1*)) ∧ *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*: (∀ *i*<*length xs* − *Suc 0*. *xs* ! *i* < *xs* ! *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 length-pos-if-in-set less-imp-le nth-Cons-0 sorted.simps sorted-many sorted-single*)
  **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** (∀ *n*. ¬ *n* < *length* (*x* # *xs*) − *1* ∨ (*x* # *xs*) ! *n* < (*x* # *xs*) ! (*n* + *1*)) ∧ *set* (*x* # *xs*) = *A*
      **by** (*meson* ‹*is-sorted-list-of-set A* (*x* # *xs*)› *is-sorted-list-of-set-def*)
  **then show** *?thesis*
      **by** (*metis Nat.add-0-right One-nat-def* ‹*distinct xs*› ‹*sorted* (*x* # *xs*)› *add-Suc-right diff-Suc-Suc diff-zero distinct.simps*(*2*) *insert-iff length-pos-if-in-set linorder-not-less list.set*(*2*) *list.simps*(*1*) *list.simps*(*3*) *list.size*(*3*) *list.size*(*4*) *not-less-iff-gr-or-eq nth-Cons-0 nth-Cons-Suc sorted.cases*)
  **qed**
**qed**

**lemma** *is-sorted-list-of-set-alt-def*:
  *is-sorted-list-of-set A xs* ⟷ *sorted* (*xs*) ∧ *distinct* (*xs*) ∧ *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* ⇒ *'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* ⟹ *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* ⟹ *sorted-list-of-set*(*A*) = (*THE xs*. *sorted*(*xs*) ∧ *distinct*(*xs*) ∧ *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* ⟹ *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* ⇒ *'a set* ⇒ *'a list* ⇒ *bool* **where**
*is-sorted-list-of-set-by R A xs* = ((∀ *i*<*length*(*xs*) − *1*. (*xs*!*i*, *xs*!(*i* + *1*)) ∈ *R*) ∧ *set*(*xs*) = *A*)

**definition** *sorted-list-of-set-by* :: *'a rel* ⇒ *'a set* ⇒ *'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* ⇒ *'a set rel* **where**
*fin-set-lexord R* = {(*A*, *B*). *finite A* ∧ *finite B* ∧
(∃ *xs ys*. *is-sorted-list-of-set-by R A xs* ∧ *is-sorted-list-of-set-by R B ys*
∧ (*xs*, *ys*) ∈ *lexord R*)}

**lemma** *is-sorted-list-of-set-by-mono*:
⟦ *R* ⊆ *S*; *is-sorted-list-of-set-by R A xs* ⟧ ⟹ *is-sorted-list-of-set-by S A xs*
**by** (*auto simp add*: *is-sorted-list-of-set-by-def*)

**lemma** *lexord-mono'*:
⟦ (⋀ *x y*. *f x y* ⟶ *g x y*); (*xs*, *ys*) ∈ *lexord* {(*x*, *y*). *f x y*} ⟧ ⟹ (*xs*, *ys*) ∈ *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*]:
(⋀ *x y*. *f x y* ⟶ *g x y*) ⟹ (*xs*, *ys*) ∈ *fin-set-lexord* {(*x*, *y*). *f x y*} ⟶ (*xs*, *ys*) ∈ *fin-set-lexord* {(*x*, *y*). *g x y*}
**proof**
**assume**
*fin*: (*xs*, *ys*) ∈ *fin-set-lexord* {(*x*, *y*). *f x y*} **and**
*hyp*: (⋀ *x y*. *f x y* ⟶ *g x y*)

**from** *fin* **have** *finite xs finite ys*
**using** *fin-set-lexord-def* **by** *fastforce*+

**with** *fin hyp* **show** (*xs*, *ys*) ∈ *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*:
$[\![\ x \in set\ xs;\ x \neq hd(xs)\ ]\!] \Longrightarrow x \in set(tl(xs))$
**by** (*metis in-set-insert insert-Nil list.collapse list.distinct*(*2*) *set-ConsD*)

### 3.3.3 List Update

**lemma** *listsum-update*:
**fixes** *xs* :: $'a$::*ring list*
**assumes** $i < length\ xs$
**shows** *list-sum* $(xs[i := v]) = list\text{-}sum\ xs - xs\ !\ i + v$
**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**

### 3.3.4 Drop While and Take While

**lemma** *dropWhile-sorted-le-above*:
$[\![\ sorted\ xs;\ x \in set\ (dropWhile\ (\lambda\ x.\ x \le n)\ xs)\ ]\!] \Longrightarrow x > n$
**apply** (*induct xs*)
**apply** (*auto*)
**apply** (*rename-tac a xs*)
**apply** (*case-tac* $a \le n$)
**apply** (*simp-all*)
**using** *sorted-Cons* **apply** *blast*
**apply** (*meson dual-order.trans not-less sorted-Cons*)
**done**

**lemma** *set-dropWhile-le*:
$sorted\ xs \Longrightarrow set\ (dropWhile\ (\lambda\ x.\ x \le n)\ xs) = \{x{\in}set\ xs.\ x > n\}$
**apply** (*induct xs*)
**apply** (*simp*)
**apply** (*rename-tac x xs*)
**apply** (*subgoal-tac sorted xs*)

**apply** (*simp*)
**apply** (*safe*)
**apply** (*simp-all*)
**apply** (*meson not-less order-trans sorted-Cons*)
**using** *sorted-Cons* **apply** *auto*
**done**

**lemma** *set-takeWhile-less-sorted*:
$[\![$ *sorted I*; $x \in$ *set I*; $x < n$ $]\!] \Longrightarrow x \in$ *set* (*takeWhile* ($\lambda x.\ x < n$) *I*)
**proof** (*induct I arbitrary*: *x*)
**case** *Nil* **thus** *?case*
**by** (*simp*)
**next**
**case** (*Cons a I*) **thus** *?case*
**by** (*auto*, (*meson le-less-trans sorted-Cons*)+)
**qed**

**lemma** *nth-le-takeWhile-ord*: $[\![$ *sorted xs*; $i \geq$ *length* (*takeWhile* ($\lambda\ x.\ x \leq n$) *xs*); $i <$ *length xs* $]\!] \Longrightarrow$ $n \leq xs\ !\ i$
**apply** (*induct xs arbitrary*: *i*, *auto*)
**apply** (*rename-tac x xs i*)
**apply** (*case-tac* $x \leq n$)
**apply** (*auto simp add*: *sorted-Cons*)
**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*:
$[\![$ $a \in$ *set xs*; $\neg$ *P a* $]\!] \Longrightarrow$ *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*; ($\exists\ a \in$ *set xs*. $a \geq n$) $]\!] \Longrightarrow xs\ !$ *length* (*takeWhile* ($\lambda x.\ x < n$) *xs*) $\geq n$
**apply** (*induct xs*, *auto*)
**using** *sorted-Cons* **apply** *blast*
**done**

### 3.3.5 Last and But Last

**lemma** *length-gt-zero-butlast-concat*:
**assumes** *length ys* $> 0$
**shows** *butlast* (*xs* @ *ys*) = *xs* @ (*butlast ys*)
**using** *assms* **by** (*metis butlast-append length-greater-0-conv*)

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

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

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

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

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

### 3.3.6 Prefixes and Strict Prefixes

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

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

**lemma** *prefix-map-inj*:
$\llbracket$ *inj-on f (set xs* $\cup$ *set ys)*; *prefix (map f xs) (map f ys)* $\rrbracket \Longrightarrow$
*prefix xs 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*]:
*inj-on f (set xs* $\cup$ *set ys)* $\Longrightarrow$
*prefix (map f xs) (map f ys)* $\longleftrightarrow$ *prefix xs ys*
**by** (*metis map-prefixI prefix-map-inj*)

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

**lemma** *strict-prefix-map-inj*:
$\llbracket$ *inj-on f (set xs* $\cup$ *set ys)*; *strict-prefix (map f xs) (map f ys)* $\rrbracket \Longrightarrow$
*strict-prefix xs 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*]:

*inj-on f* (*set xs* ∪ *set ys*) $\Longrightarrow$
*strict-prefix* (*map f xs*) (*map f ys*) $\longleftrightarrow$ *strict-prefix xs ys*
**by** (*metis inj-on-map-eq-map map-prefixI prefix-map-inj prefix-order.less-le*)

**lemma** *prefix-drop*:
$[\![$ *drop* (*length xs*) *ys* = *zs*; *prefix xs ys* $]\!]$
$\Longrightarrow$ *ys* = *xs* @ *zs*
**by** (*metis append-eq-conv-conj prefix-def*)

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

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

**lemma** *prefix-not-empty-length-gt-zero*:
**assumes** *strict-prefix xs ys* **and** *xs* ≠ []
**shows** *length ys* > *0*
**using** *assms prefix-not-empty* **by** *auto*

**lemma** *butlast-prefix-suffix-not-empty*:
**assumes** *strict-prefix* (*butlast xs*) *ys*
**shows** *ys* ≠ []
**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*:
*prefix s t* $\longleftrightarrow$ (∃ *xs* . *s* @ *xs* = *t*)
**using** *Sublist.prefixE Sublist.prefixI* **by** *blast*

**lemma** *strict-prefix-eq-exists*:
*strict-prefix s t* $\longleftrightarrow$ (∃ *xs* . *s* @ *xs* = *t* ∧ (*length xs*) > *0*)
**using** *prefix-def strict-prefix-def* **by** *auto*

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

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

**lemma** *prefix-imp-length-lteq*:
**assumes** *prefix s t*
**shows** *length s* ≤ *length t*

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

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

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

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

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

**finally show** *?thesis* **.**
**qed**

Greatest common prefix

**fun** *gcp* :: *'a list* ⇒ *'a list* ⇒ *'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**

### 3.3.7 Lexicographic Order

**lemma** *lexord-append*:
**assumes** $(xs_1 \mathbin{@} ys_1,\ xs_2 \mathbin{@} ys_2) \in lexord\ R$ $length(xs_1) = length(xs_2)$
**shows** $(xs_1,\ xs_2) \in lexord\ R \lor (xs_1 = xs_2 \land (ys_1,\ ys_2) \in lexord\ R)$
**using** *assms*
**proof** (*induct* $xs_2$ *arbitrary*: $xs_1$)
**case** (*Cons* $x_2$ $xs_2'$) **note** *hyps* = *this*
**from** *hyps*(*3*) **obtain** $x_1$ $xs_1'$ **where** $xs_1$: $xs_1 = x_1 \# xs_1'$ $length(xs_1') = length(xs_2')$
**by** (*auto*, *metis Suc-length-conv*)
**with** *hyps*(*2*) **have** *xcases*: $(x_1,\ x_2) \in R \lor (xs_1' \mathbin{@} ys_1,\ xs_2' \mathbin{@} ys_2) \in lexord\ R$
**by** (*auto*)
**show** *?case*
**proof** (*cases* $(x_1,\ x_2) \in R$)
**case** *True* **with** $xs_1$ **show** *?thesis*
**by** (*auto*)
**next**
**case** *False*
**with** *xcases* **have** $(xs_1' \mathbin{@} ys_1,\ xs_2' \mathbin{@} ys_2) \in lexord\ R$
**by** (*auto*)
**with** *hyps*(*1*) $xs_1$ **have** *dichot*: $(xs_1',\ xs_2') \in lexord\ R \lor (xs_1' = xs_2' \land (ys_1,\ ys_2) \in lexord\ R)$
**by** (*auto*)
**have** $x_1 = x_2$
**using** *False hyps*(*2*) $xs_1$(*1*) **by** *auto*
**with** *dichot* $xs_1$ **show** *?thesis*
**by** (*simp*)
**qed**
**next**
**case** *Nil* **thus** *?case*
**by** *auto*
**qed**

**lemma** *strict-prefix-lexord-rel*:
*strict-prefix xs ys* $\Longrightarrow$ (*xs*, *ys*) $\in$ *lexord R*
**by** (*metis Sublist.strict-prefixE' lexord-append-rightI*)

**lemma** *strict-prefix-lexord-left*:
**assumes** *trans R* (*xs*, *ys*) $\in$ *lexord R strict-prefix xs' xs*
**shows** (*xs'*, *ys*) $\in$ *lexord R*
**by** (*metis assms lexord-trans strict-prefix-lexord-rel*)

**lemma** *prefix-lexord-right*:
**assumes** *trans R* (*xs*, *ys*) $\in$ *lexord R strict-prefix ys ys'*
**shows** (*xs*, *ys'*) $\in$ *lexord R*
**by** (*metis assms lexord-trans strict-prefix-lexord-rel*)

**lemma** *lexord-eq-length*:
  **assumes** $(xs,\ ys) \in$ *lexord R length xs* $=$ *length ys*
  **shows** $\exists\ i.\ (xs!i,\ ys!i) \in R \wedge i <$ *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* $(x,\ y) \in R$)
    **case** *True* **with** *ys* **show** *?thesis*
      **by** (*rule-tac x=0* **in** *exI*, *simp*)
  **next**
    **case** *False*
    **with** *ys hyps*(*2*) **have** *xy*: $x = y\ (xs,\ ys') \in$ *lexord R*
      **by** *auto*
    **with** *hyps*(*1*,*3*) *ys* **obtain** *i* **where** $(xs!i,\ ys'!i) \in R\ i <$ *length xs* $(\forall\ j{<}i.\ xs!j = ys'!j)$
      **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** *length xs* $> i$ *length ys* $> i\ (xs!i,\ ys!i) \in R\ \forall\ j{<}i.\ xs!j = ys!j$
  **shows** $(xs,\ ys) \in$ *lexord R*
**using** *assms* **proof** (*induct i arbitrary*: *xs ys*)
  **case** *0* **thus** *?case*
    **by** (*auto*, *metis lexord-cons-cons list.exhaust nth-Cons-0*)
**next**
  **case** (*Suc i*) **note** *hyps* = *this*
  **then obtain** *x' y' xs' ys'* **where** $xs = x'\ \#\ xs'\ ys = y'\ \#\ ys'$
    **by** (*metis Suc-length-conv Suc-lessE*)
  **moreover with** *hyps*(*5*) **have** $\forall j{<}i.\ xs' ! j = ys' ! j$
    **by** (*auto*)
  **ultimately show** *?case* **using** *hyps*
    **by** (*auto*)
**qed**

