Z Mathematical Toolkit in Isabelle/HOL

Simon Foster, Pedro Ribeiro, Frank Zeyda, and Jim Woodcock
University of York, UK

simon.foster@york.ac.uk

October 17, 2025

Abstract

The objective of this theory development is an implementation of the Z mathematical toolkit in Isabelle/HOL that is both efficient for proof and faithful to the standard. We construct the Z metalanguage and type universe on top of HOL, and link this to corresponding concrete types (finite functions, lists etc.) in Isabelle, to enable efficient proof automation. We then utilise coercive subtyping and overloading to support processing of Z-like expressions in Isabelle/HOL. We then use this to develop the mathematical toolkit for sets, relations, functions, and sequences.

Contents

1 Overview

The objective of this theory development is an implementation of the Z mathematical toolkit [1] (ISO 13568:2002[1]) that is both efficient for proof and faithful to the standard.

The main challenge to overcome is a mismatch between the type system of Z, and the way that Isabelle/HOL theories are typically developed. This is because the objectives of Z and HOL are a little different: Z targets a mathematically pure foundational development for formal specification based on ZF set theory, whereas HOL targets an efficient proof system capable of scalable verification. The aim then is to reconcile these two objectives in one development.

In Z, the type system is very simple, consisting of given types closed under powerset and product constructions. For example, in Z a total function is encoded as its graph in a relation, and a relation is simply a set of pairs. There is no distinct type constructor for functions. Similarly, a sequence (list in HOL) is a finite function whose domain is $\{1..n\}$, for some natural number n. This means in Z, we can write expressions that compare a relation and sequence, since they have the same type.

In contrast, in HOL it is impossible to directly compare a relation and a list since they have distinct type constructors, and only values of the same types can typically be related. It is necessary to insert explicit coercions between values of different types in this case. Indeed, the dominant paradigm for theory development in HOL is to constantly extend the type system to capture new mathematical concepts, such as vectors, bounded continuous functions, and physical quantities, to name a few examples. This approach has proven to be very successful and scalable, as evidenced by large verification projects like seL4, and the ever growing Archive of Formal Proofs[2] (AFP).

Now, it is entirely possible to reconstruct the Z mathematical toolkit in the way described above, following the ISO standard, such that everything boils down to sets. However, there is a major downside to this, which is that we cannot easily use the results in the HOL standard library (*Main*) and the AFP, since these are all built using the HOL type universe extension paradigm. There are also several benefits to the HOL approach, notably that the type system can be used to deduce when a function is closed under a set. This in turn greatly improves proof automation, since there is no obligation to check well-formedness of expressions as part of the proof. Consequently, we chose to stick with the HOL approach.

However, in order to be faithful with Z, we also implement the Z universe as a set of definitions, based on the ISO standard. Much of this already in implemented in the theory *HOL.Relation*, but we extend it with functions

[1] Z formal specification notation. https://www.iso.org/standard/21573.html
[2] Archive of Formal Proofs. http://www.isa-afp.org

like application, domain restriction, and overriding, which are all part of the Z meta-language. Crucially, this development is all based on sets and relations, not HOL functions, and therefore is a faithful encoding with Z. Upon this foundation, we construct a hierarchy of types corresponding to partial functions, finite functions, and total functions, and we reuse the HOL *'a list* type. We then prove that every HOL typed construction can be safely converted into a Z-like set-based construction, which provides the link between the efficiently implement HOL functions, and their Z counterparts.

In order to achieve compatibility between this HOL type hierarchy, and the Z mathematical toolkit, the principle problem to solve is the necessity of type coercions. As mentioned, in Z, sequences are subtypes of sets, and so set-based functions can be directly applied to functions, which is often beneficial. For example, the domain of a sequence is the set of indices of that sequence. So the technical goal is to allow HOL to accept expressions of this kind. Our solution is to use a mixture of coercive subtyping and type overloading to achieve this. This allows the user to write Z expressions into Isabelle, which are then internally mapped into HOL expressions.

There are basically two types of situation we need to capture. The first is the use of a more abstract type (e.g. set) to act as a view on a more concrete type (e.g. a sequence). Thus we can find the length of a list by asking for its cardinality. The second is the composition of two abstract types. For example, concatenation of two sequences always results in a sequence, which is readily the case in HOL. There are more subtle examples, for example we can take union of two partial functions with disjoint domains, and produce a new partial function. In Z, we would need to prove this, whereas with HOL's type system this can be deduced by construction.

The first of these situations can be captured by coercive subtyping. In HOL, we can create a function between a concrete type $'c$ and an abstract type $'a$, and request that whenever a value of type $'a$ is required, but $'c$ is present, then the coercion f is inserted automatically during type inference. The second situation can be solve by overloading the operators in Z that are essentially polymorphic. Union is a good example: in HOL we can create a polymorphic function symbol with different implementations for different types. Thus, we can take the union of two partial functions, under the aforementioned disjointness conditions. This development therefore implements the Z toolkit in this way.

In conclusion, we wish to retain the generality of Z, whilst also taking full advantage of the automation afforded by Isabelle/HOL. We hope our theory development achieves this.

2 Lists: extra functions and properties

theory *List-Extra*

imports
HOL−Library.Sublist
HOL−Library.Monad-Syntax
HOL−Library.Prefix-Order
Optics.Lens-Instances
begin

2.1 Useful Abbreviations

abbreviation *list-sum xs* ≡ *foldr* (+) *xs 0*

2.2 Sets

lemma *set-Fpow* [*simp*]: *set xs* ∈ *Fpow A* ⟷ *set xs* ⊆ *A*
by (*auto simp add*: *Fpow-def*)

lemma *Fpow-as-Pow*: *finite A* ⟹ *Fpow A* = *Pow A*
by (*auto simp add*: *Pow-def Fpow-def finite-subset*)

lemma *Fpow-set* [*code*]:
Fpow (*set* []) = {{}}
Fpow (*set* (*x* # *xs*)) = (*let A* = *Fpow* (*set xs*) *in A* ∪ *insert x* ‘ *A*)
by (*simp-all add*: *Fpow-as-Pow Pow-set del*: *set-simps*)

2.3 Folds

context *abel-semigroup*
begin

lemma *foldr-snoc*: *foldr* (∗) (*xs* • [*x*]) *k* = (*foldr* (∗) *xs k*) ∗ *x*
by (*induct xs*, *simp-all add*: *commute left-commute*)

end

2.4 List Lookup

The following variant of the standard *nth* function returns ⊥ if the index is out of range.

primrec
nth-el :: *'a list* ⇒ *nat* ⇒ *'a option* (-⟨-⟩$_l$ [*90*, *0*] *91*)
where
[]⟨*i*⟩$_l$ = *None*
| (*x* # *xs*)⟨*i*⟩$_l$ = (*case i of 0* ⇒ *Some x* | *Suc j* ⇒ *xs* ⟨*j*⟩$_l$)

lemma *nth-el-appendl*[*simp*]: *i* < *length xs* ⟹ (*xs* • *ys*)⟨*i*⟩$_l$ = *xs*⟨*i*⟩$_l$
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*
then show *?thesis* **by** *simp*
next
case (*Suc nat*)
with *Cons* **show** *?thesis* **by** *simp*
qed
qed

lemma *nth-el-appendr*[*simp*]: *length xs* ≤ *i* ⟹ (*xs* • *ys*)⟨*i*⟩$_l$ = *ys*⟨*i* − *length xs*⟩$_l$
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*
with *Cons* **show** *?thesis*
by *fastforce*
next
case (*Suc nat*)
with *Cons* **show** *?thesis* **by** *simp*
qed
qed

2.5 Extra List Theorems

2.5.1 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*) ∘ *Suc*
by *auto*
hence *map* (*nth xs*) [*m*..<*n* − *1*] = *map* (*nth* (*x* # *xs*) ∘ *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

2.5.2 Sorted Lists

lemma *sorted-last* [*simp*]: ⟦ *x* ∈ *set xs*; *sorted xs* ⟧ ⟹ *x* ≤ *last xs*

by (*induct xs*, *auto*)

lemma *sorted-prefix*:
assumes $xs \leq ys$ *sorted ys*
shows *sorted xs*
proof −
obtain *zs* **where** $ys = xs \bullet zs$
using *Prefix-Order.prefixE assms*(*1*) **by** *auto*
thus *?thesis*
using *assms*(*2*) *sorted-append* **by** *blast*
qed

lemma *sorted-map*: ⟦ *sorted xs*; *mono f* ⟧ ⟹ *sorted* (*map f xs*)
by (*simp add*: *monoD sorted-iff-nth-mono*)

lemma *sorted-distinct* [*intro*]: ⟦ *sorted* (*xs*); *distinct*(*xs*) ⟧ ⟹ (∀ *i*<*length xs* − *1*. *xs*!*i* < *xs*!(*i* + *1*))
apply (*induct xs*)
apply (*simp-all*)
apply (*metis* (*no-types*, *opaque-lifting*) *Suc-leI Suc-less-eq Suc-pred gr0-conv-Suc not-le not-less-iff-gr-or-eq nth-Cons-Suc nth-mem nth-non-equal-first-eq*)
done

The concatenation of two lists is sorted provided (1) both the lists are sorted, and (2) the final and first elements are ordered.

lemma *sorted-append-middle*:
sorted(*xs*•*ys*) = (*sorted xs* ∧ *sorted ys* ∧ (*xs* ≠ [] ∧ *ys* ≠ [] ⟶ *xs*!(*length xs* − *1*) ≤ *ys*!*0*))
proof −
have ⋀*x y*. ⟦ *sorted xs*; *sorted ys*; *xs* ! (*length xs* − *Suc 0*) ≤ *ys* ! *0* ⟧ ⟹ *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ *x* ≤ *y*
proof −
fix *x y*
assume *sorted xs sorted ys xs* ! (*length xs* − *Suc 0*) ≤ *ys* ! *0 x* ∈ *set xs y* ∈ *set ys*
moreover then obtain *i j* **where** *i*: *x* = *xs*!*i i* < *length xs* **and** *j*: *y* = *ys*!*j j* < *length ys*
by (*auto simp add*: *in-set-conv-nth*)
moreover have *xs* ! *i* ≤ *xs*!(*length xs* − *1*)
by (*metis One-nat-def Suc-diff-Suc Suc-leI Suc-le-mono* ‹*i* < *length xs*› ‹*sorted xs*› *diff-less diff-zero gr-implies-not-zero nat.simps*(*3*) *sorted-iff-nth-mono zero-less-iff-neq-zero*)
moreover have *ys*!*0* ≤ *ys* ! *j*
by (*simp add*: *calculation*(*2*) *calculation*(*9*) *sorted-nth-mono*)
ultimately have *xs* ! *i* ≤ *ys* ! *j*
by (*metis One-nat-def dual-order.trans*)
thus *x* ≤ *y*
by (*simp add*: *i j*)
qed

thus *?thesis*
by (*auto simp add*: *sorted-append*)
qed

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 (*simp-all add*: *is-sorted-list-of-set-def*)
apply (*metis* (*mono-tags*, *lifting*) *all-nth-imp-all-set less-le-trans linorder-not-less not-less-iff-gr-or-eq nth-Cons-0 sorted-iff-nth-mono zero-order*(*3*))
done
from *srt hyps*(*2*) **have** *distinct xs*
by (*simp add*: *is-sorted-list-of-set-def*)
with *isl* **show** *distinct* (*x* # *xs*)
proof −
have (∀ *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* ‹*distinct xs*› *add.commute add-diff-cancel-left*′ *distinct.simps*(*2*) *leD length-Cons length-greater-0-conv length-pos-if-in-set less-le nth-Cons-0 nth-Cons-Suc plus-1-eq-Suc set-ConsD sorted-wrt.elims*(*2*) *srtd*)
qed
qed

lemma *is-sorted-list-of-set-alt-def*:
is-sorted-list-of-set A xs ⟷ *sorted* (*xs*) ∧ *distinct* (*xs*) ∧ *set*(*xs*) = *A*
by (*metis is-sorted-list-of-set-def sorted-distinct sorted-is-sorted-list-of-set*(*1*,*2*))

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*)
using *sorted-distinct sorted-list-of-set*(*2*) **by** (*fastforce simp add*: *is-sorted-list-of-set-def*)

lemma *sorted-list-of-set-other-def*:
finite A ⟹ *sorted-list-of-set*(*A*) = (*THE xs*. *sorted*(*xs*) ∧ *distinct*(*xs*) ∧ *set xs* = *A*)
by (*metis* (*mono-tags*, *lifting*) *sorted-list-of-set.distinct-sorted-key-list-of-set sorted-list-of-set.idem-if-sorted-distinct sorted-list-of-set.set-sorted-key-list-of-set sorted-list-of-set.sorted-sorted-key-list-of-set the-equality*)

lemma *sorted-list-of-set-alt* [*simp*]:
finite A ⟹ *sorted-list-of-set-alt*(*A*) = *sorted-list-of-set*(*A*)
by (*metis is-sorted-list-of-set is-sorted-list-of-set-def sorted-is-sorted-list-of-set*(*1*,*2*) *sorted-list-of-set.idem-if-sorted-distinct sorted-list-of-set-alt-def sorted-list-of-set-eq-Nil-iff the-equality*)

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

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

with *fin hyp* **show** $(xs,\ ys) \in$ *fin-set-lexord* $\{(x,\ y).\ g\ x\ y\}$
by (*simp add*: *fin-set-lexord-def*, *metis case-prod-conv is-sorted-list-of-set-by-def lexord-mono′ mem-Collect-eq*)

qed

definition *distincts* :: *′a set* ⇒ *′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*)