## 3.4 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 list set* $\Rightarrow$ *'a list set* $\Rightarrow$ *'a list set* (**infixr** $\frown$ *100*) **where**
*dist-concat ls1 ls2* $=$ (*uncurry* (*op* @) ‘ (*ls1* $\times$ *ls2*))

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

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

**lemma** *dist-concat-insert* [*simp*]:
*insert l ls1* $\frown$ *ls2* = ((*op* @ *l*) ‘ ( *ls2*)) ∪ (*ls1* $\frown$ *ls2*)
  **by** (*auto simp add*: *dist-concat-def*)

## 3.5 List Domain and Range

**abbreviation** *seq-dom* :: *'a list* ⇒ *nat set* ($dom_l$) **where**
*seq-dom xs* ≡ {*0..<length xs*}

**abbreviation** *seq-ran* :: *'a list* ⇒ *'a set* ($ran_l$) **where**
*seq-ran xs* ≡ *set xs*

## 3.6 Extracting List Elements

**definition** *seq-extract* :: *nat set* ⇒ *'a list* ⇒ *'a list* (**infix** $\upharpoonleft_l$ *80*) **where**
*seq-extract A xs* = *nths xs A*

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

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

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

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

**lemma** *seq-extract-split*:
  **assumes** *i* ≤ *length xs*
  **shows** {*0..<i*} $\upharpoonleft_l$ *xs* @ {*i..<length xs*} $\upharpoonleft_l$ *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*. *Suc j* < *i*} = {*0..<i* − *1*}
    **by** (*auto*)
  **moreover have** {*j*. *i* ≤ *Suc j* ∧ *j* < *length xs*} = {*i* − *1..<length xs*}
    **by** (*auto*)
  **ultimately show** *?case*
    **using** *hyp* **by** (*force simp add*: *seq-extract-def nths-Cons*)
**qed**

**lemma** *seq-extract-append*:
  $A \upharpoonleft_l$ (*xs* @ *ys*) = ($A \upharpoonleft_l$ *xs*) @ ({*j*. *j* + *length xs* ∈ *A*} $\upharpoonleft_l$ *ys*)
  **by** (*simp add*: *seq-extract-def nths-append*)

**lemma** *seq-extract-range*: $A \upharpoonleft_l xs = (A \cap dom_l(xs)) \upharpoonleft_l xs$

```
  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 ∩ dom_l(xs) = {} ⟹ A ↿_l xs = []
  by (metis seq-extract-def seq-extract-range nths-empty)

lemma seq-extract-length [simp]:
  length (A ↿_l xs) = card (A ∩ dom_l(xs))
proof −
  have {i. i < length(xs) ∧ i ∈ A} = (A ∩ {0..<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} ↿_l (x # xs) = (if (m = 0) then x # ({0..<n−1} ↿_l xs) else {m−1..<n−1} ↿_l xs)
proof −
  have {j. Suc j < n} = {0..<n − Suc 0}
    by (auto)
  moreover have {j. m ≤ Suc j ∧ Suc j < n} = {m − Suc 0..<n − Suc 0}
    by (auto)

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

lemma seq-extract-singleton:
  assumes i < length xs
  shows {i} ↿_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. Suc j = i} = {i − 1})
  apply (auto)
done

lemma seq-extract-as-map:
  assumes m < n n ≤ length xs
  shows {m..<n} ↿_l xs = map (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.prems(1) upt-eq-Cons-conv by blast
  moreover have map (nth (x # xs)) [Suc m..<n] = map (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*:
  $xs = ys\ @\ zs \longleftrightarrow (\exists\ i \le length(xs).\ ys = \{0..<i\} \upharpoonleft_l xs \wedge zs = \{i..<length(xs)\} \upharpoonleft_l xs)$
**proof**
  **assume** *xs*: $xs = ys\ @\ zs$
  **moreover have** $ys = \{0..<length\ ys\} \upharpoonleft_l (ys\ @\ zs)$
    **by** (*simp add*: *seq-extract-append*)
  **moreover have** $zs = \{length\ ys..<length\ ys + length\ zs\} \upharpoonleft_l (ys\ @\ zs)$
  **proof** −
    **have** $\{length\ ys..<length\ ys + length\ zs\} \cap \{0..<length\ ys\} = \{\}$
      **by** *auto*
    **moreover have** *s1*: $\{j.\ j < length\ zs\} = \{0..<length\ zs\}$
      **by** *auto*
    **ultimately show** *?thesis*
      **by** (*simp add*: *seq-extract-append seq-extract-out-of-range*)
  **qed**
  **ultimately show** $(\exists\ i \le length(xs).\ ys = \{0..<i\} \upharpoonleft_l xs \wedge zs = \{i..<length(xs)\} \upharpoonleft_l xs)$
    **by** (*rule-tac x=length ys* **in** *exI*, *auto*)
**next**
  **assume** $\exists\, i \le length\ xs.\ ys = \{0..<i\} \upharpoonleft_l xs \wedge zs = \{i..<length\ xs\} \upharpoonleft_l xs$
  **thus** $xs = ys\ @\ zs$
    **by** (*auto simp add*: *seq-extract-split*)
**qed**

## 3.7 Filtering a list according to a set

**definition** *seq-filter* :: *'a list* ⇒ *'a set* ⇒ *'a list* (**infix** $\restriction_l$ *80*) **where**
*seq-filter xs A* = *filter* ($\lambda\ x.\ x \in A$) *xs*

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

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

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

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

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

## 3.8 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* ≤ *x* ⟹ *length*(*x* − *y*) = *length*(*x*) − *length*(*y*)
  **by** (*simp add*: *less-eq-list-def minus-list-def*)

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* ≠ []
  **using** *assms* **by** (*metis append-Nil2 prefix-concat-minus strict-prefix-def*)

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

**lemma** *butlast-strict-prefix-length-lt-imp-last-tl-minus-butlast-eq-last*:

**assumes** *length s > 0 strict-prefix* (*butlast s*) *t length s < length t*
**shows** *last* (*tl* (*t* − (*butlast s*))) = (*last t*)
**using** *assms* **by** (*metis last-append last-tl length-tl-list-minus-butlast-gt-zero less-numeral-extra*(*3*) *list.size*(*3*) *append-minus strict-prefix-eq-exists*)

**lemma** *tl-list-minus-butlast-not-empty*:
**assumes** *strict-prefix* (*butlast s*) *t* **and** *length s > 0* **and** *length t > length s*
**shows** *tl* (*t* − (*butlast s*)) ≠ []
**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* − (*butlast s*)) = []
**using** *assms* **by** (*simp add*: *list-minus-butlast-eq-butlast-list*)

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

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

**lemma** *prefix-length-tl-minus*:
**assumes** *strict-prefix s t*
**shows** *length* (*tl* (*t*−*s*)) = (*length* (*t*−*s*)) − *1*
**by** (*auto*)

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

**end**

# 4 Infinite Sequences

**theory** *Sequence*
**imports**
*Real*
*List-Extra*
*HOL−Library.Sublist*
*HOL−Library.Nat-Bijection*
**begin**

**typedef** *'a seq* = *UNIV* :: (*nat* ⇒ *'a*) *set*
**by** (*auto*)

**setup-lifting** *type-definition-seq*

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

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

**abbreviation** *sinit* :: *nat* ⇒ *'a seq* ⇒ *'a list* **where**
*sinit i xs* ≡ *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* ≤ *j* ⟹ *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* ≤ *j* ⟹ *prefix* (*sinit i xs*) (*sinit j xs*)
  **by** (*auto intro*: *map-prefixI simp add*: *ssubstr-def*)

**lemma** *sinit-strict-prefix*:
  *i* < *j* ⟹ *strict-prefix* (*sinit i xs*) (*sinit j xs*)
  **by** (*metis sinit-len sinit-prefix le-less nat-neq-iff prefix-order.dual-order.strict-iff-order*)

**lemma** *nth-sinit*:
  *i* < *n* ⟹ *sinit n xs* ! *i* = *xs* $!_s$ *i*
  **apply** (*auto simp add*: *ssubstr-def*)
  **apply** (*transfer*, *auto*)
  **done**

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

**lemma** *sinit-linear-asym-lemma1*:
  **assumes** *asym R i* < *j* (*sinit i xs*, *sinit i ys*) ∈ *lexord R* (*sinit j ys*, *sinit j xs*) ∈ *lexord R*
  **shows** *False*
**proof** −
  **have** *sinit-xs*: *sinit j xs* = *sinit i xs* @ *ssubstr i j xs*
    **by** (*metis assms*(*2*) *sinit-append-split*)
  **have** *sinit-ys*: *sinit j ys* = *sinit i ys* @ *ssubstr i j ys*
    **by** (*metis assms*(*2*) *sinit-append-split*)
  **from** *sinit-xs sinit-ys assms*(*4*)
  **have** (*sinit i ys*, *sinit i xs*) ∈ *lexord R* ∨ (*sinit i ys* = *sinit i xs* ∧ (*ssubstr i j ys*, *ssubstr i j xs*) ∈ *lexord R*)
    **by** (*auto dest*: *lexord-append*)
  **with** *assms lexord-asymmetric* **show** *False*

  **by** (*force*)
**qed**

**lemma** *sinit-linear-asym-lemma2*:
  **assumes** *asym R* (*sinit i xs*, *sinit i ys*) ∈ *lexord R* (*sinit j ys*, *sinit j xs*) ∈ *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** ∀ *i* :: *nat*. ∀ *x*∈{*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*:
  (∀ *i*. *sinit i xs* = *sinit i ys*) ⟹ *xs* = *ys*
  **by** (*simp add*: *ssubstr-def*, *transfer*, *auto intro*: *range-ext*)

**definition** *seq-lexord* :: *'a rel* ⇒ (*'a seq*) *rel* **where**
*seq-lexord R* = {(*xs*, *ys*). (∃ *i*. (*sinit i xs*, *sinit i ys*) ∈ *lexord R*)}

**lemma** *seq-lexord-irreflexive*:
  ∀ *x*. (*x*, *x*) ∉ *R* ⟹ (*xs*, *xs*) ∉ *seq-lexord R*
  **by** (*auto dest*: *lexord-irreflexive simp add*: *irrefl-def seq-lexord-def*)

**lemma** *seq-lexord-irrefl*:
  *irrefl R* ⟹ *irrefl* (*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** ∃ *i*. (*sinit i xs*, *sinit i zs*) ∈ *lexord R*
**proof** (*cases i* ≤ *j*)
  **case** *True* **note** *lt* = *this*
  **with** *sinitm sinitn* **have** (*sinit n xs*!*i*, *sinit n zs*!*i*) ∈ *R*
    **by** (*metis assms le-eq-less-or-eq le-less-trans nth-sinit transD*)
  **moreover from** *lt sinitm sinitn* **have** ∀ *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*) ∈ *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*) ∈ *R*
    **by** (*metis less-trans nth-sinit*)
  **moreover from** *ge sinitm sinitn* **have** ∀ *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*) ∈ *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*:
  ⟦ (*xs*, *ys*) ∈ *seq-lexord R*; (*ys*, *zs*) ∈ *seq-lexord R*; *trans R* ⟧ ⟹ (*xs*, *zs*) ∈ *seq-lexord R*
  **by** (*meson seq-lexord-transitive transE*)

**lemma** *seq-lexord-antisym*:
  ⟦ *asym R*; (*a*, *b*) ∈ *seq-lexord R* ⟧ ⟹ (*b*, *a*) ∉ *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 (∀ a b. (a,b)∈ R ∨ a = b ∨ (b,a) ∈ R)
  shows (x,y) ∈ seq-lexord R ∨ x = y ∨ (y,x) ∈ 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 ⇒ 'a seq ⇒ bool where
less-seq xs ys ⟷ (xs, ys) ∈ seq-lexord {(xs, ys). xs < ys}

definition less-eq-seq :: 'a seq ⇒ 'a seq ⇒ bool where
less-eq-seq xs ys = (xs = ys ∨ xs < ys)

instance ..

end

instance seq :: (order) order
proof
  fix xs :: 'a seq
  show xs ≤ xs by (simp add: less-eq-seq-def)
next
  fix xs ys zs :: 'a seq
  assume xs ≤ ys and ys ≤ zs
  then show xs ≤ 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 ≤ ys and ys ≤ 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 ⟷ xs ≤ ys ∧ ¬ ys ≤ 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) ∈ seq-lexord {(u, v). u < v} ∨ xs = ys ∨ (ys, xs) ∈ seq-lexord {(u, v). u < v}
    by (rule seq-lexord-linear) auto
  then show xs ≤ ys ∨ ys ≤ xs
    by (auto simp add: less-eq-seq-def less-seq-def)
qed

lemma seq-lexord-mono [mono]:
  (⋀ x y. f x y ⟶ g x y) ⟹ (xs, ys) ∈ seq-lexord {(x, y). f x y} ⟶ (xs, ys) ∈ 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 ⇒ 'a ⇒ 'a list ⇒ 'a list where
insort-rel R x [] = [x] |
insort-rel R x (y # ys) = (if (x = y ∨ (x,y) ∈ R) then x # y # ys else y # insort-rel R x ys)

inductive sorted-rel :: 'a rel ⇒ 'a list ⇒ bool where
Nil-rel [iff]: sorted-rel R [] |
Cons-rel: ∀ y ∈ set xs. (x = y ∨ (x, y) ∈ R) ⟹ sorted-rel R xs ⟹ sorted-rel R (x # xs)

definition list-of-set :: 'a rel ⇒ 'a set ⇒ 'a list where
list-of-set R = folding.F (insort-rel R) []

lift-definition seq-inj :: 'a seq seq ⇒ 'a seq is
λ f i. f (fst (prod-decode i)) (snd (prod-decode i)) .

lift-definition seq-proj :: 'a seq ⇒ 'a seq seq is
λ f i j. f (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
```