2.5.3 List Update

lemma *listsum-update*:
fixes *xs* :: *′a*::*ring list*
assumes $i <$ *length xs*
shows *list-sum* (*xs*[*i* := *v*]) = *list-sum xs* − *xs* ! *i* + *v*
using *assms* **proof** (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** (*simp*)
next
case (*Cons a xs*)
then show *?case*
proof (*cases i*)
case *0*
thus *?thesis*
by (*simp add*: *add.commute*)
next
case (*Suc i′*)
with *Cons* **show** *?thesis*
by (*auto*)
qed
qed

2.5.4 Drop While and Take While

lemma *dropWhile-sorted-le-above*:
$[\![$ *sorted xs*; $x \in$ *set* (*dropWhile* (λ *x*. $x \leq n$) *xs*) $]\!] \Longrightarrow x > n$
proof (*induct xs*)
case *Nil*
then show *?case*
by *simp*

next
case (*Cons a xs*)
then show *?case*
proof (*cases* $a \leq n$)
case *True*
with *Cons* **show** *?thesis* **by** *simp*
next
case *False*
with *Cons* **show** *?thesis*
by *force*
qed
qed

lemma *set-dropWhile-le*:
sorted xs $\Longrightarrow$ *set* (*dropWhile* (λ *x*. $x \leq n$) *xs*) = $\{x \in set\ xs.\ x > n\}$
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
hence *sorted xs*
using *sorted-simps*(*2*) **by** *blast*
with *Cons* **show** *?case*
by *force*
qed

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*
qed

lemma *nth-le-takeWhile-ord*: $[\![$ *sorted xs*; $i \geq length$ (*takeWhile* (λ *x*. $x \leq n$) *xs*);
$i < length\ xs$ $]\!] \Longrightarrow n \leq xs\ !\ i$
proof (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by (*meson dual-order.trans nle-le nth-length-takeWhile order-le-less-trans sorted-iff-nth-mono*)

qed

lemma *length-takeWhile-less*:

⟦ $a \in set\ xs$; $\neg\ P\ a$ ⟧ ⟹ $length\ (takeWhile\ P\ xs) < length\ xs$
by (*metis in-set-conv-nth length-takeWhile-le nat-neq-iff not-less set-takeWhileD takeWhile-nth*)

lemma *nth-length-takeWhile-less*:
⟦ *sorted xs*; *distinct xs*; (∃ $a \in set\ xs.\ a \geq n$) ⟧ ⟹ $xs\ !\ length\ (takeWhile\ (\lambda x.\ x < n)\ xs) \geq n$
by (*induct xs*, *auto*)

2.5.5 Last and But Last

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

lemma *length-eq-zero-butlast-concat*:
assumes $length\ ys = 0$
shows $butlast\ (xs \bullet ys) = butlast\ xs$
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 $last\ (s \bullet t) = last\ s$
by (*metis append-Nil2 assms length-0-conv*)

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

2.5.6 Prefixes and Strict Prefixes

lemma *prefix-length-eq*:
⟦ $length\ xs = length\ ys$; *prefix xs ys* ⟧ ⟹ $xs = ys$
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*:
⟦ *inj-on f* (*set xs* ∪ *set ys*); *prefix* (*map f xs*) (*map f ys*) ⟧ ⟹
prefix xs ys
proof (*induct xs arbitrary*:*ys*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons x xs*)
obtain *ys'* **where** *map f ys* = *f x* # *ys' prefix* (*map f xs*) *ys'*
using *Cons.prems*(*2*) **by** *auto*
with *Cons* **show** *?case*
by (*simp*, *safe*, *metis Diff-iff Sublist.Cons-prefix-Cons Un-insert-right empty-iff image-eqI inj-on-insert insert-iff list.simps*(*15*))
qed

lemma *prefix-map-inj-eq* [*simp*]:
inj-on f (*set xs* ∪ *set ys*) ⟹
prefix (*map f xs*) (*map f ys*) ⟷ *prefix xs ys*
using *map-mono-prefix prefix-map-inj* **by** *blast*

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

lemma *strict-prefix-map-inj*:
⟦ *inj-on f* (*set xs* ∪ *set ys*); *strict-prefix* (*map f xs*) (*map f ys*) ⟧ ⟹
strict-prefix xs ys
apply (*induct xs arbitrary*:*ys*)
apply (*simp-all*)
using *prefix-bot.not-eq-extremum* **apply** *fastforce*
apply (*erule strict-prefix-Cons-elim*)
apply (*safe*)
apply (*metis Diff-iff Sublist.strict-prefix-simps*(*3*) *Un-insert-right empty-iff imageI inj-on-insert insert-iff*
list.simps(*15*))
done

lemma *strict-prefix-map-inj-eq* [*simp*]:
inj-on f (*set xs* ∪ *set ys*) ⟹
strict-prefix (*map f xs*) (*map f ys*) ⟷ *strict-prefix xs ys*
by (*simp add*: *inj-on-map-eq-map strict-prefix-def*)

lemma *prefix-drop*:
⟦ *drop* (*length xs*) *ys* = *zs*; *prefix xs ys* ⟧
⟹ *ys* = *xs* • *zs*

by (*metis append-eq-conv-conj prefix-def*)

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

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

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

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

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

lemma *prefix-eq-exists*:
prefix s t $\longleftrightarrow$ $(\exists\, xs\ .\ s \bullet xs = t)$
using *prefix-def* **by** *auto*

lemma *strict-prefix-eq-exists*:
strict-prefix s t $\longleftrightarrow$ $(\exists\, xs\ .\ s \bullet xs = t \wedge (length\ xs) > 0)$
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* $\leq$ *length t*

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

lemma *prefix-imp-length-not-gt*:
assumes *prefix s t*
shows ¬ *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 ¬ (*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 (¬ (*length s* < *length t*)) = (*length s* = *length t*)
proof −
have (¬ (*length s* < *length t*)) = ((*length t* < *length s*) ∨ (*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

lemma *list-prefix-iff*:
(*prefix xs ys* ⟷ (*length xs* ≤ *length ys* ∧ (∀ *i*<*length xs*. *xs*!*i* = *ys*!*i*)))
apply (*safe*)
apply (*simp add*: *prefix-imp-length-lteq*)
apply (*metis nth-append prefix-def*)
apply (*metis nth-take-lemma order-refl take-all take-is-prefix*)
done

lemma *list-le-prefix-iff*:
(*xs* ≤ *ys* ⟷ (*length xs* ≤ *length ys* ∧ (∀ *i*<*length xs*. *xs*!*i* = *ys*!*i*)))
by (*simp add*: *less-eq-list-def list-prefix-iff*)

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
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case
  proof (cases ys)
    case Nil
    then show ?thesis by simp
  next
    case (Cons a list)
    then show ?thesis
      by (simp add: Cons.hyps)
  qed
qed

lemma gcp-lb2: prefix (gcp xs ys) ys
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case
  proof (cases ys)
    case Nil
    then show ?thesis by simp
  next
    case (Cons a list)
    then show ?thesis
      by (simp add: Cons.hyps)
  qed
qed

interpretation prefix-semilattice: semilattice-inf gcp prefix strict-prefix
proof
  fix xs ys :: 'a list
  show prefix (gcp xs ys) xs
  proof (induct xs arbitrary: ys)
    case Nil
    then show ?case by simp
  next
    case (Cons a xs)
    then show ?case
```