# 5 Finite Sets: extra functions and properties

**theory** *FSet-Extra*
**imports**
  ~~/*src*/*HOL*/*Library*/*FSet*
  ~~/*src*/*HOL*/*Library*/*Countable-Set-Type*
**begin**

**setup-lifting** *type-definition-fset*

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

**syntax**
  *-FinFset* :: *args* => *'a fset*  ($\{\!|(\text{-})|\!\}$)

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

**term** *fBall*

**syntax**
  *-fBall* :: *pttrn* => *'a fset* => *bool* => *bool* ((*3*∀ -|∈|-./ -) [*0*, *0*, *10*] *10*)
  *-fBex* :: *pttrn* => *'a fset* => *bool* => *bool* ((*3*∃ -|∈|-./ -) [*0*, *0*, *10*] *10*)

**translations**
  ∀ $x$|∈|$A$. $P$ == *CONST fBall A* (%$x$. $P$)
  ∃ $x$|∈|$A$. $P$ == *CONST fBex A* (%$x$. $P$)

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

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

**lemma** *fset-intro*:
  *fset* $x$ = *fset* $y$ ⟹ $x = y$
  **by** (*simp add:fset-inject*)

**lemma** *fset-elim*:
  ⟦ $x = y$; *fset* $x$ = *fset* $y$ ⟹ $P$ ⟧ ⟹ $P$
  **by** (*auto*)

**lemma** *fmember-intro*:
  $\llbracket\ x \in \mathit{fset}(xs)\ \rrbracket \Longrightarrow x\ |{\in}|\ xs$
  **by** (*metis fmember.rep-eq*)

**lemma** *fmember-elim*:
  $\llbracket\ x\ |{\in}|\ xs;\ x \in \mathit{fset}(xs) \Longrightarrow P\ \rrbracket \Longrightarrow P$
  **by** (*metis fmember.rep-eq*)

**lemma** *fnmember-intro* [*intro*]:
  $\llbracket\ x \notin \mathit{fset}(xs)\ \rrbracket \Longrightarrow x\ |{\notin}|\ xs$
  **by** (*metis fmember.rep-eq*)

**lemma** *fnmember-elim* [*elim*]:
  $\llbracket\ x\ |{\notin}|\ xs;\ x \notin \mathit{fset}(xs) \Longrightarrow P\ \rrbracket \Longrightarrow P$
  **by** (*metis fmember.rep-eq*)

**lemma** *fsubset-intro* [*intro*]:
  $\langle xs\rangle_f \subseteq \langle ys\rangle_f \Longrightarrow xs\ |{\subseteq}|\ ys$
  **by** (*metis less-eq-fset.rep-eq*)

**lemma** *fsubset-elim* [*elim*]:
  $\llbracket\ xs\ |{\subseteq}|\ ys;\ \langle xs\rangle_f \subseteq \langle ys\rangle_f \Longrightarrow P\ \rrbracket \Longrightarrow P$
  **by** (*metis less-eq-fset.rep-eq*)

**lemma** *fBall-intro* [*intro*]:
  $\mathit{Ball}\ \langle A\rangle_f\ P \Longrightarrow \mathit{fBall}\ A\ P$
  **by** (*metis* (*poly-guards-query*) *fBallI fmember.rep-eq*)

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

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

**context** *linorder*
**begin**

**lemma** *sorted-list-of-set-inj*:
  ⟦ *finite xs*; *finite ys*; *sorted-list-of-set xs* = *sorted-list-of-set ys* ⟧
   $\Longrightarrow xs = ys$
  **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* ⇒ *'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*]: ⟦ *sorted xs*; *distinct xs* ⟧ ⟹ *flist* (*finset xs*) = *xs*
**apply** (*simp add:finset-def flist-def fset-inverse*)
**apply** (*metis local.sorted-list-of-set-sort-remdups local.sorted-sort-id remdups-id-iff-distinct*)
**done**

**lemma** *fcard-flist*:
*fcard xs* = *length* (*flist xs*)
**apply** (*simp add:fcard-def*)
**apply** (*fold flist-set*)
**apply** (*unfold distinct-card*[*OF flist-props*(*2*)])
**apply** (*rule refl*)
**done**

**lemma** *flist-nth*:
*i* < *fcard vs* ⟹ *flist vs* ! *i* |∈| *vs*
**apply** (*simp add*: *fmember-def flist-def fcard-def*)
**apply** (*metis distinct-card finite-fset nth-mem sorted-list-of-set*)
**done**

**definition** *fmax* :: ′*a fset* ⇒ ′*a* **where**
*fmax xs* = (*if* (*xs* = {||}) *then undefined else last* (*flist xs*))

**end**

**definition** *flists* :: ′*a fset* ⇒ ′*a list set* **where**
*flists A* = {*xs*. *distinct xs* ∧ *finset xs* = *A*}

**lemma** *flists-nonempty*: ∃ *xs*. *xs* ∈ *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*: ⟦ *x* ∈ *flists A*; *x* ∈ *flists B* ⟧ ⟹ *A* = *B*
**by** (*simp add*: *flists-def*)

**definition** *flist-arb* :: *'a fset* ⇒ *'a list* **where**
*flist-arb A* = (*SOME xs. xs* ∈ *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* ⊆ *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* ⇒ *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* ∘ *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* ⇒ *'a fset* ⇒ *'a fset* **where**
*flub A t* = (*if* (∀ *a*∈*A*. *a* |⊆| *t*) *then Abs-fset* ($\bigcup$*x*∈*A*. $\langle x\rangle_f$) *else t*)

**lemma** *finite-Union-subsets*:
  ⟦ ∀ *a* ∈ *A*. *a* ⊆ *b*; *finite b* ⟧ ⟹ *finite* ($\bigcup$*A*)
  **by** (*metis Sup-le-iff finite-subset*)

**lemma** *finite-UN-subsets*:
  ⟦ ∀ *a* ∈ *A*. *B a* ⊆ *b*; *finite b* ⟧ ⟹ *finite* ($\bigcup$*a*∈*A*. *B a*)
  **by** (*metis UN-subset-iff finite-subset*)

**lemma** *flub-rep-eq*:
  $\langle$*flub A t*$\rangle_f$ = (*if* (∀ *a*∈*A*. *a* |⊆| *t*) *then* ($\bigcup$*x*∈*A*. $\langle x\rangle_f$) *else* $\langle t\rangle_f$)
  **apply** (*subgoal-tac* (*if* (∀ *a*∈*A*. *a* |⊆| *t*) *then* ($\bigcup$*x*∈*A*. $\langle x\rangle_f$) *else* $\langle t\rangle_f$) ∈ {*x*. *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* ⇒ *'a fset* ⇒ *'a fset* **where**
*fglb A t* = (*if* (*A* = {}) *then t else Abs-fset* ($\bigcap$*x*∈*A*. $\langle x\rangle_f$))

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

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

**lemma** *FUnion-rep-eq* [*simp*]:
$\langle \bigcup_f\ xs\rangle_f = (\bigcup x{\in}\langle xs\rangle_f.\ \langle x\rangle_f)$
**by** (*simp add*:*FUnion-def*)

**lemma** *FInter-rep-eq* [*simp*]:
$xs \neq \{\!|\!\} \Longrightarrow \langle \bigcap_f\ xs\rangle_f = (\bigcap x{\in}\langle xs\rangle_f.\ \langle x\rangle_f)$
**apply** (*simp add*:*FInter-def*)
**apply** (*subgoal-tac finite* $(\bigcap x{\in}\langle xs\rangle_f.\ \langle x\rangle_f)$)
 **apply** (*simp*)
**apply** (*metis* (*poly-guards-query*) *bot-fset.rep-eq fglb-rep-eq finite-fset fset-inverse*)
**done**

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

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

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

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

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

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

**end**

# 6 Countable Sets: Extra functions and properties

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

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

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

## 6.2 Countable set functions

**setup-lifting** *type-definition-cset*

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

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

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

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

**syntax**
  *-cBall* :: *pttrn* => *'a cset* => *bool* => *bool* ((*3*$\forall$ -$\in_c$-./ -) [*0*, *0*, *10*] *10*)
  *-cBex* :: *pttrn* => *'a cset* => *bool* => *bool* ((*3*$\exists$ -$\in_c$-./ -) [*0*, *0*, *10*] *10*)

**translations**
  $\forall$ $x \in_c A$. *P* == *CONST cBall A* (%*x*. *P*)
  $\exists$ $x \in_c A$. *P* == *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* => *bool* => *'a cset* ((*1*{-./ -}$_c$))

**translations**
$\{x\ .\ P\}_c \rightleftharpoons$ (*CONST cset-Collect*) ($\lambda$ *x* . *P*)

**syntax** (*xsymbols*)
*-cCollect* :: *pttrn* => *'a cset* => *bool* => *'a cset* ((*1*{- $\in_c$/ -./ -}$_c$))
**translations**
$\{x \in_c A.\ P\}_c$ => *CONST cset-Coll A* ($\lambda$ *x*. *P*)

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

**lemma** *cset-CollI*: $[\![\ a \in_c A;\ P\ a\ ]\!] \Longrightarrow a \in_c \{x \in_c A.\ P\ x\}_c$
**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*: $(\bigwedge x.\ P\ x = Q\ x)$ ==> $\{x.\ P\ x\}_c = \{x.\ Q\ x\}_c$
**by** *simp*

— Avoid eta-contraction for robust pretty-printing.

**print-translation** ‹
[*Syntax-Trans.preserve-binder-abs-tr'*
@{*const-syntax cset-Collect*} @{*syntax-const -cColl*}]
›

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

**lemma** *countable-finite-power*:
*countable*(*A*) $\Longrightarrow$ *countable* $\{B.\ B \subseteq A \wedge$ *finite*(*B*)$\}$
**by** (*metis Collect-conj-eq Int-commute countable-Collect-finite-subset*)