```
    by (simp add: gcp-lb1)
  qed
  show prefix (gcp xs ys) ys
  proof (induct xs arbitrary: ys)
    case Nil
    then show ?case by simp
  next
    case (Cons a xs)
    then show ?case
      by (simp add: gcp-lb2)
  qed
next
  fix xs ys zs :: 'a list
  assume prefix xs ys prefix xs zs
  thus prefix xs (gcp ys zs)
    by (simp add: prefix-def, auto)
qed
```

2.5.7 Lexicographic Order

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

qed

lemma *strict-prefix-lexord-rel*:
strict-prefix xs ys $\Longrightarrow$ (*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 *force*
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* **have** ((*x* # *xs*) ! *Suc i*, *ys* ! *Suc i*) $\in$ *R* $\wedge$ *Suc i* < *length* (*x* # *xs*) $\wedge$ ($\forall$ *j*<*Suc i*. (*x* # *xs*) ! *j* = *ys* ! *j*)
by (*auto simp add*: *less-Suc-eq-0-disj*)
thus *?thesis*
by *blast*
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 (*simp*, *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** ∀ *j*<*i*. *xs'* ! *j* = *ys'* ! *j*
by (*auto*)
ultimately show *?case* **using** *hyps*
by (*auto*)
qed

2.6 Distributed Concatenation

definition *uncurry* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a* × '*b* ⇒ '*c*) **where**
[*simp*]: *uncurry f* = (λ(*x*, *y*). *f x y*)

definition *dist-concat* ::
'*a list set* ⇒ '*a list set* ⇒ '*a list set* (**infixr** ⁀ *100*) **where**
dist-concat ls1 ls2 = (*uncurry* (•) ‘ (*ls1* × *ls2*))

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

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

lemma *dist-concat-insert* [*simp*]:
insert l ls1 ⁀ *ls2* = ((•) *l* ‘ (*ls2*)) ∪ (*ls1* ⁀ *ls2*)
by (*auto simp add*: *dist-concat-def*)

2.7 List Domain and Range

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

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

2.8 Extracting List Elements

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

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

lemma *seq-extract-Cons*:
A ⌈$_l$ (*x* # *xs*) = (*if 0* ∈ *A then* [*x*] *else* []) • {*j*. *Suc j* ∈ *A*} ⌈$_l$ *xs*

by (*simp add*: *seq-extract-def nths-Cons*)

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

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

lemma *seq-extract-split*:
assumes $i \leq length\ xs$
shows $\{0..<i\} \restriction_l xs \bullet \{i..<length\ xs\} \restriction_l xs = xs$
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 $\{j.\ i \leq Suc\ j \wedge j < length\ xs\} = \{i - 1..<length\ xs\}$
by (*auto*)
ultimately show *?case*
using *hyp* **by** (*force simp add*: *seq-extract-def nths-Cons*)
qed

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

lemma *seq-extract-range*: $A \restriction_l xs = (A \cap dom_l(xs)) \restriction_l xs$
apply (*simp add*: *seq-extract-def nths-def*)
apply (*metis* (*no-types*, *lifting*) *atLeastLessThan-iff filter-cong in-set-zip nth-mem set-upt*)
done

lemma *seq-extract-out-of-range*:
$A \cap dom_l(xs) = \{\} \Longrightarrow A \restriction_l xs = []$
by (*metis seq-extract-def seq-extract-range nths-empty*)

lemma *seq-extract-length* [*simp*]:
$length\ (A \restriction_l xs) = card\ (A \cap dom_l(xs))$
proof −
have $\{i.\ i < length(xs) \wedge i \in A\} = (A \cap \{0..<length(xs)\})$
by (*auto*)
thus *?thesis*
by (*simp add*: *seq-extract-def length-nths*)
qed

lemma *seq-extract-Cons-atLeastLessThan*:

assumes $m < n$
shows $\{m..<n\} \restriction_l (x \# xs) = (if\ (m = 0)\ then\ x \# (\{0..<n-1\} \restriction_l xs)\ else\ \{m-1..<n-1\} \restriction_l xs)$
proof −
have $\{j.\ Suc\ j < n\} = \{0..<n - Suc\ 0\}$
by (*auto*)
moreover have $\{j.\ m \leq Suc\ j \wedge Suc\ j < n\} = \{m - Suc\ 0..<n - Suc\ 0\}$
by (*auto*)

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

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

lemma *seq-extract-as-map*:
assumes $m < n\ n \leq length\ xs$
shows $\{m..<n\} \restriction_l xs = map\ (nth\ xs)\ [m..<n]$
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*:
$xs = ys \bullet zs \longleftrightarrow (\exists\ i \leq length(xs).\ ys = \{0..<i\} \restriction_l xs \wedge zs = \{i..<length(xs)\} \restriction_l xs)$
proof
assume *xs*: $xs = ys \bullet zs$
moreover have $ys = \{0..<length\ ys\} \restriction_l (ys \bullet zs)$
by (*simp add*: *seq-extract-append*)
moreover have $zs = \{length\ ys..<length\ ys + length\ zs\} \restriction_l (ys \bullet zs)$

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 ($\exists\ i \leq length(xs).\ ys = \{0..<i\} \restriction_l xs \wedge zs = \{i..<length(xs)\} \restriction_l xs$)
using *le-iff-add* **by** *auto*
next
assume $\exists\, i \leq length\ xs.\ ys = \{0..<i\} \restriction_l xs \wedge zs = \{i..<length\ xs\} \restriction_l xs$
thus $xs = ys \bullet zs$
by (*auto simp add*: *seq-extract-split*)
qed

2.9 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*: $(xs \bullet ys) \restriction_l A = (xs \restriction_l A) \bullet (ys \restriction_l A)$
by (*simp add*: *seq-filter-def*)

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

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

2.10 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*]: $(xs \bullet ys) - xs = ys$
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*: $(s \bullet t) - (s \bullet z) = t - z$
by (*simp add*: *minus-list-def*)

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

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

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

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

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

lemma *prefix-minus-concat*:
assumes *prefix s t*
shows $(t - s) \bullet z = (t \bullet z) - s$
using *assms* **by** (*simp add*: *Sublist.prefix-length-le minus-list-def*)

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

lemma *strict-prefix-diff-minus*:

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

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

lemma *list-minus-butlast-eq-butlast-list*:
assumes *length t* = *length s* **and** *strict-prefix* (*butlast s*) *t*
shows *t* − (*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*)

lemma *length-minus-le*: *length* (*ys* − *xs*) ≤ *length ys*
by (*simp add*: *minus-list-def*)

lemma *length-minus-less*: ⟦ *xs* ≤ *ys*; *xs* ≠ [] ⟧ ⟹ *length* (*ys* − *xs*) < *length ys*
by (*simp add*: *minus-list-def less-eq-list-def*)
(*metis diff-less length-greater-0-conv prefix-bot.extremum-uniqueI*)

lemma *filter-minus* [*simp*]: *ys* ≤ *xs* ⟹ *filter P* (*xs* − *ys*) = *filter P xs* − *filter P ys*
by (*simp add*: *minus-list-def less-eq-list-def filter-mono-prefix*)
(*metis filter-append filter-mono-prefix prefix-drop same-append-eq*)

2.11 Laws on *list-update*

lemma *list-update-0*: *length*(*xs*) > *0* ⟹ *xs*[*0* := *x*] = *x* # *tl xs*
by (*metis length-0-conv list.collapse list-update-code*(*2*) *nat-less-le*)

lemma *tl-list-update*: ⟦ *length xs* > *0*; *k* > *0* ⟧ ⟹ *tl*(*xs*[*k* := *v*]) = (*tl xs*)[*k*−*1* := *v*]
by (*metis One-nat-def Suc-pred length-greater-0-conv list.collapse list.sel*(*3*) *list-update-code*(*3*))

2.12 Laws on *take*, *drop*, and *nths*

lemma *take-prefix*: *m* ≤ *n* ⟹ *take m xs* ≤ *take n xs*
by (*metis Prefix-Order.prefixI append-take-drop-id min-absorb2 take-append take-take*)

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

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

lemma *nths-atLeastAtMost-prefix*: *m* ≤ *n* ⟹ *nths xs* {*0*..*m*} ≤ *nths xs* {*0*..*n*}
by (*simp add*: *nths-atLeastAtMost-0-take take-prefix*)

lemma *sorted-nths-atLeastAtMost-0*: ⟦ *m* ≤ *n*; *sorted* (*nths xs* {*0*..*n*}) ⟧ ⟹ *sorted* (*nths xs* {*0*..*m*})
using *nths-atLeastAtMost-prefix sorted-prefix* **by** *blast*

lemma *sorted-nths-atLeastLessThan-0*: ⟦ *m* ≤ *n*; *sorted* (*nths xs* {*0*..<*n*}) ⟧ ⟹ *sorted* (*nths xs* {*0*..<*m*})
by (*metis atLeast0LessThan nths-upt-eq-take sorted-prefix take-prefix*)

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

lemma *nths-list-update-out*: $k \notin A \Longrightarrow nths\ (list\text{-}update\ xs\ k\ x)\ A = nths\ xs\ A$
proof (*induct xs arbitrary*: $k\ x\ A$)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases k*, *auto simp add*: *nths-Cons*)
qed

lemma *nths-list-augment-out*: $[\![\ k < length\ xs;\ \ k \notin A\ ]\!] \Longrightarrow nths\ (list\text{-}augment\ xs\ k\ x)\ A = nths\ xs\ A$
  by (*simp add*: *list-augment-as-update nths-list-update-out*)

lemma *nths-none*:
  assumes $\forall\, i \in I.\ i \geq length\ xs$
  shows $nths\ xs\ I = []$
proof −
  from *assms* **have** $\forall\, x{\in}set\ (zip\ xs\ [0..{<}length\ xs]).\ snd\ x \notin I$
    by (*metis atLeastLessThan-iff in-set-zip leD nth-mem set-upt*)
  thus *?thesis*
    by (*simp add*: *nths-def*)
qed

lemma *nths-uptoLessThan*:
  $[\![\ m \leq n;\ \ n < length\ xs\ ]\!] \Longrightarrow nths\ xs\ \{m..n\} = xs\ !\ m\ \#\ nths\ xs\ \{Suc\ m..n\}$
proof (*induct xs arbitrary*: *m n*)
case *Nil*
  then show *?case* **by** (*simp*)
next
  case (*Cons a xs*)
  have *l1*: $\bigwedge\ m\ n.\ [\![\ 0 < m;\ \ m \leq n\ ]\!] \Longrightarrow \{j.\ m \leq Suc\ j \wedge Suc\ j \leq n\} = \{m{-}1..n{-}1\}$
    by (*auto*)
  have *l2*: $\bigwedge\ m\ n.\ [\![\ 0 < m;\ \ m \leq n\ ]\!] \Longrightarrow \{j.\ m \leq j \wedge Suc\ j \leq n\} = \{m..n{-}1\}$
    by (*auto*)
  from *Cons* **show** *?case* **by** (*auto simp add*: *nths-Cons l1 l2*)
qed

lemma *nths-upt-nth*: $[\![\ j < i;\ \ i < length\ xs\ ]\!] \Longrightarrow (nths\ xs\ \{0..{<}i\})\ !\ j = xs\ !\ j$
  by (*metis lessThan-atLeast0 nth-take nths-upt-eq-take*)

lemma *nths-upt-length*: $[\![\ m \leq n;\ \ n \leq length\ xs\ ]\!] \Longrightarrow length\ (nths\ xs\ \{m..{<}n\}) = n{-}m$

by (*metis atLeastLessThan-empty diff-is-0-eq length-map length-upt list.size(3) not-less nths-empty seq-extract-as-map seq-extract-def*)

lemma *nths-upt-le-length*:
⟦ $m \leq n$; $Suc\ n \leq length\ xs$ ⟧ ⟹ $length\ (nths\ xs\ \{m..n\}) = Suc\ n - m$
by (*metis atLeastLessThanSuc-atLeastAtMost le-SucI nths-upt-length*)

lemma *sl1*: $n > 0 \Longrightarrow \{j.\ Suc\ j \leq n\} = \{0..n-1\}$
by (*auto*)

lemma *sl2*: ⟦ $0 < m$; $m \leq n$ ⟧ ⟹ $\{j.\ m \leq Suc\ j \wedge Suc\ j \leq n\} = \{m-1..n-1\}$
by *auto*

lemma *nths-upt-le-nth*: ⟦ $m \leq n$; $Suc\ n \leq length\ xs$; $i < Suc\ n - m$ ⟧
⟹ $(nths\ xs\ \{m..n\})\ !\ i = xs\ !\ (i + m)$
proof (*induct xs arbitrary*: *m n i*)
case *Nil*
then show *?case* **by** (*simp*)
next
case (*Cons a xs*)
then show *?case*
proof (*cases i = 0*)
case *True*
with *Cons* **show** *?thesis* **by** (*auto simp add*: *nths-Cons sl2*)
next
case *False*
with *Cons* **show** *?thesis* **by** (*auto simp add*: *nths-Cons sl1 sl2*)
qed
qed

lemma *nths-split-union*:
assumes $\bigwedge x\ y.\ x \in A \Longrightarrow y \in B \Longrightarrow x < y$
shows $nths\ l\ A \bullet nths\ l\ B = nths\ l\ (A \cup B)$
proof (*induct l rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc x xs*)
{ assume ∗: $length\ xs \notin (A \cup B)$
then have *L*: $nths\ (xs \bullet [x])\ (A \cup B) = nths\ xs\ (A \cup B)$
by (*simp add*: *nths-append*)
from ∗ **have** *R*: $nths\ (xs \bullet [x])\ A = nths\ xs\ A\ nths\ (xs \bullet [x])\ B = nths\ xs\ B$
by (*simp add*: *nths-append*)+
have *?case*
using *L R snoc* **by** *presburger*
} note *length-notin* = *this*

{ assume ∗: $length\ xs \in (A \cup B)$
then have *new-nths*: $nths\ (xs \bullet [x])\ (A \cup B) = nths\ xs\ (A \cup B) \bullet [x]$

```
    by (simp add: nths-append)
  from ∗ consider (A) length xs ∈ A | (B) length xs ∈ B
    using assms by auto
  then have ?case
  proof cases
    case A
    then have i ∈ B ⟹ i > length xs for i
      using assms by force
    then have nths-B: nths (xs • [x]) B = []
        by (metis Suc-less-eq le-simps(2) length-Cons length-rotate1 nths-none
rotate1.simps(2))
    from A have nths (xs • [x]) A = nths xs A • [x]
      by (simp add: nths-append)
    then have nths (xs • [x]) A • nths (xs • [x]) B = (nths xs A • nths xs B) •
[x]
      by (metis append-is-Nil-conv nths-B nths-append self-append-conv)
    then show ?thesis
      using snoc new-nths by presburger
  next
    case B
    then have length xs ∉ A
      using assms by blast
    then have nths (xs • [x]) A = nths xs A
      by (simp add: nths-append)
    moreover have nths (xs • [x]) B = nths xs B • [x]
      by (simp add: B nths-append)
    ultimately show ?thesis
      by (simp add: new-nths snoc)
  qed
 } note length-in=this

 show ?case
   using length-in length-notin by blast
qed

corollary nths-upt-le-append-split:
 j ≤ i ⟹ nths xs {0..<j} • nths xs {j..i} = nths xs {0..i}
proof −
 assume j ≤ i
 have ⋀x y. x ∈ {0..<j} ⟹ y ∈ {j..i} ⟹ x < y
   by auto
 then have nths xs {0..<j} • nths xs {j..i} = nths xs ({0..<j} ∪ {j..i})
   by (rule nths-split-union)
 moreover have {0..<j} ∪ {j..i} = {0..i}
   by (simp add: ivl-disj-un-two(7) ‹j ≤ i›)
 ultimately show ?thesis
   using ‹j ≤ i› by presburger
qed
```

lemma *nths-Cons-atLeastAtMost*: $n > m \Longrightarrow nths\ (x \# xs)\ \{m..n\} = (if\ m = 0\ then\ x \# nths\ xs\ \{0..n-1\}\ else\ nths\ xs\ \{m-1..n-1\})$
apply (*simp add*: *nths-Cons*)
apply *safe*
using *One-nat-def sl1* **apply** *presburger*
using *One-nat-def le-eq-less-or-eq sl2* **apply** *presburger*
done

lemma *nths-atLeastAtMost-eq-drop-take*: $nths\ xs\ \{m..n\} = drop\ m\ (take\ (n+1)\ xs)$
by (*induct xs rule*: *rev-induct*, *simp-all split*: *nat-diff-split add*: *nths-append*, *linarith*)

lemma *drop-as-map*: $drop\ m\ xs = map\ (nth\ xs)\ [m..<length\ xs]$
by (*metis add.commute add.right-neutral drop-map drop-upt map-nth*)

lemma *take-as-map*: $take\ m\ xs = map\ (nth\ xs)\ [0..<min\ m\ (length\ xs)]$
by (*metis* (*no-types*, *lifting*) *add-0 map-nth min.cobounded2 min.commute min-def take-all-iff take-map take-upt*)

lemma *nths-atLeastAtMost-as-map*: $nths\ xs\ \{m..n\} = map\ (\lambda\ i.\ xs\ !\ i)\ [m..<min\ (n+1)\ (length\ xs)]$
by (*simp add*: *nths-atLeastAtMost-eq-drop-take drop-as-map take-as-map*)

lemma *nths-single*: $nths\ xs\ \{k\} = (if\ k < length\ xs\ then\ [xs\ !\ k]\ else\ [])$
using *nths-atLeastAtMost-as-map*[*of xs k k*] **by** *simp*

lemma *nths-list-update-in-range*: $k \in \{m..n\} \Longrightarrow nths\ (list\text{-}update\ xs\ k\ x)\ \{m..n\} = list\text{-}update\ (nths\ xs\ \{m..n\})\ (k - m)\ x$
proof (*induct xs arbitrary*: *k x m n*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
then show *?case*
proof (*cases k*)
case *0*
then show *?thesis* **apply** (*simp*)
by (*smt* (*verit*, *best*) *Cons.prems append-Cons list-update-code*(*2*) *nths-Cons*)
next
case (*Suc nat*)
with *Cons* **show** *?thesis*
proof (*cases n m rule*: *linorder-cases*)
case *less*
then show *?thesis*
by *force*
next
case *equal*
with *Suc* **show** *?thesis*

by (*cases k*, *force*)
(*metis* (*no-types*, *opaque-lifting*) *Cons.prems nths-single atLeastAtMost-singleton diff-is-0-eq' length-list-update list-update-code*(*2*) *list-update-code*(*3*) *list-update-nonempty nle-le nth-list-update-eq singletonD*)
next
case *greater*
with *Suc Cons*(*1*)[*of nat m − Suc 0 n − Suc 0 x*] **show** *?thesis*
by (*cases k*, *simp-all add*: *nths-Cons-atLeastAtMost nths-atLeastAtMost-0-take take-update-swap*, *safe*)
(*metis Cons.prems Suc-le-mono Suc-pred atLeastAtMost-iff diff-Suc-Suc le-zero-eq not-gr-zero*)
qed
qed
qed

lemma *length-nths-atLeastAtMost* [*simp*]: *length* (*nths xs* {*m*..*n*}) = *min* (*Suc n*) (*length xs*) − *m*
by (*simp add*: *nths-atLeastAtMost-as-map*)

lemma *hd-nths-atLeastAtMost*: ⟦ *m* < *length xs*; *m* ≤ *n* ⟧ ⟹ *hd* (*nths xs* {*m*..*n*}) = *xs* ! *m*
by (*simp add*: *nths-atLeastAtMost-as-map upt-conv-Cons*)

lemma *tl-nths-atLeastAtMost*: *tl* (*nths xs* {*m*..*n*}) = *nths xs* {*Suc m*..*n*}
by (*simp add*: *nths-atLeastAtMost-as-map*, *metis map-tl tl-upt*)

lemma *set-nths-atLeastAtMost*: *set* (*nths xs* {*m*..*n*}) = {*xs*!*i* | *i*. *m* ≤ *i* ∧ *i* ≤ *n* ∧ *i* < *length xs*}
by (*auto simp add*: *nths-atLeastAtMost-as-map*)

lemma *nths-atLeastAtMost-neq-Nil* [*simp*]: ⟦ *m* ≤ *n*; *length xs* > *m* ⟧ ⟹ *nths xs* {*m*..*n*} ≠ []
by (*force simp add*: *nths-atLeastAtMost-as-map*)

lemma *nths-atLeastAtMost-head*: ⟦ *m* ≤ *n*; *m* < *length xs* ⟧ ⟹ *nths xs* {*m*..*n*} = *xs* ! *m* # (*nths xs* {*Suc m*..*n*})
by (*simp add*: *nths-atLeastAtMost-as-map upt-conv-Cons*)

lemma *sorted-hd-le-all*: ⟦ *xs* ≠ []; *sorted xs*; *x* ∈ *set xs* ⟧ ⟹ *hd xs* ≤ *x*
by (*metis Orderings.order-eq-iff list.sel*(*1*) *list.set-cases sorted-simps*(*2*))

2.13 List power

overloading
listpow ≡ *compow* :: *nat* ⇒ *'a list* ⇒ *'a list*
begin

fun *listpow* :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
listpow 0 xs = []

| *listpow* (*Suc n*) *xs* = *xs* • *listpow n xs*

end

lemma *listpow-Nil* [*simp*]: [] ^^ *n* = []
by (*induct n*) *simp-all*

lemma *listpow-Suc-right*: *xs* ^^ *Suc n* = *xs* ^^ *n* • *xs*
by (*induct n*) *simp-all*

lemma *listpow-add*: *xs* ^^ (*m* + *n*) = *xs* ^^ *m* • *xs* ^^ *n*
by (*induct m*) *simp-all*

2.14 Alternative List Lexicographic Order

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

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

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

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

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

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

2.15 Bounded List Universe

Analogous to *List.n-lists*, but includes all lists with a length up to the given number.

definition *b-lists* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
b-lists n xs = *concat* (*map* (λ *n*. *List.n-lists n xs*) [*0*..<*Suc n*])

lemma *b-lists-Nil* [*simp*]: *b-lists n* [] = [[]]
unfolding *b-lists-def* **by** (*induct n*) *simp-all*

lemma *length-b-lists-elem*: *ys* ∈ *set* (*b-lists n xs*) ⟹ *length ys* ≤ *n*
unfolding *b-lists-def*
by (*auto simp add*: *length-n-lists-elem*)

lemma *b-lists-in-lists*: *ys* ∈ *set* (*b-lists n xs*) ⟹ *ys* ∈ *lists* (*set xs*)
by (*auto simp add*: *b-lists-def in-mono set-n-lists*)

lemma *in-blistsI*:
assumes *length xs* ≤ *n xs* ∈ *lists* (*set A*)
shows *xs* ∈ *set* (*b-lists n A*)
proof −
have *xs* ∉ *set* (*List.n-lists n A*) ⟹ *xs* ∈ *set* (*List.n-lists* (*length xs*) *A*)
using *assms*(*2*) *in-lists-conv-set set-n-lists* **by** *fastforce*
with *assms* **show** *?thesis*
by (*force simp add*: *set-n-lists subsetI b-lists-def*)
qed

lemma *ex-list-nonempty-carrier*:
assumes *A* ≠ {}
obtains *xs* **where** *length xs* = *n set xs* ⊆ *A*
proof −
obtain *a* **where** *a*: *a* ∈ *A*
using *assms* **by** *blast*
hence *set* (*replicate n a*) ⊆ *A*
by (*simp add*: *set-replicate-conv-if*)
with *that* **show** *?thesis*
by (*meson length-replicate*)
qed

lemma *n-lists-inj*:
assumes *xs* ≠ [] *List.n-lists m xs* = *List.n-lists n xs*
shows *m* = *n*
proof (*rule ccontr*)
assume *mn*: *m* ≠ *n*
hence *m* < *n* ∨ *m* > *n*
by *auto*
moreover have *m* < *n* ⟶ *False*
proof
assume *m* < *n*
then obtain *ys* **where** *ys*: *length ys* = *n set ys* ⊆ *set xs*
by (*metis all-not-in-conv assms*(*1*) *ex-list-nonempty-carrier length-0-conv neq0-conv nth-mem*)
hence *ys* ∈ *set* (*List.n-lists n xs*)
by (*simp add*: *set-n-lists*)
moreover have *ys* ∉ *set* (*List.n-lists m xs*)
using *length-n-lists-elem mn ys*(*1*) **by** *blast*
ultimately show *False*
by (*simp add*: *assms*(*2*))
qed
moreover have *m* > *n* ⟶ *False*
proof
assume *n* < *m*

then obtain *ys* **where** *ys*: *length ys* = *m set ys* ⊆ *set xs*
by (*metis all-not-in-conv assms*(*1*) *ex-list-nonempty-carrier length-0-conv neq0-conv nth-mem*)
hence *ys* ∈ *set* (*List.n-lists m xs*)
by (*simp add*: *set-n-lists*)
moreover have *ys* ∉ *set* (*List.n-lists n xs*)
using *length-n-lists-elem mn ys*(*1*) **by** *blast*
ultimately show *False*
by (*simp add*: *assms*(*2*))
qed
ultimately show *False*
by *blast*
qed

lemma *distinct-b-lists*: *distinct xs* ⟹ *distinct* (*b-lists n xs*)
apply (*cases xs* = [])
apply (*simp*)
apply (*simp add*: *b-lists-def*, *safe*)
apply (*rule distinct-concat*)
apply (*simp add*: *distinct-map*)
apply (*simp add*: *inj-onI n-lists-inj*)
using *distinct-n-lists* **apply** *auto*[*1*]
apply (*safe*)
apply (*metis ex-map-conv length-n-lists-elem*)
apply (*simp add*: *distinct-n-lists*)
apply (*metis atLeastLessThan-iff length-n-lists-elem order-less-irrefl*)
done

definition *bounded-lists* :: *nat* ⇒ *'a set* ⇒ *'a list set* **where**
bounded-lists n A = {*xs*∈*lists A*. *length xs* ≤ *n*}

lemma *bounded-lists-b-lists* [*code*]: *bounded-lists n* (*set xs*) = *set* (*b-lists n xs*)
apply (*simp add*: *bounded-lists-def in-blistsI in-listsI*, *safe*)
using *in-blistsI* **apply** *blast*
apply (*meson b-lists-in-lists in-lists-conv-set*)
apply (*meson length-b-lists-elem*)
done

2.16 Disjointness and Partitions

definition *list-disjoint* :: *'a set list* ⇒ *bool* **where**
list-disjoint xs = (∀ *i* < *length xs*. ∀ *j* < *length xs*. *i* ≠ *j* ⟶ *xs*!*i* ∩ *xs*!*j* = {})

definition *list-partitions* :: *'a set list* ⇒ *'a set* ⇒ *bool* **where**
list-partitions xs T = (*list-disjoint xs* ∧ ⋃ (*set xs*) = *T*)

lemma *list-disjoint-Nil* [*simp*]: *list-disjoint* []
by (*simp add*: *list-disjoint-def*)

lemma *list-disjoint-Cons* [*simp*]: *list-disjoint* (*A* # *Bs*) = ((∀ *B* ∈ *set Bs*. *A* ∩ *B* = {}) ∧ *list-disjoint Bs*)
apply (*simp add*: *list-disjoint-def disjoint-iff*)
apply (*safe*)
apply (*metis Suc-less-eq in-set-conv-nth nat.distinct*(*1*) *neq0-conv nth-Cons-0 nth-Cons-Suc*)
apply (*metis lessI lift-Suc-mono-less-iff nat.inject nth-Cons-Suc*)
apply (*metis less-Suc-eq-0-disj*[*of - length Bs*] *nth-Cons-0*[*of A Bs*] *nth-Cons-Suc*[*of A Bs*] *nth-mem*[*of - Bs*])
done

2.17 Code Generation

lemma *set-singleton-iff*: *set xs* = {*x*} ⟷ *remdups xs* = [*x*]
apply (*safe*)
apply (*metis card-set empty-set insert-not-empty length-0-conv length-Suc-conv list.simps*(*15*) *remdups.simps*(*1*) *remdups.simps*(*2*) *set-remdups the-elem-set*)
apply (*metis empty-iff empty-set set-ConsD set-remdups*)
apply (*metis list.set-intros*(*1*) *set-remdups*)
done

lemma *list-singleton-iff*: (∃ *x*. *xs* = [*x*]) ⟷ (*length xs* = *1*)
by (*auto simp add*: *length-Suc-conv*)

end

3 Infinite Sequences

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

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

setup-lifting *type-definition-infseq*

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

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

abbreviation *sinit* :: *nat* ⇒ *'a infseq* ⇒ *'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*, *simp*, *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 (*simp add*: *map-mono-prefix prefix-upt-0 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 (*unfold 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 (*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 *infseq-lexord* :: *'a rel* ⇒ (*'a infseq*) *rel* **where**
infseq-lexord R = {(*xs*, *ys*). (∃ *i*. (*sinit i xs*, *sinit i ys*) ∈ *lexord R*)}

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

lemma *infseq-lexord-irrefl*:
irrefl R ⟹ *irrefl* (*infseq-lexord R*)
by (*simp add*: *irrefl-def infseq-lexord-irreflexive*)

lemma *infseq-lexord-transitive*:
assumes *trans R*
shows *trans* (*infseq-lexord R*)

unfolding *infseq-lexord-def*
proof (*rule transI*, *clarify*)
fix *xs ys zs* :: *'a infseq* **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*
by (*metis assms las*(*1*,*2*) *lexord-append lexord-sufI lexord-trans linorder-le-cases linorder-less-linear sinit-append-split sinit-len*)
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*)
moreover hence ∀ *k*<*j*. *sinit n xs* ! *k* = *sinit n zs* ! *k*
by (*metis dual-order.strict-trans ge nth-sinit sinitm*(*2*) *sinitn*(*2*))
ultimately have (*sinit n xs*, *sinit n zs*) ∈ *lexord R*
by (*metis lexord-intro-elems sinit-len sinitn*(*2*))
thus *?thesis* **by** *auto*
qed
qed

lemma *infseq-lexord-trans*:
⟦ (*xs*, *ys*) ∈ *infseq-lexord R*; (*ys*, *zs*) ∈ *infseq-lexord R*; *trans R* ⟧ ⟹ (*xs*, *zs*) ∈ *infseq-lexord R*
by (*meson infseq-lexord-transitive transE*)

lemma *infseq-lexord-antisym*:
⟦ *asym R*; (*a*, *b*) ∈ *infseq-lexord R* ⟧ ⟹ (*b*, *a*) ∉ *infseq-lexord R*
by (*auto dest*: *sinit-linear-asym-lemma2 simp add*: *infseq-lexord-def*)

lemma *infseq-lexord-asym*:
assumes *asym R*

shows *asym* (*infseq-lexord R*)
by (*simp add*: *assms asym-onI infseq-lexord-antisym*)

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

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

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

instantiation *infseq* :: (*ord*) *ord*
begin

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

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

instance ..

end

instance *infseq* :: (*order*) *order*
proof
fix *xs* :: *'a infseq*
show *xs* $\leq$ *xs* **by** (*simp add*: *less-eq-infseq-def*)
next
fix *xs ys zs* :: *'a infseq*
assume *xs* $\leq$ *ys* **and** *ys* $\leq$ *zs*
then show *xs* $\leq$ *zs*

by (*force dest*: *infseq-lexord-trans simp add*: *less-eq-infseq-def less-infseq-def trans-def*)
next
fix *xs ys* :: *'a infseq*
assume $xs \leq ys$ **and** $ys \leq xs$
then show $xs = ys$
apply (*simp-all add*: *less-eq-infseq-def less-infseq-def*)
apply *safe*
apply (*metis asym-onI case-prodD infseq-lexord-antisym mem-Collect-eq order-less-asym*)
done
next
fix *xs ys* :: *'a infseq*
show $xs < ys \longleftrightarrow xs \leq ys \wedge \neg\ ys \leq xs$
apply (*simp-all add*: *less-infseq-def less-eq-infseq-def*)
apply *safe*
apply (*simp-all add*: *infseq-lexord-irreflexive*)
apply (*metis asym-onI case-prodD infseq-lexord-antisym mem-Collect-eq order-less-imp-not-less*)
done
qed

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

lemma *infseq-lexord-mono* [*mono*]:
$(\bigwedge x\ y.\ f\ x\ y \longrightarrow g\ x\ y) \Longrightarrow (xs, ys) \in$ *infseq-lexord* $\{(x, y).\ f\ x\ y\} \longrightarrow (xs, ys) \in$ *infseq-lexord* $\{(x, y).\ g\ x\ y\}$
apply (*simp add*: *infseq-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*. $(x = y \vee (x, y) \in R) \Longrightarrow$ *sorted-rel R xs* $\Longrightarrow$ *sorted-rel R* (*x* # *xs*)

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

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

lift-definition *infseq-proj* :: *'a infseq* ⇒ *'a infseq infseq* **is**
λ *f i j*. *f* (*prod-encode* (*i*, *j*)) .

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

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

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

lemma *infseq-inj-surj*: *bij infseq-inj*
apply (*rule bijI*)
apply (*simp add*: *infseq-inj*, *safe*, *simp-all*)
apply (*metis rangeI infseq-proj-inverse*)
done

end

4 Countable Sets: Extra functions and properties

theory *Countable-Set-Extra*
imports
HOL−Library.Countable-Set-Type
Infinite-Sequence

begin

4.1 Extra syntax

notation *cempty* ({}$_c$)
notation *cin* (**infix** ∈$_c$ *50*)
notation *cUn* (**infixl** ∪$_c$ *65*)
notation *cInt* (**infixl** ∩$_c$ *70*)
notation *cDiff* (**infixl** −$_c$ *65*)
notation *cUnion* (⋃$_c$- [*900*] *900*)
notation *cimage* (**infixr** ‘$_c$ *90*)

abbreviation *csubseteq* :: *'a cset* ⇒ *'a cset* ⇒ *bool* ((-/ ⊆$_c$ -) [*51*, *51*] *50*)
where *A* ⊆$_c$ *B* ≡ *A* ≤ *B*

abbreviation *csubset* :: *'a cset* ⇒ *'a cset* ⇒ *bool* ((-/ ⊂$_c$ -) [*51*, *51*] *50*)

where $A \subset_c B \equiv A < B$

4.2 Countable set functions

setup-lifting *type-definition-cset*

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

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

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

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

syntax
-cBall :: *pttrn => 'a cset => bool => bool* ((*3*∀ *-*∈$_c$*-./ -*) [*0*, *0*, *10*] *10*)
-cBex :: *pttrn => 'a cset => bool => bool* ((*3*∃ *-*∈$_c$*-./ -*) [*0*, *0*, *10*] *10*)

translations
∀ *x*∈$_c$*A*. *P* == *CONST cBall A* (%*x*. *P*)
∃ *x*∈$_c$*A*. *P* == *CONST cBex A* (%*x*. *P*)

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

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

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

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

by (*simp add*: *cin-def*)

lemma *cset-CollI*: ⟦ $a \in_c A$; $P\ a$ ⟧ ⟹ $a \in_c \{x \in_c A.\ P\ x\}_c$
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*

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

lemma *countable-finite-power*:
countable(*A*) ⟹ *countable* {*B*. *B* ⊆ *A* ∧ *finite*(*B*)}
by (*metis Collect-conj-eq Int-commute countable-Collect-finite-subset*)

lift-definition *cInter* :: *'a cset cset* ⇒ *'a cset* ($\bigcap_c$- [*900*] *900*)
is λ*A*. *if A* = {} *then* {} *else* ⋂ *A*
using *countable-INT* [*of* - - *id*] **by** *auto*

abbreviation (*input*) *cINTER* :: *'a cset* ⇒ (*'a* ⇒ *'b cset*) ⇒ *'b cset*
where *cINTER A f* ≡ *cInter* (*cimage f A*)

lift-definition *cfinite* :: *'a cset* ⇒ *bool* **is** *finite* **.**
lift-definition *cInfinite* :: *'a cset* ⇒ *bool* **is** *infinite* **.**
lift-definition *clist* :: *'a*::*linorder cset* ⇒ *'a list* **is** *sorted-list-of-set* **.**
lift-definition *ccard* :: *'a cset* ⇒ *nat* **is** *card* **.**
lift-definition *cPow* :: *'a cset* ⇒ *'a cset cset* **is** λ *A*. {*B*. *B* ⊆$_c$ *A* ∧ *cfinite*(*B*)}
proof −
fix *A*
have {*B* :: *'a cset*. *B* ⊆$_c$ *A* ∧ *cfinite B*} = *acset* ‘ {*B* :: *'a set*. *B* ⊆ *rcset A* ∧ *finite B*}
proof −
have ⋀*x*. *rcset x* ⊆ *rcset A* ⟹ *finite* (*rcset x*) ⟹ *x* ∈ *acset* ‘ {*B*. *B* ⊆ *rcset A* ∧ *finite B*}
using *image-iff* **by** *fastforce*
thus *?thesis* **by** (*auto simp add*: *cfinite.rep-eq cin-def less-eq-cset-def countable-finite*)
qed
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
unfolding *cset-mapM-def*
by (*metis cimageE comp-the-Some cset.map-comp cset.map-id option.simps*(*3*))

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
by (*cases CCollect xs*, *auto simp add*:*CCollect-ext-def*)

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

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-seq A i = (*if* (*i* ∈ *range* (*cset-count A*) ∧ *inv-into* (*rcset A*) (*cset-count A*) *i* ∈$_c$ *A*)
then Some (*inv-into* (*rcset A*) (*cset-count A*) *i*)
else None)

lemma *cset-seq-ran*: *ran* (*cset-seq A*) = *rcset*(*A*)
apply (*simp add*: *ran-def cset-seq-def cin.rep-eq*)
apply *safe*
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 *cset2infseq* :: *'a cset* ⇒ *'a infseq*
is (λ *A i*. *if* (*i* ∈ *cset-count A* ‘ *rcset A*) *then inv-into* (*rcset A*) (*cset-count A*) *i* *else* (*SOME x*. *x* ∈$_c$ *A*)) **.**

lemma *range-cset2infseq*:
A ≠ {}$_c$ ⟹ *range* (*Rep-infseq* (*cset2infseq A*)) = *rcset A*
by (*force intro*: *someI2 simp add*: *cset2infseq.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 *cset2infseq-inj*:
inj-on cset2infseq {*A*. *A* ≠ {}$_c$}
apply (*rule inj-onI*)
apply (*simp*)
apply (*metis range-cset2infseq rcset-inject*)
done

lift-definition *nat-infseq2set* :: *nat infseq* ⇒ *nat set* **is**
λ *f*. *prod-encode* ‘ {(*x*, *f x*) | *x*. *True*} **.**

lemma *inj-nat-infseq2set*: *inj nat-infseq2set*
proof (*rule injI*, *transfer*)
fix *f g*
assume *prod-encode* ‘ {(*x*, *f x*) |*x*. *True*} = *prod-encode* ‘ {(*x*, *g x*) |*x*. *True*}
hence {(*x*, *f x*) |*x*. *True*} = {(*x*, *g x*) |*x*. *True*}
by (*simp add*: *inj-image-eq-iff*[*OF inj-prod-encode*])
thus *f* = *g*
by (*auto simp add*: *set-eq-iff*)
qed

lift-definition *bit-infseq-of-nat-set* :: *nat set* ⇒ *bool infseq*
is λ *A i*. *i* ∈ *A* **.**

lemma *bit-infseq-of-nat-set-inj*: *inj bit-infseq-of-nat-set*
apply (*rule injI*)
apply *transfer*
apply (*auto simp add*: *fun-eq-iff*)
done

lemma *bit-infseq-of-nat-cset-bij*: *bij bit-infseq-of-nat-set*
apply (*rule bijI*)
apply (*fact bit-infseq-of-nat-set-inj*)
apply *transfer*
apply (*rule surjI*)
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-infseq-of-nat-set* ∘ *infseq-inj* ∘ *cset2infseq* ∘ (λ *A*. (*bit-infseq-of-nat-set* ‘$_c$ *A*))

lemma *nat-set-cset-collapse-inj*: *inj-on nat-set-cset-collapse* {*A*. *A* ≠ {}$_c$}
proof −
have (‘$_c$) *bit-infseq-of-nat-set* ‘ {*A*. *A* ≠ {}$_c$} ⊆ {*A*. *A* ≠ {}$_c$}
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-infseq-of-nat-set-inj cset.inj-map injD inj-onI*)
apply (*rule comp-inj-on*)
apply (*metis cset2infseq-inj subset-inj-on*)
apply (*rule comp-inj-on*)
apply (*rule subset-inj-on*)

apply (*rule infseq-inj*)
apply (*simp*)
apply (*meson UNIV-I bij-imp-bij-inv bij-is-inj bit-infseq-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 (*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 (*simp add*: *csets-def*, *metis UNIV-eq-I acset-cases image-iff*)

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* ((‘$_c$) *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* ⟧ ⟹ (‘$_c$) *f* ‘ *csets A* = *csets B*
apply (*simp add*: *cimage-def csets-def image-mono map-fun-def image-comp image-Collect*)
apply *safe*
apply *auto*[*1*]
apply (*metis acset-inverse countable-subset-image mem-Collect-eq*)
done

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

5 Infinity Supplement

theory *Infinity*

imports *HOL.Real*
HOL−Library.Infinite-Set
Optics.Two
begin

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

5.1 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*)

5.2 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*) ⟹
infinite (*UNIV* :: (*'a* ⇒ *'b*) *set*)
apply (*erule contrapos-nn*)
apply (*erule finite-fun-UNIVD2*)
done

theorem *infinite-UNIV-set* [*simp*]:
infinite (*UNIV* :: *'a set*) ⟹
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*) ⟹
infinite (*UNIV* :: (*'a* × *'b*) *set*)
apply (*erule contrapos-nn*)
apply (*simp add*: *finite-prod*)
done

theorem *infinite-UNIV-prod2* [*simp*]:
infinite (*UNIV* :: *'b set*) ⟹
infinite (*UNIV* :: (*'a* × *'b*) *set*)
apply (*erule contrapos-nn*)
apply (*simp add*: *finite-prod*)
done

theorem *infinite-UNIV-sum1* [*simp*]:
infinite (*UNIV* :: *'a set*) ⟹
infinite (*UNIV* :: (*'a* + *'b*) *set*)
apply (*erule contrapos-nn*)
apply (*simp*)
done

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

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

theorem *infinite-transfer* :
infinite B ⟹ *B* ⊆ *f* ‘ *A* ⟹ *infinite A*
using *infinite-super*

apply (*blast*)
done

5.3 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

6 Positive Subtypes

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

6.1 Type Definition

typedef (**overloaded**) $'a$::{*zero*, *linorder*} *pos* = {x::$'a$. $x \geq 0$}
by *blast*

syntax
-type-pos :: *type* ⇒ *type* (-$^+$ [*999*] *999*)

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

setup-lifting *type-definition-pos*

type-synonym *preal* = *real pos*

6.2 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*]]

6.3 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 (≤) :: *'a* ⇒ *'a* ⇒ *bool* **.**
  lift-definition *less-pos* :: *'a pos* ⇒ *'a pos* ⇒ *bool*
    is (<) :: *'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 (+) by (simp)
 lift-definition minus-pos :: 'a pos ⇒ 'a pos ⇒ 'a pos
  is λx y. if y ≤ 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 times by (simp)
 instance
  apply (intro-classes; transfer; simp?)
        apply (simp add: add.assoc)
       apply (simp add: add.commute)
      apply (safe; clarsimp?) [1]
        apply (simp add: diff-diff-add)
       apply (metis add-le-cancel-left le-add-diff-inverse)
      apply (simp add: add.commute add-le-imp-le-diff)
     apply (metis add-increasing2 antisym linear)
    apply (simp add: mult.assoc)
   apply (simp add: mult.commute)
  apply (simp add: comm-semiring-class.distrib)
 apply (simp add: mult-strict-left-mono)
 apply (safe; clarsimp?) [1]
  apply (simp add: right-diff-distrib')
 apply (simp add: mult-left-mono)
  using mult-left-le-imp-le apply (fastforce)
  apply (simp add: distrib-left)
  done
end

instantiation pos :: (linordered-field) semidom-divide
begin
 lift-definition divide-pos :: 'a pos ⇒ 'a pos ⇒ 'a pos
  is divide by (simp)
 instance
  apply (intro-classes; transfer)
   apply (simp-all)
  done
end

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

lemma pos-positive [simp]: 0 ≤ (x::'a::{zero,linorder} pos)
 by (transfer, simp)
```