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

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

**lift-definition** *cfinite* :: *'a cset* $\Rightarrow$ *bool* **is** *finite* **.**
**lift-definition** *cInfinite* :: *'a cset* $\Rightarrow$ *bool* **is** *infinite* **.**
**lift-definition** *clist* :: *'a*::*linorder cset* $\Rightarrow$ *'a list* **is** *sorted-list-of-set* **.**
**lift-definition** *ccard* :: *'a cset* $\Rightarrow$ *nat* **is** *card* **.**
**lift-definition** *cPow* :: *'a cset* $\Rightarrow$ *'a cset cset* **is** $\lambda$ *A*. $\{B.\ B \subseteq_c A \wedge$ *cfinite*(*B*)$\}$
**proof** −
**fix** *A*

```
  have {B :: 'a cset. B ⊆c A ∧ cfinite B} = acset ‘ {B :: 'a set. B ⊆ rcset A ∧ finite B}
    apply (auto simp add: cfinite.rep-eq cin-def less-eq-cset-def countable-finite)
    using image-iff apply fastforce
    done

  moreover have countable {B :: 'a set. B ⊆ rcset A ∧ finite B}
    by (auto intro: countable-finite-power)

  ultimately show countable {B. B ⊆c A ∧ cfinite B}
    by simp
qed

definition CCollect :: ('a ⇒ bool option) ⇒ 'a cset option where
CCollect p = (if (None ∉ range p) then Some (cset-Collect (the ∘ p)) else None)

definition cset-mapM :: 'a option cset ⇒ 'a cset option where
cset-mapM A = (if (None ∈c A) then None else Some (the ‘c A))

lemma cset-mapM-Some-image [simp]:
  cset-mapM (cimage Some A) = Some A
  apply (auto simp add: cset-mapM-def)
  apply (metis cimage-cinsert cinsertI1 option.sel set-cinsert)
done

definition CCollect-ext :: ('a ⇒ 'b option) ⇒ ('a ⇒ bool option) ⇒ 'b cset option where
CCollect-ext f p = do { xs ← CCollect p; cset-mapM (f ‘c xs) }

lemma the-Some-image [simp]:
  the ‘ Some ‘ xs = xs
  by (auto simp add:image-iff)

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

lift-definition list-of-cset :: 'a :: linorder cset ⇒ 'a list is sorted-list-of-set .

lift-definition fset-cset :: 'a fset ⇒ 'a cset is id
  using uncountable-infinite by auto

definition cset-count :: 'a cset ⇒ 'a ⇒ nat where
cset-count A =
  (if (finite (rcset A))
   then (SOME f::'a⇒nat. inj-on f (rcset A))
   else (SOME f::'a⇒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 ⇒ 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* ⇒ *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* ⇒ *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* ⇒ (*nat* ⇀ *'a*) **where**
$cset\text{-}seq\ A\ i = (if\ (i \in range\ (cset\text{-}count\ A) \wedge inv\text{-}into\ (rcset\ A)\ (cset\text{-}count\ A)\ i \in_c A)$
$then\ Some\ (inv\text{-}into\ (rcset\ A)\ (cset\text{-}count\ A)\ i)$
$else\ None)$

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

**lemma** *range-cset2seq*:
$A \neq \{\}_c \Longrightarrow range\ (Rep\text{-}seq\ (cset2seq\ A)) = rcset\ A$
**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*) ⟹ *surj* (*cset-count A*)
**using** *bij-betw-imp-surj cset-count-infinite-bij* **by** *auto*

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

**done**

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

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

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

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

**lemma** *bit-seq-of-nat-cset-bij*: *bij bit-seq-of-nat-set*
  **apply** (*rule bijI*)
   **apply** (*fact bit-seq-of-nat-set-inj*)
  **apply** (*auto simp add*: *image-def*)
  **apply** (*transfer*)
  **apply** (*rename-tac x*)
  **apply** (*rule-tac* $x=\{i.\ x\ i = 1\}$ **in** *exI*)
  **apply** (*auto*)
  **done**

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

**definition** *nat-set-cset-collapse* :: *nat set cset* ⇒ *nat set* **where**
*nat-set-cset-collapse* = *inv bit-seq-of-nat-set* ∘ *seq-inj* ∘ *cset2seq* ∘ ($\lambda$ *A*. (*bit-seq-of-nat-set* $‘_c$ *A*))

**lemma** *nat-set-cset-collapse-inj*: *inj-on nat-set-cset-collapse* $\{A.\ A \neq \{\}_c\}$
**proof** −
  **have** *op* $‘_c$ *bit-seq-of-nat-set* ‘ $\{A.\ A \neq \{\}_c\} \subseteq \{A.\ A \neq \{\}_c\}$
    **by** (*auto simp add*:*cimage.rep-eq*)
  **thus** *?thesis*
    **apply** (*simp add*: *nat-set-cset-collapse-def*)
    **apply** (*rule comp-inj-on*)
     **apply** (*meson bit-seq-of-nat-set-inj cset.inj-map injD inj-onI*)
    **apply** (*rule comp-inj-on*)
     **apply** (*metis cset2seq-inj subset-inj-on*)
    **apply** (*rule comp-inj-on*)
     **apply** (*rule subset-inj-on*)
      **apply** (*rule seq-inj*)
     **apply** (*simp*)

**apply** (*meson UNIV-I bij-imp-bij-inv bij-is-inj bit-seq-of-nat-cset-bij subsetI subset-inj-on*)
**done**
**qed**

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

**lemma** *range-csingle*:
*range csingle* $\subseteq$ $\{A.\ A \neq \{\}_c\}$
**by** (*auto*)

**lift-definition** *csets* :: *'a set* $\Rightarrow$ *'a cset set* **is**
$\lambda$ *A*. $\{B.\ B \subseteq A \wedge$ *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 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* ⇒ *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* ⟹ *inj-on* (*op* ‘$_c$ *f*) (*csets A*)
**by** (*fastforce simp add*: *inj-on-def cimage-def cin-def csets-def*)

**lemma** *image-csets-surj*:
⟦ *inj-on f A*; *f* ‘ *A* = *B* ⟧ ⟹ *op* ‘$_c$ *f* ‘ *csets A* = *csets B*
**apply** (*auto simp add*: *cimage-def csets-def image-mono map-fun-def*)
**apply** (*simp add*: *image-comp*)
**apply** (*auto simp add*: *image-Collect*)
**apply** (*erule subset-imageE*)
**apply** (*auto*)
**apply** (*metis countable-image rcset-inverse rcset-to-rcset subset-inj-on the-inv-into-onto*)
**done**

**lemma** *bij-betw-image-csets*:
*bij-betw f A B* ⟹ *bij-betw* (*op* ‘$_c$ *f*) (*csets A*) (*csets B*)
**by** (*simp add*: *bij-betw-def inj-on-image-csets image-csets-surj*)
**end**

# 7 Map Type: extra functions and properties

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

## 7.1 Functional Relations

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

**definition** *functional-list* :: ('*a* ∗ '*b*) *list* ⇒ *bool* **where**
*functional-list xs* = (∀ *x y z*. *ListMem* (*x*,*y*) *xs* ∧ *ListMem* (*x*,*z*) *xs* ⟶ *y* = *z*)

**lemma** *functional-insert* [*simp*]: *functional* (*insert* (*x*,*y*) *g*) ⟷ (*g*‘‘{*x*} ⊆ {*y*} ∧ *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*)
  **apply** (*force*)
  **done**

## 7.2 Graphing Maps

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

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

**definition** *graph-map'* :: $('a \times 'b)$ *set* $\rightharpoonup ('a \rightharpoonup 'b)$ **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*) $\Longrightarrow$ *countable* (*map-graph f*)
  **apply** (*auto simp add:map-graph-def countable-def*)
  **apply** (*rename-tac f'*)
  **apply** (*rule-tac x=f'* $\circ$ *fst* **in** *exI*)
  **apply** (*auto simp add:inj-on-def dom-def*)
  **apply** *fastforce*
  **done**

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

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

**lemma** *graph-map-insert* [*simp*]: $[\![$*functional g*; $g``\{x\} \subseteq \{y\}]\!] \Longrightarrow$ *graph-map* (*insert* $(x,y)$ *g*) = (*graph-map g*)($x \mapsto y$)
  **by** (*rule ext*, *auto simp add:graph-map-def*)

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

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *map-graph-comp*: *map-graph* $(g \circ_m f)$ = (*map-graph f*) *O* (*map-graph g*)
  **apply** (*auto simp add*: *map-comp-def map-graph-def relcomp-unfold*)

**apply** (*rename-tac a b*)
**apply** (*case-tac f a*, *auto*)
**done**

## 7.3 Map Application

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

## 7.4 Map Membership

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

**lemma** *map-ext*:
$[\![ \bigwedge x\ y.\ (x,\ y) \in_m A \longleftrightarrow (x,\ y) \in_m B ]\!] \Longrightarrow A = B$
**by** (*rule ext*, *auto*, *metis not-Some-eq*)

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

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

## 7.5 Preimage

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

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

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

**lemma** *countable-preimage*:
$[\![$ *countable A*; *inj-on f* (*preimage f A*) $]\!] \Longrightarrow$ *countable* (*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**

## 7.6 Minus operation for maps

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

**lemma** *map-minus-apply* [*simp*]: $y \in dom(f\ --\ g) \Longrightarrow (f\ --\ g)(y)_m = f(y)_m$

**by** (*auto simp add*: *map-minus-def dom-def map-apply-def*)

**lemma** *map-member-plus*:
$(x,\ y) \in_m f\ ++\ g \longleftrightarrow ((x \notin dom(g) \wedge (x,\ y) \in_m f) \vee (x,\ y) \in_m g)$
**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*:
$dom(g) \cap dom(h) = \{\} \Longrightarrow (f\ --\ g)\ ++\ h = (f\ ++\ h)\ --\ g$
**apply** (*rule map-ext*)
**apply** (*auto simp add*: *map-member-plus map-member-minus simp del*: *map-member.simps*)
**apply** (*auto simp add*: *map-member-alt-def*)
**done**

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

**lemma** *map-minus-common-subset*:
$[\![\ h \subseteq_m f;\ h \subseteq_m g\ ]\!] \Longrightarrow (f\ --\ h = g\ --\ h) = (f = g)$
**by** (*auto simp add*: *map-eq-graph map-graph-minus map-le-graph*)

## 7.7 Map Bind

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

**lemma** *option-bindSomeE* [*elim!*]:
$[\![\ X >>= F = Some(v);\ \bigwedge x.\ [\![\ X = Some(x);\ F(x) = Some(v)\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
**by** (*case-tac X*, *auto*)

**lemma** *option-bindSomeI* [*intro*]:
$[\![\ X = Some(x);\ F(x) = Some(y)\ ]\!] \Longrightarrow X >>= F = Some(y)$
**by** (*simp*)

**lemma** *ifSomeE* [*elim*]: $[\![\ (if\ c\ then\ Some(x)\ else\ None) = Some(y);\ [\![\ c;\ x = y\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
**by** (*case-tac c*, *auto*)

## 7.8 Range Restriction

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

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

**lemma** *ran-restrict-empty* [*simp*]: $f{\restriction}_{\{\}}$ = *Map.empty*
**by** (*simp add*:*ran-restrict-map-def*)

**lemma** *ran-restrict-ran* [*simp*]: $f{\restriction}_{ran(f)} = f$
**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*]: $ran(f{\restriction}_B) = ran(f) \cap B$

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

**lemma** *dom-ran-restrict*: $dom(f\upharpoonright_B) \subseteq dom(f)$
**by** (*auto simp add*:*ran-restrict-map-def dom-def*)

**lemma** *ran-restrict-finite-dom* [*intro*]:
$finite(dom(f)) \Longrightarrow finite(dom(f\upharpoonright_B))$
**by** (*metis finite-subset dom-ran-restrict*)

**lemma** *dom-Some* [*simp*]: $dom\ (Some \circ f) = UNIV$
**by** (*auto*)

**lemma** *dom-left-map-add* [*simp*]: $x \in dom\ g \Longrightarrow (f\ {+}{+}\ g)\ x = g\ x$
**by** (*auto simp add*:*map-add-def dom-def*)

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

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

## 7.9 Map Inverse and Identity

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

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

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

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

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

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

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

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

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

**lemma** *map-inv-id* [*simp*]:
$map\text{-}inv\ (map\text{-}id\text{-}on\ xs) = map\text{-}id\text{-}on\ xs$

**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* $]\!] \Longrightarrow$ *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*) $\Longrightarrow$ *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*) $\Longrightarrow$ *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*=$\lambda$ *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*) $\Longrightarrow$ *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* ∃ *y*. *map-inv f y* = *Some x*)
**apply** (*auto*)[*1*]
**apply** (*simp add*:*map-inv-def*)
**apply** (*rename-tac x y*)
**apply** (*case-tac y* ∈ *ran f*, *simp-all*)
**apply** (*auto*)
**apply** (*rule someI2-ex*)
**apply** (*simp add*:*ran-def*)
**apply** (*simp*)
**apply** (*metis assms dom-image-ran dom-map-inv image-iff map-add-dom-app-simps*(*2*) *map-add-dom-app-simps*(*3*) *ran-map-inv*)
**done**
**qed**

**lemma** *map-self-adjoin-complete* [*intro*]:
**assumes** *dom f* ∩ *ran f* = {} *inj-on f* (*dom f*)
**shows** *inj-on* (*map-inv f* ++ *f*) (*dom f* ∪ *ran f*)
**apply** (*rule inj-onI*)
**apply** (*insert assms*)
**apply** (*rename-tac x y*)
**apply** (*case-tac x* ∈ *dom f*)
**apply** (*simp*)
**apply** (*case-tac y* ∈ *dom f*)
**apply** (*simp add*:*inj-on-def*)
**apply** (*case-tac y* ∈ *ran f*)
**apply** (*subgoal-tac y* ∈ *dom* (*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* ∈ *dom f*)
**apply** (*simp*)
**apply** (*case-tac y* ∈ *ran f*)
**apply** (*subgoal-tac y* ∈ *dom* (*map-inv f*))
**apply** (*simp*)
**apply** (*metis Int-iff domD empty-iff ranI ran-map-inv*)
**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*]:
⟦ *dom f* = *ran f*; *inj-on f* (*dom f*) ⟧ ⟹ *inj* (*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* ∈ *dom f*)
**apply** (*simp*)
**apply** (*case-tac y* ∈ *dom f*)
**apply** (*simp*)
**apply** (*simp add*:*ran-def*)
**apply** (*case-tac y* ∈ *dom f*)

```
   apply (auto intro:ranI)
  done

lemma bij-completed-map [intro]:
  ⟦ dom f = ran f; inj-on f (dom f) ⟧ ⟹
   bij-betw (Some ++ f) UNIV (range Some)
  apply (auto intro: inj-completed-map simp add:bij-betw-def)
   apply (rename-tac x)
   apply (case-tac x ∈ 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) ⟹ bij-betw (the ∘ 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‘(dom m ∩ dom n) = n‘(dom m ∩ dom n) ⟹
   ran(m++n) = ran n ∪ ran m
  apply (auto simp add:ran-def)
   apply (metis map-add-find-right)
  apply (rename-tac x a)
  apply (case-tac a ∈ dom n)
   apply (subgoal-tac ∃ b. n b = Some x)
    apply (auto)
    apply (rename-tac x a b y)
    apply (rule-tac x=b in exI)
    apply (simp)
   apply (metis (hide-lams, no-types) IntI domI image-iff)
 apply (metis (full-types) map-add-None map-add-dom-app-simps(1) map-add-dom-app-simps(3) not-None-eq)
  done

lemma ran-maplets [simp]:
  ⟦ length xs = length ys; distinct xs ⟧ ⟹ ran [xs [↦] ys] = 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 ∩ ran g = {} ⟧ ⟹
   inj-on (f ++ g) (dom f ∪ 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* $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *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* $\in$ *ran g*)
**apply** (*simp*)
**apply** (*case-tac x* $\in$ *ran f*)
**apply** (*simp-all*)
**done**
**qed**