6.4 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 ≤ x*; *x ≤ y* ⟧ ⟹ *mk-pos x ≤ mk-pos y*
by (*transfer*, *auto*)

lemma *mk-pos-less*:
⟦ *0 ≤ x*; *x < y* ⟧ ⟹ *mk-pos x < mk-pos y*
by (*transfer*, *auto*)

lemma *real-of-pos* [*simp*]: *x ≥ 0* ⟹ *real-of-pos* (*mk-pos x*) = *x*
by (*transfer*, *simp*)

lemma *mk-pos-real-of-pos* [*simp*]: *mk-pos* (*real-of-pos x*) = *x*
by (*transfer*, *simp*)

6.5 Transfer to Reals

named-theorems *pos-transfer*

lemma *real-of-pos-0* [*pos-transfer*]:
real-of-pos 0 = 0
by (*transfer*, *auto*)

lemma *real-of-pos-1* [*pos-transfer*]:
real-of-pos 1 = 1
by (*transfer*, *auto*)

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

lemma *real-op-pos-minus* [*pos-transfer*]:
x ≥ y ⟹ *real-of-pos* (*x − y*) = *real-of-pos x − real-of-pos y*
by (*transfer*, *simp*)

lemma *real-op-pos-mult* [*pos-transfer*]:
real-of-pos (*x ∗ y*) = *real-of-pos x ∗ real-of-pos y*
by (*transfer*, *simp*)

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

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

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

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

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

end

7 Show class for code generation

theory *Haskell-Show*
imports *HOL−Library.Code-Target-Int HOL−Library.Code-Target-Nat*
begin

The aim of this theory is support code generation of serialisers for datatypes using the Haskell show class. We take inspiration from https://www.isa-afp.org/entries/Show.html, but we are more interested in code generation than being able to derive the show function for any algebraic datatype. Sometimes we give actual instance that can reasoned about in Isabelle, but mostly opaque types and code printing to Haskell instance is sufficient.

7.1 Show class

The following class should correspond to the Haskell type class Show, but currently it has only part of the signature.

class *show* =
fixes *show* :: *'a* ⇒ *String.literal* — We use *String.literal* for code generation

We set up code printing so that this class, and the constants therein, are mapped to the Haskell Show class.

code-printing
type-class *show* ⇀ (*Haskell*) *Prelude.Show*
| **constant** *show* ⇀ (*Haskell*) *Prelude.show*

7.2 Instances

We create an instance for bool, that generates an Isabelle function.

```
instantiation bool :: show
begin

fun show-bool :: bool ⇒ String.literal where
show-bool True = STR ''True'' |
show-bool False = STR ''False''

instance ..

end
```

We map the instance for bool to the built-in Haskell show, and have the code generator use the built-in class instance.

```
code-printing
  constant show-bool-inst.show-bool ⇀ (Haskell) Prelude.show
| class-instance bool :: show ⇀ (Haskell) –
```

```
instantiation unit :: show
begin

fun show-unit :: unit ⇒ String.literal where
show-unit () = STR ''()''

instance ..

end
```

```
code-printing
  constant show-unit-inst.show-unit ⇀ (Haskell) Prelude.show
| class-instance unit :: show ⇀ (Haskell) –
```

Actually, we don't really need to create the show function if all we're interested in is code generation. Here, for the *integer* instance, we omit the definition. This is because *integer* is set up to correspond to the built-in Haskell type Integer, which already has a Show instance.

```
instantiation integer :: show
begin

instance ..

end
```

For the code generator, the crucial line follows. This maps the (unspecified) Isabelle show function to the Haskell show function, which is built-in. We also specify that no instance of Show should be generated for integer, as it exists already.

```
code-printing
  constant show-integer-inst.show-integer ⇀ (Haskell) Prelude.show
```

| **class-instance** *integer* :: *show* ⇀ (*Haskell*) –

For *int*, we are effectively dealing with a packaged version of *integer* in the code generation set up. So, we simply define show in terms of the "underlying" integer using *integer-of-int*.

instantiation *int* :: *show*
begin

definition *show-int* :: *int* ⇒ *String.literal* **where**
show-int x = *show* (*integer-of-int x*)

instance ..

end

As a result, we can prove a code equation that will mean that our show instance for *int* simply calls the built-in show function for *integer*.

lemma *show-int-of-integer* [*code*]: *show* (*int-of-integer x*) = *show x*
by (*simp add*: *show-int-def*)

instantiation *nat* :: *show*
begin

definition *show-nat* :: *nat* ⇒ *String.literal* **where**
show-nat x = *show* (*integer-of-nat x*)

instance ..

end

lemma *show-Nat*: *show* (*Nat x*) = *show* (*max 0 x*)
using *Code-Target-Nat.Nat-def integer-of-nat-eq-of-nat nat-of-integer-def of-nat-of-integer show-nat-def* **by** *presburger*

instantiation *String.literal* :: *show*
begin

definition *show-literal* :: *String.literal* ⇒ *String.literal* **where**
show-literal x = *STR* ''''+ *x* + *STR* ''''

instance ..

end

code-printing
constant *show-literal-inst.show-literal* ⇀ (*Haskell*) *Prelude.show*
| **class-instance** *String.literal* :: *show* ⇀ (*Haskell*) –

instantiation *prod* :: (*show*, *show*) *show*
begin

instance ..

end

code-printing
  constant *show-prod-inst.show-prod* ⇀ (*Haskell*) *Prelude.show*
| **class-instance** *prod* :: *show* ⇀ (*Haskell*) –

instantiation *list* :: (*show*) *show*
begin

instance ..

end

code-printing
  constant *show-list-inst.show-list* ⇀ (*Haskell*) *Prelude.show*
| **class-instance** *list* :: *show* ⇀ (*Haskell*) –

end

8 Enumeration Types

theory *Enum-Type*
  imports *Haskell-Show*
  keywords *enumtype* :: *thy-defn*
begin

ML-file ‹*Enum-Type.ML*›

ML ‹
val - =
  Outer-Syntax.command •{*command-keyword enumtype*} *define enumeration types*
    ((*Parse.name* –– (•{*keyword =*} |–– *Scan.repeat* (*Parse.name* ––| •{*keyword* |}) –– *Parse.name*)) >> (*fn* (*tname*, (*cs*, *c*)) => *Toplevel.theory* (*Enum-Type.enum-type tname* (*cs* • [*c*])))));
›

declare *UNIV-enum* [*code-unfold*]

end

9 Default Class Instances for Record Types

theory *Record-Default-Instance*

imports *Main*
keywords *record-default* :: *thy-defn*
begin

ML-file ‹*Record-Default-Instance.ML*›

ML ‹
val - =
Outer-Syntax.command •{*command-keyword record-default*} *define default instances for records*
(*Parse.name >>* (*fn tname => Toplevel.theory* (*Record-Default-Instance.mk-rec-default-instance tname*)));
›

end

10 Defining Declared Constants

theory *Def-Const*
imports *Main*
keywords *def-consts* :: *thy-defn*
begin

Add a simple command to define previously declared polymorphic constants. This is particularly useful for handling given sets in Z.

ML ‹

structure Def-Const =
struct

fun def-const (*n*, *v*) *thy* =
let val Const (*pn*, *typ*) = *Proof-Context.read-const* {*proper* = *false*, *strict* = *false*} (*Named-Target.theory-init thy*) *n*
val ctx = *Overloading.overloading* [(*n*, (*pn*, *dummyT*), *false*)] *thy*
val pt = *Syntax.check-term ctx* (*Type.constraint typ* (*Syntax.parse-term ctx v*))
in (*Local-Theory.exit-global o snd o Local-Theory.define* (((*Binding.name n*), *NoSyn*), ((*Binding.name* (*n* ^ *-def*), [*Attrib.check-src* •{*context*} (*Token.make-src* (*code*, *Position.none*) [])]), *pt*))) *ctx*
end
val def-consts = fold def-const

end;

Outer-Syntax.command •{*command-keyword def-consts*} *define a declared constant*
(*Scan.repeat1* ((*Parse.name* −−| •{*keyword* =}) −− *Parse.term*) >> (*Toplevel.theory o Def-Const.def-consts*));
›

end

11 Polymorphic Overriding Operator

theory *Overriding*
imports *Main*
begin

We here use type classes to create the overriding operator and instantiate it for relations, partial function, and finite functions.

class *oplus* =
fixes *oplus* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\oplus$ *65*)

class *compatible* =
fixes *compatible* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $\#\#$ *60*)
assumes *compatible-sym*: $x \mathbin{\#\#} y \Longrightarrow y \mathbin{\#\#} x$

unbundle *lattice-syntax*

class *override* = *oplus* + *bot* + *compatible* +
assumes *compatible-zero* [*simp*]: $x \mathbin{\#\#} \bot$
and *override-idem* [*simp*]: $P \oplus P = P$
and *override-assoc*: $P \oplus (Q \oplus R) = (P \oplus Q) \oplus R$
and *override-lzero* [*simp*]: $\bot \oplus P = P$
and *override-comm*: $P \mathbin{\#\#} Q \Longrightarrow P \oplus Q = Q \oplus P$
and *override-compat*: $[\![P \mathbin{\#\#} Q;\ (P \oplus Q) \mathbin{\#\#} R]\!] \Longrightarrow P \mathbin{\#\#} R$
and *override-compatI*: $[\![P \mathbin{\#\#} Q;\ P \mathbin{\#\#} R;\ Q \mathbin{\#\#} R]\!] \Longrightarrow (P \oplus Q) \mathbin{\#\#} R$
begin

lemma *override-rzero* [*simp*]: $P \oplus \bot = P$
by (*metis compatible-zero override-comm override-lzero*)

lemma *override-compat'*: $[\![P \mathbin{\#\#} Q;\ (P \oplus Q) \mathbin{\#\#} R]\!] \Longrightarrow Q \mathbin{\#\#} R$
by (*metis compatible-sym override-comm override-compat*)

lemma *override-compat-iff*: $P \mathbin{\#\#} Q \Longrightarrow (P \oplus Q) \mathbin{\#\#} R \longleftrightarrow (P \mathbin{\#\#} R) \wedge (Q \mathbin{\#\#} R)$
by (*meson override-compat override-compatI override-compat'*)

end

end

12 Relational Universe

theory *Relation-Extra*
imports *HOL−Library.FuncSet HOL−Library.AList List-Extra Overriding*
begin

no-notation *SCons* (**infixr** ‹##› *65*)

This theory develops a universe for a Z-like relational language, including the core operators of the ISO Z metalanguage. Much of this already exists in *HOL.Relation*, but we need to add some additional functions and sets. It characterises relations, partial functions, total functions, and finite functions.

12.1 Type Syntax

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

syntax
-rel-type :: *type* ⇒ *type* ⇒ *type* (**infixr** ↔ *0*)

translations
(*type*) *'a* ↔ *'b* == (*type*) (*'a* × *'b*) *set*

Setup pretty printing for homogeneous relations.

print-translation ‹
let fun tr' ctx [*Const* (•{*type-syntax prod*},-) $ *a* $ *b*] =
  if a = *b then Syntax.const* •{*type-syntax rel*} $ *a else raise Match*;
in [(•{*type-syntax set*},*tr'*)]
end
›

12.2 Notation for types as sets

definition *TUNIV* (*a*::*'a itself*) = (*UNIV* :: *'a set*)

syntax *-tvar* :: *type* ⇒ *logic* ([-]$_T$)
translations [*'a*]$_T$ == *CONST TUNIV TYPE*(*'a*)

lemma *TUNIV-mem* [*simp*]: *x* ∈ [*'a*]$_T$
  by (*simp add*: *TUNIV-def*)

12.3 Relational Function Operations

These functions are all adapted from their ISO Z counterparts.

definition *rel-apply* :: (*'a* ↔ *'b*) ⇒ *'a* ⇒ *'b* (-'(-')$_r$ [*999*,*0*] *999*) **where**
rel-apply R x = (*if* (∃! *y*. (*x*, *y*) ∈ *R*) *then THE y*. (*x*, *y*) ∈ *R else undefined*)

If there exists a unique e_3 such that (e_2, e_3) is in e_1, then the value of $e_1(e_2)_r$ is e_3, otherwise each $e_1(e_2)_r$ has a fixed but unknown value (i.e. *undefined*).

definition *rel-domres* :: *'a set* ⇒ (*'a* ↔ *'b*) ⇒ *'a* ↔ *'b* (**infixr** ◁$_r$ *85*) **where**
rel-domres A R = {*p* ∈ *R*. *fst p* ∈ *A*}

lemma *rel-domres-math-def*: $A \lhd_r R = \{(k, v) \in R.\ k \in A\}$
by (*auto simp add*: *rel-domres-def*)

Domain restriction ($A \lhd_r R$ contains the set of pairs in R, such that the first element of every such pair in in A.

definition *rel-ranres* :: $('a \leftrightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a \leftrightarrow 'b$ (**infixl** $\rhd_r$ *85*) **where**
rel-ranres R A = $\{p \in R.\ snd\ p \in A\}$

We employ some type class trickery to enable a polymorphic operator for override that can instantiate *'a set*, which is needed for relational overriding. The following class's sole purpose is to allow pairs to be the only valid instantiation element for the set type.

class *pre-restrict* =
fixes *cmpt* :: *'a set* ⇒ *'a set* ⇒ *bool*
and *res* :: *'a set* ⇒ *'a set* ⇒ *'a set*

instantiation *prod* :: (*type*, *type*) *pre-restrict*
begin

Relations are compatible is they agree on the values for maplets they both possess.

definition *cmpt-prod* :: $('a \leftrightarrow 'b) \Rightarrow ('a \leftrightarrow 'b) \Rightarrow bool$
where [*simp*]: *cmpt-prod R S* = $((Domain\ R) \lhd_r S = (Domain\ S) \lhd_r R)$
definition *res-prod* :: $('a \leftrightarrow 'b) \Rightarrow ('a \leftrightarrow 'b) \Rightarrow 'a \leftrightarrow 'b$
where [*simp*]: *res-prod R S* = $((-\ Domain\ S) \lhd_r R) \cup S$
instance ..
end

instantiation *set* :: (*pre-restrict*) *oplus*
begin

definition *oplus-set* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
oplus-set = *res*

instance ..

end

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

Relational update adds a new pair to a relation.

definition *rel-disjoint* :: $('a \leftrightarrow 'b\ set) \Rightarrow bool$ **where**
rel-disjoint f = $(\forall\ p \in f.\ \forall\ q \in f.\ p \neq q \longrightarrow snd\ p \cap snd\ q = \{\})$

definition *rel-partitions* :: $('a \leftrightarrow 'b\ set) \Rightarrow 'b\ set \Rightarrow bool$ **where**
rel-partitions f a = $(rel\text{-}disjoint\ f \wedge \bigcup\ (Range\ f) = a)$