**lemma** *map-add-lookup* [*simp*]:
*x* $\notin$ *dom f* $\Longrightarrow$ ([*x* $\mapsto$ *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]:
  ⟦ length xs = length ys; distinct xs; distinct ys; set xs ∩ set ys = {} ⟧ ⟹
```

*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*]:
⟦ *length xs* = *length ys*; *distinct xs* ⟧ ⟹
∀ *i* < *length ys*. [*xs* [↦] *ys*] (*xs* ! *i*) = *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** *the-Some*[*simp*]: *the* ∘ *Some* = *id*
**by** (*simp add*:*comp-def id-def*)

**theorem** *inv-map-inv*:
⟦ *inj-on f* (*dom f*); *ran f* = *dom f* ⟧
⟹ *inv* (*the* ∘ (*Some* ++ *f*)) = *the* ∘ *map-inv* (*Some* ++ *f*)
**apply** (*rule ext*)
**apply** (*simp add*:*map-add-Some*)
**apply** (*simp add*:*inv-def*)
**apply** (*rename-tac x*)
**apply** (*case-tac* ∃ *y*. *f y* = *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* ∉ *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*: $dom\ (g \circ_m f) \subseteq dom\ f$
  **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*]: $f \circ_m Some = f$
  **by** (*simp add*: *map-comp-def*)

**lemma** *map-comp-lunit* [*simp*]: $Some \circ_m f = f$
**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*)

## 7.10 Merging of compatible maps

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

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

**lemma** *comp-map-refl*: $f \|_m f$
  **by** (*simp add*: *comp-map-def*)

**lemma** *comp-map-sym*: $f \|_m g \Longrightarrow g \|_m f$
  **by** (*simp add*: *comp-map-def*)

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

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

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

## 7.11 Conversion between lists and maps

**definition** *map-of-list* :: $'a$ *list* $\Rightarrow$ (*nat* $\rightharpoonup$ $'a$) **where**
*map-of-list xs* = ($\lambda$ *i*. *if* (*i* < *length xs*) *then Some* (*xs*!*i*) *else None*)

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

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

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

**definition** *list-of-map* :: (*nat* $\rightharpoonup$ $'a$) $\Rightarrow$ $'a$ *list* **where**
*list-of-map f* = (*if* (*f* = *Map.empty*) *then* [] *else map* (*the* $\circ$ *f*) [*0* ..< *Suc*(*GREATEST x*. *x* $\in$ *dom f*)])

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

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

**lemma** *map-of-list-inv* [*simp*]: *list-of-map* (*map-of-list xs*) = *xs*
**proof** (*cases xs* = [])
**case** *True* **thus** *?thesis* **by** (*simp*)
**next**
**case** *False*
**moreover hence** (*GREATEST x*. *x* $\in$ *dom* (*map-of-list xs*)) = *length xs* − *1*
**by** (*auto intro*: *Greatest-equality*)
**moreover from** *False* **have** *map-of-list xs* $\neq$ *Map.empty*
**by** (*metis ran-empty ran-map-of-list set-empty*)
**ultimately show** *?thesis*
**by** (*auto simp add*: *list-of-map-def map-of-list-def nth-equalityI*)
**qed**

## 7.12 Map Comprehension

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

**syntax**
*-Mapcompr* :: $'a$ $\Rightarrow$ $'b$ $\Rightarrow$ *idts* $\Rightarrow$ *bool* $\Rightarrow$ $'a$ $\rightharpoonup$ $'b$ ((*1*[- $\mapsto$ - |/-./ -]))

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

**lemma** *map-compr-eta*:
  $[x \mapsto y \mid x\ y.\ (x,\ y) \in_m f] = f$
  **apply** (*rule ext*)
  **apply** (*auto simp add*: *graph-map-def*)
  **apply** (*metis* (*mono-tags*, *lifting*) *Domain.DomainI fst-eq-Domain mem-Collect-eq old.prod.case option.distinct*(*1*) *option.expand option.sel*)
  **done**

**lemma** *map-compr-simple*:
  $[x \mapsto F\ x\ y \mid x\ y.\ (x,\ y) \in_m f] = (\lambda\ x.\ do\ \{\ y \leftarrow f(x);\ Some(F\ x\ y)\ \})$
  **apply** (*rule ext*)
  **apply** (*auto simp add*: *graph-map-def image-Collect*)
  **done**

**lemma** *map-compr-dom-simple* [*simp*]:
  $dom\ [x \mapsto f\ x \mid x.\ P\ x] = \{x.\ P\ x\}$
  **by** (*force simp add*: *graph-map-dom image-Collect*)

**lemma** *map-compr-ran-simple* [*simp*]:
  $ran\ [x \mapsto f\ x \mid x.\ P\ x] = \{f\ x \mid x.\ P\ x\}$
  **apply** (*auto simp add*: *graph-map-def ran-def*)
  **apply** (*metis* (*mono-tags*, *lifting*) *fst-eqD image-eqI mem-Collect-eq someI*)
  **done**

**lemma** *map-compr-eval-simple* [*simp*]:
  $[x \mapsto f\ x \mid x.\ P\ x]\ x = (if\ (P\ x)\ then\ Some\ (f\ x)\ else\ None)$
  **by** (*auto simp add*: *graph-map-def image-Collect*)

## 7.13 Sorted lists from maps

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

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

**lemma** *sorted-list-of-map-inv*:
  **assumes** *finite*(*dom*(*f*))
  **shows** *map-of* (*sorted-list-of-map f*) = *f*
**proof** −
  **obtain** *A* **where** *finite A A* = *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*: *sorted-list-of-map-def*, *metis finite-insert map-of-map-keys sorted-list-of-set*)
  **qed**
**qed**

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

### 7.14 Extra map lemmas

**lemma** *map-eqI*:
  $[\![$ *dom* $f$ = *dom* $g$; $\forall$ $x \in$*dom*$(f)$. *the*$(f\ x)$ = *the*$(g\ x)$ $]\!] \Longrightarrow f = g$
  **by** (*metis domIff map-le-antisym map-le-def option.expand*)

**lemma** *map-restrict-dom-compl*: $f$ |' ($-$ *dom* $f$) = *Map.empty*
  **by** (*metis dom-eq-empty-conv dom-restrict inf-compl-bot*)

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

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

**lemma** *map-plus-eq-left*:
  **assumes** $f$ ++ $h = g$ ++ $h$
  **shows** $(f$ |' $(-$ *dom* $h)) = (g$ |' $(-$ *dom* $h))$
**proof** $-$
  **have** $h$ |' $(-$ *dom* $h)$ = *Map.empty*
    **by** (*metis Compl-disjoint dom-eq-empty-conv dom-restrict*)
  **then have** *f2*: $f$ |' $(-$ *dom* $h) = (f$ ++ $h)$ |' $(-$ *dom* $h)$
    **by** (*simp add*: *map-plus-restrict-dist*)
  **have** $h$ |' $(-$ *dom* $h)$ = *Map.empty*
    **by** (*metis* (*no-types*) *Compl-disjoint dom-eq-empty-conv dom-restrict*)
  **then show** *?thesis*
    **using** *f2 assms* **by** (*simp add*: *map-plus-restrict-dist*)
**qed**

**lemma** *map-add-split*:
  *dom*$(f) = A \cup B \Longrightarrow (f$ |' $A)$ ++ $(f$ |' $B) = f$
  **by** (*rule ext*, *auto simp add*: *map-add-def restrict-map-def option.case-eq-if*)

**lemma** *map-le-via-restrict*:
  $f \subseteq_m g \longleftrightarrow g$ |' *dom*$(f) = f$
  **by** (*auto simp add*: *map-le-def restrict-map-def dom-def fun-eq-iff*)

**end**

# 8 Alternative List Lexicographic Order

**theory** *List-Lexord-Alt*
  **imports** *Main*
**begin**

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

**definition** *list-lex-less* :: *'a::linorder list* $\Rightarrow$ *'a list* $\Rightarrow$ *bool* (**infix** $<_l$ *50*)
**where** $xs <_l ys \longleftrightarrow (xs, ys) \in$ *lexord* $\{(u, v).\ u < v\}$

**lemma** *list-lex-less-neq* [*simp*]: $x <_l y \Longrightarrow x \neq y$
  **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*]: $a \# x <_l b \# y \longleftrightarrow a < b \lor a = b \land x <_l y$
**by** (*simp add*: *list-lex-less-def*)
**end**

# 9 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.

## 9.1 Partial function type and operations

**typedef** $('a, 'b)$ *pfun* = *UNIV* :: $('a \rightharpoonup 'b)$ *set* **..**

**setup-lifting** *type-definition-pfun*

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

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

**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** *op* $\in_m$ **.**

**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** *pdom-res* :: $'a$ *set* $\Rightarrow ('a, 'b)$ *pfun* $\Rightarrow ('a, 'b)$ *pfun* (**infixl** $\lhd_p$ *85*)
**is** $\lambda\ A\ f.$ *restrict-map f A* **.**

**lift-definition** *pran-res* :: $('a, 'b)$ *pfun* $\Rightarrow 'b$ *set* $\Rightarrow ('a, 'b)$ *pfun* (**infixl** $\rhd_p$ *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* = *card* (*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** *op* ++ **.**
**instance ..**
**end**

**instantiation** *pfun* :: (*type*, *type*) *minus*
**begin**
**lift-definition** *minus-pfun* :: $('a, 'b)$ *pfun* $\Rightarrow$ $('a, 'b)$ *pfun* $\Rightarrow$ $('a, 'b)$ *pfun* **is** *op* $--$ **.**
**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. if* $(x \in dom(f) \cap dom(g) \land f(x) = g(x))$ *then* $f(x)$ *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.* $f \subseteq_m g \land f \neq g$ **.**
**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* ≡ *less-eq*

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

**lemma** *pfun-subset-eq-least* [*simp*]:
  $\{\}_p \subseteq_p f$
  **by** (*transfer*, *auto*)

**syntax**
  *-PfunUpd* :: [(*'a*, *'b*) *pfun*, *maplets*] => (*'a*, *'b*) *pfun* (*-'(-')*$_p$ [*900*,*0*]*900*)
  *-Pfun* :: *maplets* => (*'a*, *'b*) *pfun* ((*1*{*-*}$_p$))

**translations**
  *-PfunUpd m* (*-Maplets xy ms*) == *-PfunUpd* (*-PfunUpd m xy*) *ms*
  *-PfunUpd m* (*-maplet x y*) == *CONST pfun-upd m x y*
  *-Pfun ms* => *-PfunUpd* (*CONST pempty*) *ms*
  *-Pfun* (*-Maplets ms1 ms2*) <= *-PfunUpd* (*-Pfun ms1*) *ms2*
  *-Pfun ms* <= *-PfunUpd* (*CONST pempty*) *ms*

## 9.2 Algebraic laws

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

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

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

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

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

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

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

**shows** $f - f = 0$
**by** (*transfer*, *simp add*: *map-minus-def*)

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

**lemma** *pfun-plus-minus*:
$f \subseteq_p g \Longrightarrow (g - f) + f = g$
**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*:
$[\![ h \subseteq_p f; h \subseteq_p g ]\!] \Longrightarrow (f - h = g - h) = (f = g)$
**by** (*transfer*, *simp add*: *map-minus-common-subset*)

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