12.4 Domain laws

declare *Domain-Un-eq* [*simp*]

lemma *Domain-Preimage*: $Domain\ P = P^{-1}\ ``\ UNIV$
by (*simp add*: *Image-def Domain-unfold*)

lemma *Domain-relcomp* [*simp*]: $Domain\ (P\ O\ Q) = Domain\ (P \rhd_r Domain\ Q)$
by (*force simp add*: *Domain-iff rel-ranres-def*)

lemma *Domain-relcomp-conv*: $Domain\ (P\ O\ Q) = (P^{-1}\ ``\ Domain(Q))$
by (*simp add*: *Domain-Preimage converse-relcomp relcomp-Image*)

lemma *Domain-set*: $Domain\ (set\ xs) = set\ (map\ fst\ xs)$
by (*simp add*: *Domain-fst*)

12.5 Range laws

lemma *Range-Image*: $Range\ P = P\ ``\ UNIV$
by (*simp add*: *Image-def Range-iff set-eq-iff*)

lemma *Range-relcomp*: $Range\ (P\ O\ Q) = (Q\ ``\ Range(P))$
by (*simp add*: *Range-Image relcomp-Image*)

12.6 Domain Restriction

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

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

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

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

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

lemma *rel-domres-compl-disj*: $A \cap Domain\ P = \{\} \Longrightarrow (-\ A) \lhd_r P = P$
by (*auto simp add*: *rel-domres-def*)

lemma *rel-domres-notin-Dom*: $k \notin Domain(f) \Longrightarrow (-\ \{k\}) \lhd_r f = f$
by (*simp add*: *rel-domres-compl-disj*)

lemma *rel-domres-Id-on*: $A \lhd_r R = Id\text{-}on\ A\ O\ R$
by (*auto simp add*: *rel-domres-def Id-on-def relcomp-unfold*)

lemma *rel-domres-insert* [*simp*]:
$A \lhd_r$ *insert* (k, v) $R =$ (*if* $(k \in A)$ *then insert* (k, v) $(A \lhd_r R)$ *else* $A \lhd_r R$)
by (*auto simp add*: *rel-domres-def*)

lemma *rel-domres-insert-set* [*simp*]: $x \notin$ *Domain* $P \Longrightarrow$ (*insert* x A) $\lhd_r$ $P = A \lhd_r P$
by (*auto simp add*: *rel-domres-def*)

lemma *Image-as-rel-domres*: R `` $A =$ *Range* $(A \lhd_r R)$
by (*auto simp add*: *rel-domres-def*)

lemma *rel-domres-Un* [*simp*]: $A \lhd_r (S \cup R) = (A \lhd_r S) \cup (A \lhd_r R)$
by (*auto simp add*: *rel-domres-def*)

12.7 Range Restriction

lemma *rel-ranres-UNIV* [*simp*]: $P \rhd_r$ *UNIV* $= P$
by (*auto simp add*: *rel-ranres-def*)

lemma *rel-ranres-Un* [*simp*]: $(P \cup Q) \rhd_r A = (P \rhd_r A \cup Q \rhd_r A)$
by (*auto simp add*: *rel-ranres-def*)

lemma *rel-ranres-relcomp* [*simp*]: $(P$ *O* $Q) \rhd_r A = P$ *O* $(Q \rhd_r A)$
by (*auto simp add*: *rel-ranres-def relcomp-unfold prod.case-eq-if*)

lemma *conv-rel-domres* [*simp*]: $(P \rhd_r A)^{-1} = A \lhd_r P^{-1}$
by (*auto simp add*: *rel-domres-def rel-ranres-def*)

lemma *rel-ranres-le*: $A \subseteq B \Longrightarrow f \rhd_r A \leq f \rhd_r B$
by (*simp add*: *Collect-mono-iff in-mono rel-ranres-def*)

12.8 Relational Override

class *restrict* = *pre-restrict* +
assumes *cmpt-sym*: *cmpt* P $Q \Longrightarrow$ *cmpt* Q P
and *cmpt-empty*: *cmpt* $\{\}$ P
assumes *res-idem*: *res* P $P = P$
and *res-assoc*: *res* P (*res* Q R) = *res* (*res* P Q) R
and *res-lzero*: *res* $\{\}$ $P = P$
and *res-comm*: *cmpt* P $Q \Longrightarrow$ *res* P $Q =$ *res* Q P
and *res-cmpt*: $[\![$ *cmpt* P Q; *cmpt* (*res* P Q) R $]\!] \Longrightarrow$ *cmpt* P R
and *res-cmptI*: $[\![$ *cmpt* P Q; *cmpt* P R; *cmpt* Q R $]\!] \Longrightarrow$ *cmpt* (*res* P Q) R

lemma *res-cmpt-rel*: *cmpt* $(P :: 'a \leftrightarrow 'b)$ $Q \Longrightarrow$ *cmpt* (*res* P Q) $R \Longrightarrow$ *cmpt* P R
by (*fastforce simp add*: *rel-domres-def Domain-iff*)

instance *prod* :: (*type*, *type*) *restrict*
by (*intro-classes*, *simp-all only*: *res-cmpt-rel*, *auto simp add*: *rel-domres-def Domain-unfold*)

instantiation *set* :: (*restrict*) *override*
begin
definition *compatible-set* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
compatible-set = *cmpt*

instance
apply (*intro-classes*, *simp-all add*: *oplus-set-def compatible-set-def res-idem res-assoc res-lzero cmpt-sym cmpt-empty res-comm res-cmptI*)
using *res-cmpt* **apply** *blast*
done
end

lemma *override-eq*: $R \oplus S = ((-\ Domain\ S) \lhd_r R) \cup S$
by (*simp add*: *oplus-set-def*)

lemma *Domain-rel-override* [*simp*]: $Domain\ (R \oplus S) = Domain(R) \cup Domain(S)$
by (*auto simp add*: *oplus-set-def*)

lemma *Range-rel-override*: $Range(R \oplus S) \subseteq Range(R) \cup Range(S)$
by (*auto simp add*: *oplus-set-def rel-domres-def*)

lemma *compatible-rel*: $R\ \#\#\ S = (Domain\ R \lhd_r S = Domain\ S \lhd_r R)$
by (*simp add*: *compatible-set-def*)

lemma *compatible-relI*: $Domain\ R \lhd_r S = Domain\ S \lhd_r R \Longrightarrow R\ \#\#\ S$
by (*simp add*: *compatible-rel*)

12.9 Functional Relations

abbreviation *functional* :: (*'a* ↔ *'b*) ⇒ *bool* **where**
functional R ≡ *single-valued R*

lemma *functional-def*: *functional R* ⟷ *inj-on fst R*
by (*force simp add*: *single-valued-def inj-on-def*)

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

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

lemma *functional-apply-iff*: $functional\ R \Longrightarrow (x, y) \in R \longleftrightarrow (x \in Domain\ R \wedge R(x)_r = y)$

by (*auto simp add*: *functional-apply*)

lemma *functional-elem*:
assumes *functional R x* ∈ *Domain*(*R*)
shows (*x*, *R*(*x*)$_r$) ∈ *R*
using *assms*(*1*) *assms*(*2*) *functional-apply* **by** *fastforce*

lemma *functional-override* [*intro*!]: ⟦ *functional R*; *functional S* ⟧ ⟹ *functional* (*R* ⊕ *S*)
by (*auto simp add*: *functional-algebraic oplus-set-def rel-domres-def*)

lemma *functional-union* [*intro*!]: ⟦ *functional R*; *functional S*; *R* ## *S* ⟧ ⟹ *functional* (*R* ∪ *S*)
by (*metis functional-override le-sup-iff override-comm override-eq single-valued-subset subsetI*)

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* ⟷ *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

definition *fun-rel* :: (′*a* ⇒ ′*b*) ⇒ (′*a* ↔ ′*b*) **where**
fun-rel f = {(*x*, *y*). *y* = *f x*}

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

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

by (*simp add*: *fun-rel-def rel-apply-def*)

Make a relation functional by removing any pairs that have duplicate distinct values.

definition *mk-functional* :: ('*a* ↔ '*b*) ⇒ ('*a* ↔ '*b*) **where**
mk-functional R = {(*x*, *y*) ∈ *R*. ∀ *y'*. (*x*, *y'*) ∈ *R* ⟶ *y* = *y'*}

lemma *mk-functional* [*simp*]: *functional* (*mk-functional R*)
by (*auto simp add*: *mk-functional-def single-valued-def*)

lemma *mk-functional-fp*: *functional R* ⟹ *mk-functional R* = *R*
by (*auto simp add*: *mk-functional-def single-valued-def*)

lemma *mk-functional-idem*: *mk-functional* (*mk-functional R*) = *mk-functional R*
using *mk-functional mk-functional-fp* **by** *blast*

lemma *mk-functional-subset* [*simp*]: *mk-functional R* ⊆ *R*
by (*auto simp add*: *mk-functional-def*)

lemma *Domain-mk-functional*: *Domain* (*mk-functional R*) ⊆ *Domain R*
by (*auto simp add*: *mk-functional-def*)

definition *single-valued-dom* :: ('*a* × '*b*) *set* ⇒ '*a set* **where**
single-valued-dom R = {*x* ∈ *Domain*(*R*). ∃ *y*. *R* `` {*x*} = {*y*}}

lemma *mk-functional-single-valued-dom*: *mk-functional R* = *single-valued-dom R* ◁$_r$ *R*
by (*auto simp add*: *mk-functional-def single-valued-dom-def rel-domres-math-def Domain-unfold*)
(*metis Image-singleton-iff singletonD*)

12.10 Left-Total Relations

definition *left-totalr-on* :: '*a set* ⇒ ('*a* ↔ '*b*) ⇒ *bool* **where**
left-totalr-on A R ⟷ (∀ *x*∈*A*. ∃ *y*. (*x*, *y*) ∈ *R*)

abbreviation *left-totalr R* ≡ *left-totalr-on UNIV R*

lemma *left-totalr-algebraic*: *left-totalr R* ⟷ *Id* ⊆ *R O R*$^{-1}$
by (*auto simp add*: *left-totalr-on-def*)

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

12.11 Injective Relations

definition *injective* :: ('*a* ↔ '*b*) ⇒ *bool* **where**
injective R = (*functional R* ∧ *R O R*$^{-1}$ ⊆ *Id*)

lemma *injectiveI*:
assumes *functional R* $\bigwedge$ *x y*. $[\![x \in Domain\ R;\ y \in Domain\ R;\ R(x)_r = R(y)_r]\!] \Longrightarrow x = y$
shows *injective R*
using *assms* **by** (*auto simp add*: *injective-def functional-apply-iff*)

12.12 Relation Sets

definition *rel-typed* :: *'a set* ⇒ *'b set* ⇒ (*'a* ↔ *'b*) *set* (**infixr** ↔ *55*) **where**
rel-typed A B = {*R*. *Domain*(*R*) ⊆ *A* ∧ *Range*(*R*) ⊆ *B*} — Relations

lemma *rel-typed-intro*: $[\![Domain(R) \subseteq A;\ Range(R) \subseteq B]\!] \Longrightarrow R \in A \leftrightarrow B$
by (*simp add*: *rel-typed-def*)

definition *rel-pfun* :: *'a set* ⇒ *'b set* ⇒ (*'a* ↔ *'b*) *set* (**infixr** →$_p$ *55*) **where**
rel-pfun A B = {*R*. *R* ∈ *A* ↔ *B* ∧ *functional R*} — Partial Functions

lemma *rel-pfun-intro*: $[\![R \in A \leftrightarrow B;\ functional\ R]\!] \Longrightarrow R \in A \rightarrow_p B$
by (*simp add*: *rel-pfun-def*)

definition *rel-tfun* :: *'a set* ⇒ *'b set* ⇒ (*'a* ↔ *'b*) *set* (**infixr** →$_t$ *55*) **where**
rel-tfun A B = {*R*. *R* ∈ *A* →$_p$ *B* ∧ *left-totalr R*} — Total Functions

definition *rel-ffun* :: *'a set* ⇒ *'b set* ⇒ (*'a* ↔ *'b*) *set* (**infixr** →$_f$ *55*) **where**
rel-ffun A B = {*R*. *R* ∈ *A* →$_p$ *B* ∧ *finite*(*Domain R*)} — Finite Functions

12.13 Closure Properties

These can be seen as typing rules for relational functions

named-theorems *rclos*

lemma *rel-ffun-is-pfun* [*rclos*]: *R* ∈ *rel-ffun A B* ⟹ *R* ∈ *A* →$_p$ *B*
by (*simp add*: *rel-ffun-def*)

lemma *rel-tfun-is-pfun* [*rclos*]: *R* ∈ *A* →$_t$ *B* ⟹ *R* ∈ *A* →$_p$ *B*
by (*simp add*: *rel-tfun-def*)

lemma *rel-pfun-is-typed* [*rclos*]: *R* ∈ *A* →$_p$ *B* ⟹ *R* ∈ *A* ↔ *B*
by (*simp add*: *rel-pfun-def*)

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

lemma *rel-pfun-apply* [*rclos*]: $[\![x \in Domain(R);\ R \in A \rightarrow_p B]\!] \Longrightarrow R(x)_r \in B$
using *functional-apply* **by** (*fastforce simp add*: *rel-pfun-def rel-typed-def*)