## 9.3 Membership, application, and update

**lemma** *pfun-ext*: $[\![ \bigwedge x\ y.\ (x, y) \in_p f \longleftrightarrow (x, y) \in_p g ]\!] \Longrightarrow f = g$
**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*]: $x = y \Longrightarrow (f(x \mapsto v)_p)(y)_p = v$
**by** (*transfer*, *simp*)

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

**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** $x\ y :: 'a :: linorder$

**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*]: $x \notin pdom\ g \Longrightarrow (f - g)(x)_p = f(x)_p$
**by** (*transfer*, *auto simp add*: *map-minus-def*)

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

**lemma** *pfun-app-not-in-dom*:
$x \notin pdom(f) \Longrightarrow f(x)_p = undefined$
**by** (*transfer*, *simp*)

**lemma** *pfun-upd-minus* [*simp*]:
$x \notin pdom\ g \Longrightarrow (f - g)(x \mapsto v)_p = (f(x \mapsto v)_p - g)$
**by** (*transfer*, *auto simp add*: *map-minus-def*)

**lemma** *pdom-member-minus-iff* [*simp*]:
$x \notin pdom\ g \Longrightarrow x \in pdom(f - g) \longleftrightarrow x \in pdom(f)$
**by** (*transfer*, *simp add*: *domIff map-minus-def*)

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

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

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

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

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

## 9.4 Domain laws

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

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

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

**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*]: $pdom\ (g \circ_p f) = pdom\ (f \rhd_p pdom\ g)$
**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-pdom-res* [*simp*]: $pdom\ (A \lhd_p f) = A \cap pdom(f)$
**by** (*transfer*, *auto*)

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

**lemma** *pdom-pran-res-finite* [*simp*]:
$finite\ (pdom\ f) \Longrightarrow finite\ (pdom\ (f \rhd_p A))$
**by** (*transfer*, *auto*)

**lemma** *pdom-pfun-graph-finite* [*simp*]:
$finite\ (pdom\ f) \Longrightarrow finite\ (pfun\text{-}graph\ f)$
**by** (*transfer*, *simp add*: *finite-dom-graph*)

## 9.5 Range laws

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

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

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

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

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

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

## 9.6 Domain restriction laws

**lemma** *pdom-res-zero* [*simp*]: $A \lhd_p \{\}_p = \{\}_p$
**by** (*transfer*, *auto*)

**lemma** *pdom-res-empty* [*simp*]:
$(\{\} \lhd_p f) = \{\}_p$
**by** (*transfer*, *auto*)

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

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

**lemma** *pdom-res-upd-in* [*simp*]:

$k \in A \Longrightarrow A \lhd_p f(k \mapsto v)_p = (A \lhd_p f)(k \mapsto v)_p$
**by** (*transfer*, *auto*)

**lemma** *pdom-res-upd-out* [*simp*]:
$k \notin A \Longrightarrow A \lhd_p f(k \mapsto v)_p = A \lhd_p f$
**by** (*transfer*, *auto*)

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

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

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

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

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

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

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

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

## 9.7 Range restriction laws

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

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

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

**lemma** *pran-res-alt-def*: $f \rhd_p A = pId\text{-}on\ A \circ_p f$
**by** (*transfer*, *rule ext*, *auto simp add*: *ran-restrict-map-def*)

**lemma** *pran-res-override*: $(f + g) \rhd_p A \subseteq_p (f \rhd_p A) + (g \rhd_p A)$
**apply** (*transfer*, *auto simp add*: *map-add-def ran-restrict-map-def map-le-def*)
**apply** (*rename-tac f g A a y x*)
**apply** (*case-tac g a*)
**apply** (*auto*)
**done**

## 9.8 Graph laws

**lemma** *pfun-graph-inv*: *graph-pfun* (*pfun-graph f*) = *f*
  **by** (*transfer*, *simp*)

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

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

**lemma** *pfun-graph-minus*: *pfun-graph* (*f* − *g*) = *pfun-graph f* − *pfun-graph g*
  **by** (*transfer*, *simp add*: *map-graph-minus*)

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

## 9.9 Entries

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

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

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

## 9.10 Summation

**definition** *pfun-sum* :: (*'k*, *'v*::*comm-monoid-add*) *pfun* ⇒ *'v* **where**
*pfun-sum f* = *sum* (*pfun-app f*) (*pdom f*)

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

**lemma** *pfun-sum-upd-1*:
  **assumes** *finite*(*pdom*(*m*)) $k \notin$ *pdom*(*m*)
  **shows** *pfun-sum* $(m(k \mapsto v)_p)$ = *pfun-sum m* + *v*
  **by** (*simp-all add*: *pfun-sum-def assms*, *metis add.commute assms*(*2*) *pfun-app-upd-2 sum.cong*)

**lemma** *pfun-sums-upd-2*:
  **assumes** *finite*(*pdom*(*m*))
  **shows** *pfun-sum* $(m(k \mapsto v)_p)$ = *pfun-sum* $((- \{k\}) \lhd_p m)$ + *v*
**proof** (*cases* $k \notin$ *pdom*(*m*))
  **case** *True*
  **then show** *?thesis*
    **by** (*simp add*: *pfun-sum-upd-1 assms*)
**next**
  **case** *False*
  **then show** *?thesis*
    **using** *assms pfun-sum-upd-1*[*of* $((- \{k\}) \lhd_p m)$ *k v*]

**by** (*simp add*: *pfun-sum-upd-1*)
**qed**

**lemma** *pfun-sum-dom-res-insert* [*simp*]:
**assumes** $x \in pdom\ f\ x \notin A$ *finite A*
**shows** *pfun-sum* $((insert\ x\ A) \lhd_p f) = f(x)_p +$ *pfun-sum* $(A \lhd_p f)$
**using** *assms* **by** (*simp add*: *pfun-sum-def*)

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

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

Hide implementation details for partial functions

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

**end**

# 10 Finite Functions

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

## 10.1 Finite function type and operations

**typedef** ($'a$, $'b$) *ffun* = {$f$ :: ($'a$, $'b$) *pfun*. *finite*(*pdom*($f$))}
**morphisms** *pfun-of Abs-pfun*
**by** (*rule-tac* $x=\{\}_p$ **in** *exI*, *auto*)

**setup-lifting** *type-definition-ffun*

**lift-definition** *ffun-app* :: ($'a$, $'b$) *ffun* ⇒ $'a$ ⇒ $'b$ (-'(-')$_f$ [*999*,*0*] *999*) **is** *pfun-app* **.**

**lift-definition** *ffun-upd* :: ($'a$, $'b$) *ffun* ⇒ $'a$ ⇒ $'b$ ⇒ ($'a$, $'b$) *ffun* **is** *pfun-upd* **by** *simp*

**lift-definition** *fdom* :: ($'a$, $'b$) *ffun* ⇒ $'a$ *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** *op* $\in_p$ **.**

**lift-definition** *fdom-res* :: $'a$ *set* $\Rightarrow$ ($'a$, $'b$) *ffun* $\Rightarrow$ ($'a$, $'b$) *ffun* (**infixl** $\lhd_f$ *85*)
**is** *pdom-res* **by** *simp*

**lift-definition** *fran-res* :: ($'a$, $'b$) *ffun* $\Rightarrow$ $'b$ *set* $\Rightarrow$ ($'a$, $'b$) *ffun* (**infixl** $\rhd_f$ *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** *op* + **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** *op* $-$
  **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*] => $('a, 'b)$ *ffun* (-'(-')$_f$ [*900*,*0*]*900*)
*-Ffun* :: *maplets* => $('a, 'b)$ *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* => *-FfunUpd* (*CONST fempty*) *ms*
*-Ffun* (*-Maplets ms1 ms2*) <= *-FfunUpd* (*-Ffun ms1*) *ms2*
*-Ffun ms* <= *-FfunUpd* (*CONST fempty*) *ms*

## 10.2 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-minus-plus-commute*:
$fdom(g) \cap fdom(h) = \{\} \Longrightarrow (f - g) + h = (f + h) - g$
**by** (*transfer*, *simp add*: *pfun-minus-plus-commute*)

**lemma** *ffun-plus-minus*:
$f \subseteq_f g \Longrightarrow (g - f) + f = g$
**by** (*transfer*, *simp add*: *pfun-plus-minus*)

**lemma** *ffun-minus-common-subset*:
$[\![ h \subseteq_f f; h \subseteq_f g ]\!] \Longrightarrow (f - h = g - h) = (f = g)$
**by** (*transfer*, *simp add*: *pfun-minus-common-subset*)

**lemma** *ffun-minus-plus*:
$fdom(f) \cap fdom(g) = \{\} \Longrightarrow (f + g) - g = f$
**by** (*transfer*, *simp add*: *pfun-minus-plus*)

## 10.3 Membership, application, and update

**lemma** *ffun-ext*: $[\![ \bigwedge x\ y.\ (x, y) \in_f f \longleftrightarrow (x, y) \in_f g ]\!] \Longrightarrow f = g$
**by** (*transfer*, *simp add*: *pfun-ext*)

**lemma** *ffun-member-alt-def*:
$(x, y) \in_f f \longleftrightarrow (x \in fdom\ f \wedge f(x)_f = y)$
**by** (*transfer*, *simp add*: *pfun-member-alt-def*)

**lemma** *ffun-member-plus*:
$(x, y) \in_f f + g \longleftrightarrow ((x \notin fdom(g) \wedge (x, y) \in_f f) \vee (x, y) \in_f g)$
**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*]: $x = y \Longrightarrow (f(x \mapsto v)_f)(y)_f = v$
**by** (*transfer*, *simp*)

**lemma** *ffun-app-upd-2* [*simp*]: $x \neq y \Longrightarrow (f(x \mapsto v)_f)(y)_f = f(y)_f$
**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** $x\ y :: 'a :: linorder$
**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*]: $x \notin \mathit{fdom}\ g \Longrightarrow (f - g)(x)_f = f(x)_f$
  **by** (*transfer*, *auto*)

**lemma** *ffun-upd-minus* [*simp*]:
  $x \notin \mathit{fdom}\ g \Longrightarrow (f - g)(x \mapsto v)_f = (f(x \mapsto v)_f - g)$
  **by** (*transfer*, *auto*)

**lemma** *fdom-member-minus-iff* [*simp*]:
  $x \notin \mathit{fdom}\ g \Longrightarrow x \in \mathit{fdom}(f - g) \longleftrightarrow x \in \mathit{fdom}(f)$
  **by** (*transfer*, *simp*)

**lemma** *fsubseteq-ffun-upd1* [*intro*]:
  $[\![ f \subseteq_f g; x \notin \mathit{fdom}(g) ]\!] \Longrightarrow f \subseteq_f g(x \mapsto v)_f$
  **by** (*transfer*, *auto*)

**lemma** *fsubseteq-ffun-upd2* [*intro*]:
  $[\![ f \subseteq_f g; x \notin \mathit{fdom}(f) ]\!] \Longrightarrow f \subseteq_f g(x \mapsto v)_f$
  **by** (*transfer*, *auto*)

**lemma** *psubseteq-pfun-upd3* [*intro*]:
  $[\![ f \subseteq_f g; g(x)_f = v ]\!] \Longrightarrow f \subseteq_f g(x \mapsto v)_f$
  **by** (*transfer*, *auto*)

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

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

## 10.4 Domain laws

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

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

**lemma** *fdom-inter*: $\mathit{fdom}\ (f \cap_f g) \subseteq \mathit{fdom}\ f \cap \mathit{fdom}\ g$
  **by** (*transfer*, *meson pdom-inter*)

**lemma** *fdom-comp* [*simp*]: $\mathit{fdom}\ (g \circ_f f) = \mathit{fdom}\ (f \rhd_f \mathit{fdom}\ g)$
  **by** (*transfer*, *auto*)

**lemma** *fdom-upd* [*simp*]: $\mathit{fdom}\ (f(k \mapsto v)_f) = \mathit{insert}\ k\ (\mathit{fdom}\ f)$
  **by** (*transfer*, *simp*)

**lemma** *fdom-fdom-res* [*simp*]: $\mathit{fdom}\ (A \lhd_f f) = A \cap \mathit{fdom}(f)$
  **by** (*transfer*, *auto*)

**lemma** *fdom-graph-ffun* [*simp*]:
  $\mathit{finite}\ (\mathit{Domain}\ R) \Longrightarrow \mathit{fdom}\ (\mathit{graph\text{-}ffun}\ R) = \mathit{Domain}\ R$
  **by** (*transfer*, *simp add*: *Domain-fst graph-map-dom*)

## 10.5 Range laws

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

**lemma** *fran-upd* [*simp*]: *fran* $(f(k \mapsto v)_f) =$ *insert* $v$ (*fran* $((- \{k\}) \lhd_f f))$
**by** (*transfer*, *auto*)

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

**lemma** *fran-comp* [*simp*]: *fran* $(g \circ_f f) =$ *fran* (*fran* $f \lhd_f g$)
**by** (*transfer*, *auto*)

## 10.6 Domain restriction laws

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

**lemma** *pdom-res-upd-in* [*simp*]:
$k \in A \Longrightarrow A \lhd_f f(k \mapsto v)_f = (A \lhd_f f)(k \mapsto v)_f$
**by** (*transfer*, *auto*)

**lemma** *pdom-res-upd-out* [*simp*]:
$k \notin A \Longrightarrow A \lhd_f f(k \mapsto v)_f = A \lhd_f f$
**by** (*transfer*, *auto*)

**lemma** *fdom-res-override* [*simp*]: $A \lhd_f (f + g) = (A \lhd_f f) + (A \lhd_f g)$
**by** (*metis fdom-res.rep-eq pdom-res-override pfun-of-inject plus-ffun.rep-eq*)

**lemma** *fdom-res-minus* [*simp*]: $A \lhd_f (f - g) = (A \lhd_f f) - g$
**by** (*transfer*, *auto*)

**lemma** *fdom-res-swap*: $(A \lhd_f f) \rhd_f B = A \lhd_f (f \rhd_f B)$
**by** (*transfer*, *simp add*: *pdom-res-swap*)

**lemma** *fdom-res-twice* [*simp*]: $A \lhd_f (B \lhd_f f) = (A \cap B) \lhd_f f$
**by** (*transfer*, *auto*)

**lemma** *fdom-res-comp* [*simp*]: $A \lhd_f (g \circ_f f) = g \circ_f (A \lhd_f f)$
**by** (*transfer*, *simp*)

## 10.7 Range restriction laws

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

**lemma** *fran-res-upd-1* [*simp*]: $v \in A \Longrightarrow f(x \mapsto v)_f \rhd_f A = (f \rhd_f A)(x \mapsto v)_f$
**by** (*transfer*, *auto*)

**lemma** *fran-res-upd-2* [*simp*]: $v \notin A \Longrightarrow f(x \mapsto v)_f \rhd_f A = ((- \{x\}) \lhd_f f) \rhd_f A$
**by** (*transfer*, *auto*)

**lemma** *fran-res-override*: $(f + g) \rhd_f A \subseteq_f (f \rhd_f A) + (g \rhd_f A)$
**by** (*transfer*, *simp add*: *pran-res-override*)

### 10.8 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*: *ffun-graph* (*f* − *g*) = *ffun-graph f* − *ffun-graph g*
  **by** (*transfer*, *simp add*: *pfun-graph-minus*)

**lemma** *ffun-graph-inter*: *ffun-graph* ($f \cap_f g$) = *ffun-graph f* ∩ *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**

# 11 Infinity Supplement

**theory** *Infinity*
**imports** *Main Real*
  *~~/src/HOL/Library/Infinite-Set*
  *Optics.Two*
**begin**

This theory introduces a type class *infinite* that guarantees that the underlying universe of the type is infinite. It also provides useful theorems to prove infinity of the universes for various HOL types.

### 11.1 Type class *infinite*

The type class postulates that the universe (carrier) of a type is infinite.

**class** *infinite* =
  **assumes** *infinite-UNIV* [*simp*]: *infinite* (*UNIV* :: *'a set*)

### 11.2 Infinity Theorems

Useful theorems to prove that a type's *UNIV* is infinite.

Note that *infinite-UNIV-nat* is already a simplification rule by default.

**lemmas** *infinite-UNIV-int* [*simp*]

**theorem** *infinite-UNIV-real* [*simp*]:
*infinite* (*UNIV* :: *real set*)
  **by** (*rule infinite-UNIV-char-0*)

**theorem** *infinite-UNIV-fun1* [*simp*]:
*infinite* (*UNIV* :: *'a set*) ⟹
 *card* (*UNIV* :: *'b set*) ≠ *Suc 0* ⟹
 *infinite* (*UNIV* :: (*'a* ⇒ *'b*) *set*)
  **apply** (*erule contrapos-nn*)

**apply** (*erule finite-fun-UNIVD1*)
**apply** (*assumption*)
**done**

**theorem** *infinite-UNIV-fun2* [*simp*]:
*infinite* (*UNIV* :: *'b set*) $\Longrightarrow$
*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*) $\Longrightarrow$
*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*) $\Longrightarrow$
*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*) $\Longrightarrow$
*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*) $\Longrightarrow$
*infinite* (*UNIV* :: (*'a* + *'b*) *set*)
**apply** (*erule contrapos-nn*)
**apply** (*simp*)
**done**

**theorem** *infinite-UNIV-sum2* [*simp*]:
*infinite* (*UNIV* :: *'b set*) $\Longrightarrow$
*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*) $\Longrightarrow$
*infinite* (*UNIV* :: *'a option set*)
**apply** (*erule contrapos-nn*)

**apply** (*simp*)
**done**

**theorem** *infinite-image* [*intro*]:
*infinite* $A \Longrightarrow$ *inj-on* $f$ $A \Longrightarrow$ *infinite* ($f$ ‘ $A$)
**apply** (*metis finite-imageD*)
**done**

**theorem** *infinite-transfer* :
*infinite* $B \Longrightarrow B \subseteq f$ ‘ $A \Longrightarrow$ *infinite* $A$
**using** *infinite-super*
**apply** (*blast*)
**done**

## 11.3 Instantiations

The instantiations for product and sum types have stronger caveats than in principle needed. Namely, it would be sufficient for one type of a product or sum to be infinite. A corresponding rule, however, cannot be formulated using type classes. Generally, classes are not entirely adequate for the purpose of deriving the infinity of HOL types, which is perhaps why a class such as *infinite* was omitted from the Isabelle/HOL library.

**instance** *nat* :: *infinite* **by** (*intro-classes*, *simp*)
**instance** *int* :: *infinite* **by** (*intro-classes*, *simp*)
**instance** *real* :: *infinite* **by** (*intro-classes*, *simp*)
**instance** *fun* :: (*type*, *infinite*) *infinite* **by** (*intro-classes*, *simp*)
**instance** *set* :: (*infinite*) *infinite* **by** (*intro-classes*, *simp*)
**instance** *prod* :: (*infinite*, *infinite*) *infinite* **by** (*intro-classes*, *simp*)
**instance** *sum* :: (*infinite*, *infinite*) *infinite* **by** (*intro-classes*, *simp*)
**instance** *list* :: (*type*) *infinite* **by** (*intro-classes*, *simp*)
**instance** *option* :: (*infinite*) *infinite* **by** (*intro-classes*, *simp*)

**subclass** (**in** *infinite*) *two* **by** (*intro-classes*, *auto*)

**end**

# 12 Positive Subtypes

**theory** *Positive*
**imports**
*Infinity*
*HOL−Library.Countable*
**begin**

## 12.1 Type Definition

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

**syntax**
*-type-pos* :: *type* $\Rightarrow$ *type* ($\text{-}^{+}$ [*999*] *999*)

**translations**

(*type*) $'a^+$ == (*type*) *'a pos*

**setup-lifting** *type-definition-pos*

**type-synonym** *preal* = *real pos*

## 12.2 Operators

**lift-definition** *mk-pos* :: *'a*::{*zero*, *linorder*} ⇒ *'a pos* **is**
λ *n*. *if* (*n* ≥ *0*) *then n else 0* **by** *auto*

**lift-definition** *real-of-pos* :: *real pos* ⇒ *real* **is** *id* **.**

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

## 12.3 Instantiations

**instantiation** *pos* :: ({*zero*, *linorder*}) *zero*
**begin**
  **lift-definition** *zero-pos* :: *'a pos*
    **is** *0* :: *'a* **..**
  **instance ..**
**end**

**instantiation** *pos* :: ({*zero*, *linorder*}) *linorder*
**begin**
  **lift-definition** *less-eq-pos* :: *'a pos* ⇒ *'a pos* ⇒ *bool*
    **is** *op* ≤ :: *'a* ⇒ *'a* ⇒ *bool* **.**
  **lift-definition** *less-pos* :: *'a pos* ⇒ *'a pos* ⇒ *bool*
    **is** *op* < :: *'a* ⇒ *'a* ⇒ *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* ∀ *x*::*'a pos*. *x* ≤ *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* ⇒ *'a pos* ⇒ *'a pos*

**is** *op* + **by** (*simp*)
**lift-definition** *minus-pos* :: $'a$ *pos* ⇒ $'a$ *pos* ⇒ $'a$ *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* ⇒ $'a$ *pos* ⇒ $'a$ *pos*
**is** *op* ∗ **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* ⇒ $'a$ *pos* ⇒ $'a$ *pos*
**is** *op div* **by** (*simp*)
**instance**
**apply** (*intro-classes*; *transfer*)
**apply** (*simp-all*)
**done**
**end**

**instantiation** *pos* :: (*linordered-field*) *inverse*
**begin**
**lift-definition** *inverse-pos* :: $'a$ *pos* ⇒ $'a$ *pos*
**is** *inverse* **by** (*simp*)
**instance** ..
**end**

**lemma** *pos-positive* [*simp*]: $0 \leq (x::'a::\{zero,linorder\}\ pos)$
**by** (*transfer*, *simp*)

## 12.4 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*:

$[\![\ 0 \le x;\ x \le y\ ]\!] \Longrightarrow mk\text{-}pos\ x \le mk\text{-}pos\ y$
**by** (*transfer*, *auto*)

**lemma** *mk-pos-less*:
$[\![\ 0 \le x;\ x < y\ ]\!] \Longrightarrow mk\text{-}pos\ x < mk\text{-}pos\ y$
**by** (*transfer*, *auto*)

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

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

## 12.5 Transfer to Reals

**named-theorems** *pos-transfer*

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

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

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

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

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

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

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

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

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

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

**end**

# 13 Recall Undeclarations

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

## 13.1 ML File Import

**ML-file** *Total-Recall.ML*

## 13.2 Outer Commands

```
ML ‹
  val - =
    Outer-Syntax.command @{command-keyword purge-syntax}
      purge raw syntax clauses
      ((Parse.syntax-mode -- Scan.repeat1 Parse.const-decl) >>
        (Toplevel.theory o (fn (mode, args) =>
          (TotalRecall.record-no-syntax mode args) o
          (Sign.del-syntax-cmd mode args))));

  val - =
    Outer-Syntax.local-theory @{command-keyword purge-notation}
      purge concrete syntax for constants / fixed variables
      ((Parse.syntax-mode -- Parse.and-list1 (Parse.const -- Parse.mixfix)) >>
        (fn (mode, args) =>
          (Local-Theory.background-theory
            (TotalRecall.record-no-notation mode args)) o
          (Specification.notation-cmd false mode args)));

  val - =
    Outer-Syntax.command @{command-keyword recall-syntax}
      recall undecarations of all purged items
      (Scan.succeed (Toplevel.theory TotalRecall.execute-all))
›
end
```

# 14 Injection Universes

```
theory Injection-Universe
  imports
    HOL-Library.Countable
    Optics.Lenses
begin
```

An injection universe shows how one type $'a$ can be injected into another type, $'u$. They are applied in UTP to provide local variables which require that we can injection a variety of different datatypes into a unified stack type.

**record** $('a, 'u)$ *inj-univ* =
  *to-univ* :: $'a \Rightarrow 'u$ $(to{-}univ\imath)$

**locale** *inj-univ* =
  **fixes** $I$ :: $('a, 'u)$ *inj-univ* (**structure**)
  **assumes** *inj-to-univ*: *inj* $to{-}univ$
**begin**

**definition** *from-univ* :: $'u \Rightarrow 'a$ $(from{-}univ)$ **where**
*from-univ* = *inv* $to{-}univ$

**lemma** *to-univ-inv* [*simp*]: $from{-}univ\ (to{-}univ\ x) = x$
  **by** (*simp add*: *from-univ-def inv-f-f inj-to-univ*)

Lens-based view on universe injection and projection.

**definition** *to-univ-lens* :: $'a \Longrightarrow 'u$ $(to{-}univ_L)$ **where**
*to-univ-lens* = $(\!|$ *lens-get* = $from{-}univ$, *lens-put* = $(\lambda\ s\ v.\ to{-}univ\ v)$ $|\!)$

**lemma** *mwb-to-univ-lens* [*simp*]:
  *mwb-lens to-univ-lens*
  **by** (*unfold-locales*, *simp-all add*: *to-univ-lens-def*)

**end**

Example universe based on natural numbers. Any countable type can be injected into it.

**definition** *nat-inj-univ* :: $('a$::*countable*, *nat*$)$ *inj-univ* $(\mathcal{U}_{\mathbb{N}})$ **where**
*nat-inj-univ* = $(\!|$ *to-univ* = *to-nat* $|\!)$

**lemma** *nat-inj-univ*: *inj-univ nat-inj-univ*
  **by** (*unfold-locales*, *simp add*: *nat-inj-univ-def*)

**end**

# 15 Trace Algebras

**theory** *Trace-Algebra*
  **imports**
    *List-Extra*
    *Positive*
**begin**

Trace algebras provide a useful way in the UTP of characterising different notions of trace history. They can characterise notions as diverse as discrete event sequences and piecewise continuous functions, as employed by hybrid systems. For more information, please see our journal publication [3].

## 15.1 Ordered Semigroups

**class** *ordered-semigroup* = *semigroup-add* + *order* +
  **assumes** *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
  **and** *add-right-mono*: $a \leq b \Longrightarrow a + c \leq b + c$
**begin**

**lemma** *add-mono*:

$a \le b \Longrightarrow c \le d \Longrightarrow a + c \le b + d$
**using** *local.add-left-mono local.add-right-mono local.order.trans* **by** *blast*

**end**

## 15.2 Monoid Subclasses

**class** *left-cancel-monoid* = *monoid-add* +
**assumes** *add-left-imp-eq*: $a + b = a + c \Longrightarrow b = c$

**class** *right-cancel-monoid* = *monoid-add* +
**assumes** *add-right-imp-eq*: $b + a = c + a \Longrightarrow b = c$

**class** *monoid-sum-0* = *monoid-add* +
**assumes** *zero-sum-left*: $a + b = 0 \Longrightarrow a = 0$
**begin**

**lemma** *zero-sum-right*: $a + b = 0 \Longrightarrow b = 0$
**by** (*metis local.add-0-left local.zero-sum-left*)

**lemma** *zero-sum*: $a + b = 0 \longleftrightarrow a = 0 \wedge b = 0$
**by** (*metis local.add-0-right zero-sum-right*)

**end**

**context** *monoid-add*
**begin**

An additive monoid gives rise to natural notions of order, which we here define.

**definition** *monoid-le* (**infix** $\le_m$ *50*)
**where** $a \le_m b \longleftrightarrow (\exists c.\ b = a + c)$

We can also define a subtraction operator that remove a prefix from a monoid, if possible.

**definition** *monoid-subtract* (**infixl** $-_m$ *65*)
**where** $a -_m b = (\mathit{if}\ (b \le_m a)\ \mathit{then}\ \mathit{THE}\ c.\ a = b + c\ \mathit{else}\ 0)$

**end**

## 15.3 Trace Algebras

A pre-trace algebra is based on a left-cancellative monoid with the additional property that plus has no additive inverse. The latter is required to ensure that there are no "negative traces". A pre-trace algebra has all the trace algebra axioms, but does not export the definitions of *op* $\le$ and *op* $-$.

**class** *pre-trace* = *left-cancel-monoid* + *monoid-sum-0* +
**assumes**
*sum-eq-sum-conv*: $(a + b) = (c + d) \Longrightarrow \exists\ e\ .\ a = c + e \wedge e + b = d \vee a + e = c \wedge b = e + d$
— $?a + ?b = ?c + ?d \Longrightarrow \exists e.\ ?a = ?c + e \wedge e + ?b = ?d \vee ?a + e = ?c \wedge ?b = e + ?d$ shows how two equal traces that are each composed of two subtraces, can be expressed in terms of each other.
**begin**

From our axiom set, we can derive a variety of properties of the monoid order

**lemma** *monoid-le-least-zero*: $0 \le_m a$
**by** (*simp add*: *monoid-le-def*)

**lemma** *monoid-le-refl*: $a \leq_m a$
  **by** (*simp add*: *monoid-le-def*, *metis add.right-neutral*)

**lemma** *monoid-le-trans*: $\llbracket\ a \leq_m b;\ b \leq_m c\ \rrbracket \Longrightarrow a \leq_m c$
  **by** (*metis add.assoc monoid-le-def*)

**lemma** *monoid-le-antisym*:
  **assumes** $a \leq_m b$ $b \leq_m a$
  **shows** $a = b$
**proof** −
  **obtain** $a'$ **where** $a'$: $b = a + a'$
    **using** *assms*(*1*) *monoid-le-def* **by** *auto*

  **obtain** $b'$ **where** $b'$: $a = b + b'$
    **using** *assms*(*2*) *monoid-le-def* **by** *auto*

  **have** $b' = (b' + a' + b')$
    **by** (*metis* $a'$ *add-assoc* $b'$ *local.add-left-imp-eq*)

  **hence** $a' + b' = 0$
    **by** (*metis add-assoc local.add-0-right local.add-left-imp-eq*)

  **hence** $a' = 0$ $b' = 0$
    **by** (*simp add*: *zero-sum*)+

  **with** $a'$ $b'$ **show** *?thesis*
    **by** *simp*
**qed**

**lemma** *monoid-le-add*: $a \leq_m a + b$
  **by** (*auto simp add*: *monoid-le-def*)

**lemma** *monoid-le-add-left-mono*: $a \leq_m b \Longrightarrow c + a \leq_m c + b$
  **using** *add-assoc* **by** (*auto simp add*: *monoid-le-def*)

The monoid minus operator is also the inverse of plus in this context, as expected.

**lemma** *add-monoid-diff-cancel-left* [*simp*]: $(a + b) -_m a = b$
  **apply** (*simp add*: *monoid-subtract-def monoid-le-add*)
  **apply** (*rule the-equality*)
   **apply** (*simp*)
  **using** *local.add-left-imp-eq* **apply** *blast*
  **done**

Iterating a trace

**fun** *tr-iter* :: *nat* ⇒ *'a* ⇒ *'a* **where**
*tr-iter-0*: *tr-iter* *0* $t$ = *0* |
*tr-iter-Suc*: *tr-iter* (*Suc* $n$) $t$ = *tr-iter* $n$ $t$ + $t$

**lemma** *tr-iter-empty* [*simp*]: *tr-iter* $m$ *0* = *0*
  **by** (*induct* $m$, *simp-all*)

**end**

We now construct the trace algebra by also exporting the order and minus operators.

**class** *trace* = *pre-trace* + *ord* + *minus* +
  **assumes** *le-is-monoid-le*: $a \leq b \longleftrightarrow (a \leq_m b)$
  **and** *less-iff*: $a < b \longleftrightarrow a \leq b \wedge \neg (b \leq a)$
  **and** *minus-def*: $a - b = a -_m b$
**begin**

Next we prove all the trace algebra lemmas.

  **lemma** *le-iff-add*: $a \leq b \longleftrightarrow (\exists\ c.\ b = a + c)$
    **by** (*simp add*: *local.le-is-monoid-le local.monoid-le-def*)

  **lemma** *least-zero* [*simp*]: $0 \leq a$
    **by** (*simp add*: *local.le-is-monoid-le local.monoid-le-least-zero*)

  **lemma** *le-add* [*simp*]: $a \leq a + b$
    **by** (*simp add*: *le-is-monoid-le local.monoid-le-add*)

  **lemma** *not-le-minus* [*simp*]: $\neg (a \leq b) \Longrightarrow b - a = 0$
    **by** (*simp add*: *le-is-monoid-le local.minus-def local.monoid-subtract-def*)

  **lemma** *add-diff-cancel-left* [*simp*]: $(a + b) - a = b$
    **by** (*simp add*: *minus-def*)

  **lemma** *diff-zero* [*simp*]: $a - 0 = a$
    **by** (*metis local.add-0-left local.add-diff-cancel-left*)

  **lemma** *diff-cancel* [*simp*]: $a - a = 0$
    **by** (*metis local.add-0-right local.add-diff-cancel-left*)

  **lemma** *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
    **by** (*simp add*: *local.le-is-monoid-le local.monoid-le-add-left-mono*)

  **lemma** *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
    **by** (*auto simp add*: *le-iff-add*, *metis add-assoc local.add-diff-cancel-left*)

  **lemma** *add-diff-cancel-left'* [*simp*]: $(c + a) - (c + b) = a - b$
  **proof** (*cases* $b \leq a$)
    **case** *True* **thus** *?thesis*
      **by** (*metis add-assoc local.add-diff-cancel-left local.le-iff-add*)
  **next**
    **case** *False* **thus** *?thesis*
      **using** *local.add-le-imp-le-left not-le-minus* **by** *blast*
  **qed**

  **lemma** *minus-zero-eq*: $[\![\ b \leq a;\ a - b = 0\ ]\!] \Longrightarrow a = b$
    **using** *local.le-iff-add local.monoid-le-def* **by** *auto*

  **lemma** *diff-add-cancel-left'*: $a \leq b \Longrightarrow a + (b - a) = b$
    **using** *local.le-iff-add local.monoid-le-def* **by** *auto*

  **lemma** *add-left-strict-mono*: $[\![\ a + b < a + c\ ]\!] \Longrightarrow b < c$
    **using** *local.add-le-imp-le-left local.add-left-mono local.less-iff* **by** *blast*

  **lemma** *sum-minus-left*: $c \leq a \Longrightarrow (a + b) - c = (a - c) + b$
    **by** (*metis add-assoc diff-add-cancel-left' local.add-monoid-diff-cancel-left local.minus-def*)

**lemma** *neq-zero-impl-greater*:
$x \neq 0 \Longrightarrow 0 < x$
**using** *le-is-monoid-le less-iff monoid-le-antisym monoid-le-least-zero* **by** *auto*

**lemma** *minus-cancel-le*:
$[\![ x \leq y; y \leq z ]\!] \Longrightarrow y - x \leq z - x$
**using** *add-assoc le-iff-add* **by** *auto*

The set subtraces of a common trace $c$ is totally ordered.

**lemma** *le-common-total*: $[\![ a \leq c; b \leq c ]\!] \Longrightarrow a \leq b \vee b \leq a$
**by** (*metis diff-add-cancel-left′ le-add local.sum-eq-sum-conv*)

**lemma** *le-sum-cases*: $a \leq b + c \Longrightarrow a \leq b \vee b \leq a$
**by** (*simp add*: *le-common-total*)

**lemma** *le-sum-cases′*:
$a \leq b + c \Longrightarrow a \leq b \vee b \leq a \wedge a - b \leq c$
**by** (*auto*, *metis le-sum-cases*, *metis minus-def le-is-monoid-le add-monoid-diff-cancel-left monoid-le-def sum-eq-sum-conv*)

**lemma** *le-sum-iff*: $a \leq b + c \longleftrightarrow a \leq b \vee b \leq a \wedge a - b \leq c$
**by** (*metis le-sum-cases′ add-monoid-diff-cancel-left le-is-monoid-le minus-def monoid-le-add-left-mono monoid-le-def monoid-le-trans*)

**lemma** *sum-minus-right*: $c \geq a \Longrightarrow a + b - c = b - (c - a)$
**by** (*metis diff-add-cancel-left′ local.add-diff-cancel-left′*)

**lemma** *minus-gr-zero-iff* [*simp*]:
$0 < x - y \longleftrightarrow y < x$
**by** (*metis diff-cancel le-is-monoid-le least-zero less-iff minus-zero-eq monoid-le-antisym not-le-minus*)

**lemma** *le-zero-iff* [*simp*]: $x \leq 0 \longleftrightarrow x = 0$
**using** *local.le-iff-add local.zero-sum* **by** *auto*

**lemma** *minus-assoc* [*simp*]: $x - y - z = x - (y + z)$
**by** (*metis local.add-diff-cancel-left′ local.diff-add-cancel-left′ local.le-add local.le-sum-iff local.not-le-minus local.zero-sum-right*)

**end**

Trace algebra give rise to a partial order on traces.

**instance** *trace* ⊆ *order*
**apply** (*intro-classes*)
**apply** (*simp-all add*: *less-iff le-is-monoid-le monoid-le-refl*)
**using** *monoid-le-trans* **apply** *blast*
**apply** (*simp add*: *monoid-le-antisym*)
**done**

## 15.4 Models

Lists form a trace algebra.

**instantiation** *list* :: (*type*) *monoid-add*
**begin**

**definition** *zero-list* :: *′a list* **where** *zero-list* = []

**definition** *plus-list* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where** *plus-list* = *op* @

**instance**
**by** (*intro-classes*, *simp-all add*: *zero-list-def plus-list-def*)

**end**

**lemma** *monoid-le-list*:
$(xs :: 'a\ list) \leq_m ys \longleftrightarrow xs \leq ys$
**apply** (*simp add*: *monoid-le-def plus-list-def*)
**using** *Prefix-Order.prefixE Prefix-Order.prefixI* **apply** *blast*
**done**

**lemma** *monoid-subtract-list*:
$(xs :: 'a\ list) -_m ys = xs - ys$
**apply** (*auto simp add*: *monoid-subtract-def monoid-le-list minus-list-def less-eq-list-def*)
**apply** (*rule the-equality*)
**apply** (*simp-all add*: *zero-list-def plus-list-def prefix-drop*)
**done**

**instance** *list* :: (*type*) *trace*
**apply** (*intro-classes*, *simp-all add*: *zero-list-def plus-list-def monoid-le-def monoid-subtract-list*)
**apply** (*simp add*: *append-eq-append-conv2*)
**using** *Prefix-Order.prefixE Prefix-Order.prefixI* **apply** *blast*
**apply** (*simp add*: *less-list-def*)
**done**

**lemma** *monoid-le-nat*:
$(x :: nat) \leq_m y \longleftrightarrow x \leq y$
**by** (*simp add*: *monoid-le-def nat-le-iff-add*)

**lemma** *monoid-subtract-nat*:
$(x :: nat) -_m y = x - y$
**by** (*auto simp add*: *monoid-subtract-def monoid-le-nat*)

**instance** *nat* :: *trace*
**apply** (*intro-classes*, *simp-all add*: *monoid-subtract-nat*)
**apply** (*metis Nat.diff-add-assoc Nat.diff-add-assoc2 add-diff-cancel-right' add-le-cancel-left add-le-cancel-right add-less-mono cancel-ab-semigroup-add-class.add-diff-cancel-left' less-irrefl not-le*)
**apply** (*simp add*: *nat-le-iff-add monoid-le-def*)
**apply** *linarith+*
**done**

Positives form a trace algebra.

**instance** *pos* :: (*linordered-semidom*) *trace*
**proof** (*intro-classes*, *simp-all*)
**fix** *a b c d* :: *'a pos*
**show** $a + b = 0 \Longrightarrow a = 0$
**by** (*transfer*, *simp add*: *add-nonneg-eq-0-iff*)
**show** $a + b = c + d \Longrightarrow \exists e.\ a = c + e \wedge e + b = d \vee a + e = c \wedge b = e + d$
**apply** (*cases* $c \leq a$)
**apply** (*metis* (*no-types*, *lifting*) *cancel-semigroup-add-class.add-left-imp-eq le-add-diff-inverse semiring-normalization-*
**apply** (*metis* (*no-types*, *lifting*) *cancel-semigroup-add-class.add-left-imp-eq less-imp-le linordered-semidom-class.add-di*
*semiring-normalization-rules*(*21*))
**done**

**show** $(a < b) = (a \leq b \wedge \neg\ b \leq a)$
  **by** *auto*
**show** *le-def*: $\bigwedge\ a\ b :: {'a}\ pos.\ (a \leq b) = (a \leq_m b)$
  **by** (*auto simp add*: *monoid-le-def*, *metis le-add-diff-inverse*)
**show** $a - b = a -_m b$
  **apply** (*auto simp add*: *monoid-subtract-def le-def*[*THEN sym*])
   **apply** (*rule sym*)
   **apply** (*rule the-equality*)
    **apply** (*simp-all*)
  **apply** (*transfer*, *simp*)
  **done**
**qed**

**end**

# 16 Meta-theory for UTP Toolkit

**theory** *utp-toolkit*
  **imports**
  *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*
  *Lens-Extra*
  *Countable-Set-Extra*
  *FSet-Extra*
  *Map-Extra*
  *List-Extra*
  *List-Lexord-Alt*
  *Partial-Fun*
  *Finite-Fun*
  *Infinity*
  *Positive*
  *Total-Recall*
  *Injection-Universe*
  *Trace-Algebra*
**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, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying theories of time with generalised reactive processes. *Accepted for Information Processing Letters*, Dec 2017. Preprint: https://arxiv.org/abs/1712.10213.

[4] S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In *UTP*, LNCS 8963, pages 21–41. Springer, 2014.

[5] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In *ICTAC*, LNCS 9965. Springer, 2016.

[6] T. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[7] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1998.

[8] F. Zeyda, S. Foster, and L. Freitas. An axiomatic value model for Isabelle/UTP. In *UTP*, LNCS 10134. Springer, 2016.