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

lemma *rel-typed-insert* [*rclos*]: ⟦ $R \in A \leftrightarrow B$; $x \in A$; $y \in B$ ⟧ ⟹ *insert* (x, y) $R \in A \leftrightarrow B$
by (*simp add*: *rel-typed-def*)

lemma *rel-pfun-insert* [*rclos*]: ⟦ $R \in A \rightarrow_p B$; $x \in A$; $y \in B$; $x \notin Domain(R)$ ⟧ ⟹ *insert* (x, y) $R \in A \rightarrow_p B$
by (*auto intro*: *rclos simp add*: *rel-pfun-def*)

lemma *rel-pfun-override* [*rclos*]: ⟦ $R \in A \rightarrow_p B$; $S \in A \rightarrow_p B$ ⟧ ⟹ $(R \oplus S) \in A \rightarrow_p B$
apply (*rule rel-pfun-intro*)
apply (*rule rel-typed-intro*)
apply (*simp-all add*: *rel-pfun-def rel-typed-def*, *safe*)
apply (*metis* (*no-types*, *opaque-lifting*) *Range.simps Range-Un-eq Range-rel-override Un-iff rev-subsetD*)
done

12.14 Code Generation

lemma *rel-conv-alist* [*code*]: $(set\ xs)^{-1} = set\ (map\ (\lambda(x, y).\ (y, x))\ xs)$
by (*induct xs*, *auto*)

lemma *rel-domres-alist* [*code*]: $A \lhd_r set\ xs = set\ (AList.restrict\ A\ xs)$
by (*induct xs*, *simp-all*, *safe*, *simp-all*)

lemma *Image-alist* [*code*]: *set xs* `` $A = set\ (map\ snd\ (AList.restrict\ A\ xs))$
by (*simp add*: *Image-as-rel-domres rel-domres-alist Range-snd*)

lemma *Collect-set*: $\{x \in set\ xs.\ P\ x\} = set\ (filter\ P\ xs)$
by *auto*

lemma *single-valued-dom-alist* [*code*]:
single-valued-dom (*set xs*) = *set* (*filter* ($\lambda x.$ *length* (*remdups* (*map snd* (*AList.restrict* $\{x\}$ *xs*))) = *1*) (*map fst xs*))
by (*simp only*: *single-valued-dom-def set-map*[*THEN sym*] *Image-alist Domain-set set-singleton-iff list-singleton-iff Collect-set*)

lemma *AList-restrict-in-dom*: *AList.restrict* (*set* (*filter P* (*map fst xs*))) *xs* = *filter* (λ (x, y). *P x*) *xs*
by (*auto intro*: *filter-cong simp add*: *Domain.intros fst-eq-Domain AList.restrict-eq*)

lemma *mk-functional-alist* [*code*]:
mk-functional (*set xs*) = *set* (*filter* (λ (*x*,*y*). *length* (*remdups* (*map snd* (*AList.restrict* $\{x\}$ *xs*))) = *1*) *xs*)
by (*simp only*: *mk-functional-single-valued-dom rel-domres-alist single-valued-dom-alist AList-restrict-in-dom*)

lemma *rel-apply-set* [*code*]:

```
  rel-apply (set xs) k =
  (let ys = filter (λ (k', v). k = k') xs in
   if (length ys > 0 ∧ ys = replicate (length ys) (hd ys)) then snd (hd ys) else
undefined)
proof −
  let ?ys = filter (λ(k', v). k = k') xs
  have 1: ⟦ ?ys ≠ []; ?ys = replicate (length ?ys) (hd ?ys) ⟧ ⟹
        set xs(k)_r = snd (hd ?ys)
  proof −
    assume ys: ?ys ≠ [] ?ys = replicate (length ?ys) (hd ?ys)
    have kmem: ⋀ y. (k, y) ∈ set xs ⟷ (k, y) ∈ set ?ys
      by simp
    from ys obtain v where v: (k, v) ∈ set xs
      using hd-in-set by fastforce
    hence ys':?ys = replicate (length ?ys) (k, v)
     by (metis (mono-tags) case-prodI filter-set in-set-replicate member-filter ys(2))
    hence snd (hd ?ys) = v
      by (metis hd-replicate replicate-0 snd-conv ys(1))
    moreover have (THE y. (k, y) ∈ set xs) = v
     by (metis (no-types, lifting) v in-set-replicate kmem snd-conv the-equality ys')
    moreover have (∃!y. (k, y) ∈ set xs)
      by (metis Pair-inject v in-set-replicate kmem ys')
    ultimately show set xs(k)_r = snd (hd ?ys)
      by (simp add: rel-apply-def)
  qed

  have 2: ?ys = [] ⟹ set xs(k)_r = undefined
  proof −
    assume filter (λ(k', v). k = k') xs = []
    hence ∄ v. (k, v) ∈ set xs
      by (metis (mono-tags, lifting) case-prodI filter-empty-conv)
    thus set xs(k)_r = undefined
      by (auto simp add: rel-apply-def)
  qed

  have 3: ?ys ≠ replicate (length ?ys) (hd ?ys) ⟹ set xs(k)_r = undefined
  proof −
    assume ys: ?ys ≠ replicate (length ?ys) (hd ?ys)
    have keys: ∀ (k', v') ∈ set ?ys. k' = k
      by auto
    show set xs(k)_r = undefined
    proof (cases length ?ys = 0)
      case True
      then show ?thesis
        using ys by fastforce
    next
      case False
      hence length ?ys > 1
        by (metis hd-in-set in-set-conv-nth length-0-conv less-one linorder-neqE-nat
```

replicate-length-same ys)
    have *fst* (*hd ?ys*) = *k*
      using *False hd-in-set* **by** *force*
    have ¬(∀ (*k*, *v*) ∈ *set ?ys*. *v* = *snd* (*hd ?ys*))
    proof
      assume (∀ (*k*, *v*) ∈ *set ?ys*. *v* = *snd* (*hd ?ys*))
      hence (∀ *p* ∈ *set ?ys*. *p* = (*k*, *snd* (*hd ?ys*)))
        by *fastforce*
      hence *?ys* = *replicate* (*length ?ys*) (*hd ?ys*)
        by (*metis False length-0-conv list.set-sel*(*1*) *replicate-length-same*)
      thus *False*
        using *ys* **by** *blast*
    qed
    then obtain *v* **where** (*k*, *v*) ∈ *set ?ys v* ≠ *snd* (*hd ?ys*)
      by *fastforce*
    hence (¬ (∃!*y*. (*k*, *y*) ∈ *set xs*))
      using *False list.set-sel*(*1*) **by** *fastforce*
    then show *?thesis*
      by (*simp add*: *rel-apply-def*)
  qed
 qed
 from *1 2 3* **show** *?thesis*
  by (*simp add*: *Let-unfold*)
qed

end

13 Map Type: extra functions and properties

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

13.1 Extensionality and Update

lemma *map-eq-iff*: *f* = *g* ⟷ (∀ *x* ∈ *dom*(*f*) ∪ *dom*(*g*). *f x* = *g x*)
 by (*auto simp add*: *fun-eq-iff*)

13.2 Graphing Maps

definition *map-graph* :: (′*a* ⇀ ′*b*) ⇒ (′*a* ↔ ′*b*) **where**
map-graph f = {(*x*,*y*) | *x y*. *f x* = *Some y*}

definition *graph-map* :: (′*a* ↔ ′*b*) ⇒ (′*a* ⇀ ′*b*) **where**
graph-map g = (λ *x*. *if* (*x* ∈ *fst* ‘ *g*) *then Some* (*SOME y*. (*x*,*y*) ∈ *g*) *else None*)

definition *graph-map'* :: $('a \leftrightarrow 'b) \rightharpoonup ('a \rightharpoonup 'b)$ **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*)
by (*metis countable-image graph-def graph-eq-to-snd-dom map-graph-def*)

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

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

lemma *graph-map-insert* [*simp*]: $[\![$*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 *graph-map-set*: *functional* (*set xs*) $\Longrightarrow$ *graph-map* (*set xs*) = *map-of xs*
by (*induct xs*; *force*)

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

lemma *ran-map-add-subset*:
ran (*x* ++ *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 *functional-insert*: *functional* (*insert a R*) $\Longrightarrow$ *functional R*
by (*simp add*: *single-valued-def*)

lemma *Domain-insert*: *Domain* (*insert a R*) = *insert* (*fst a*) (*Domain R*)
by (*simp add*: *Domain-fst*)

lemma *card-map-graph*: ⟦ *finite R*; *functional R* ⟧ ⟹ *card R* = *card* (*Domain R*)
by (*induct R rule*: *finite.induct*, *simp-all add*: *functional-insert card-insert-if Domain-insert finite-Domain*)
(*metis DiffD1 Diff-insert-absorb DomainE Map-Extra.Domain-insert insertI1 insert-Diff prod.collapse single-valued-def*)

lemma *graph-map-inv* [*simp*]: *functional g* ⟹ *map-graph* (*graph-map g*) = *g*
apply (*simp add*:*map-graph-def graph-map-def functional-def*, *safe*)
apply (*metis* (*lifting*, *no-types*) *image-iff option.distinct*(*1*) *option.inject someI surjective-pairing*)
apply (*simp add*:*inj-on-def*)
apply *safe*
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* ⟹ *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* ⟷ *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* $g)$
by (*metis graph-def graph-map-comp map-graph-def*)

lemma *rel-comp-map*: $R\ O$ *map-graph* $f = (\lambda\ p.\ (fst\ p,\ the\ (f\ (snd\ p))))\ `\ (R \rhd_r dom(f))$
by (*force simp add*: *map-graph-def relcomp-unfold rel-ranres-def image-def dom-def*)

lemma *map-graph-update*: *map-graph* $(f(k \mapsto v)) =$ *insert* $(k,\ v)\ ((-\ \{k\}) \lhd_r$ *map-graph* $f)$
by (*simp add*: *map-graph-def rel-domres-def*, *safe*, *simp-all*, *metis option.sel*)

13.3 Map Application

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

13.4 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*, *simp-all*, *metis not-None-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*)

13.5 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 (*simp add*: *dom-def preimage-def*, *safe*, *simp-all*)
apply (*meson map-comp-Some-iff*)
apply (*metis map-comp-def option.case-eq-if option.distinct(1)*)
done

lemma *countable-preimage*:
assumes *countable A inj-on f* (*preimage f A*)
shows *countable* (*preimage f A*)
proof −
obtain *g* :: *'a* ⇒ *nat* **where** *g*: *inj-on g A*
using *assms*(*1*) **by** *blast*
have *inj-on* (*g* ∘ *the* ∘ *f*) (*preimage f A*)
proof (*rule inj-onI*)
fix *x y*
assume *x* ∈ *preimage f A y* ∈ *preimage f A* (*g* ∘ *the* ∘ *f*) *x* = (*g* ∘ *the* ∘ *f*) *y*
with *assms g* **show** *x* = *y*
unfolding *preimage-def* **by** (*metis* (*lifting*) *comp-apply domIff inj-onD mem-Collect-eq option.expand*)
qed
thus *?thesis*
by (*simp add*: *countableI*)
qed

13.6 Minus operation for maps

definition *map-minus* :: (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'b*) (**infixl** −− *100*)
where *map-minus f g* = (λ *x*. *if* (*f x* = *g x*) *then None else f x*)

lemma *map-minus-apply* [*simp*]: *y* ∈ *dom*(*f* −− *g*) ⟹ (*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*) ∈$_m$ *f* ++ *g* ⟷ ((*x* ∉ *dom*(*g*) ∧ (*x*, *y*) ∈$_m$ *f*) ∨ (*x*, *y*) ∈$_m$ *g*)
by (*auto simp add*: *map-add-Some-iff*)

lemma *map-member-minus*:
(*x*, *y*) ∈$_m$ *f* −− *g* ⟷ (*x*, *y*) ∈$_m$ *f* ∧ (¬ (*x*, *y*) ∈$_m$ *g*)
by (*auto simp add*: *map-minus-def*)

lemma *map-minus-plus-commute*:
dom(*g*) ∩ *dom*(*h*) = {} ⟹ (*f* −− *g*) ++ *h* = (*f* ++ *h*) −− *g*
apply (*rule map-ext*)
apply (*simp add*: *map-member-plus map-member-minus 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 (*simp add*: *map-minus-def map-graph-def*, *safe*, *simp-all*, (*meson option.distinct*(*1*))+)

lemma *map-minus-common-subset*:
⟦ *h* ⊆$_m$ *f*; *h* ⊆$_m$ *g* ⟧ ⟹ (*f* −− *h* = *g* −− *h*) = (*f* = *g*)

by (*auto simp add*: *map-eq-graph map-graph-minus map-le-graph*)

13.7 Map Bind

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

lemma *option-bindSomeE* [*elim*!]:
⟦ *X* >>= *F* = *Some*(*v*); ⋀ *x*. ⟦ *X* = *Some*(*x*); *F*(*x*) = *Some*(*v*) ⟧ ⟹ *P* ⟧ ⟹ *P*
by (*cases X*, *auto*)

lemma *option-bindSomeI* [*intro*]:
⟦ *X* = *Some*(*x*); *F*(*x*) = *Some*(*y*) ⟧ ⟹ *X* >>= *F* = *Some*(*y*)
by (*simp*)

lemma *ifSomeE* [*elim*]: ⟦ (*if c then Some*(*x*) *else None*) = *Some*(*y*); ⟦ *c*; *x* = *y* ⟧ ⟹ *P* ⟧ ⟹ *P*
by (*cases c*, *auto*)

13.8 Range Restriction

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

definition *ran-restrict-map* :: ('*a* ⇀ '*b*) ⇒ '*b set* ⇒ '*a* ⇀ '*b* (-⇂- [*111*,*110*] *110*)
where
ran-restrict-map f B = (λ*x*. *do* { *v* <− *f*(*x*); *if* (*v* ∈ *B*) *then Some*(*v*) *else None* })

lemma *ran-restrict-alt-def*: $f⇂_B$ = (λ *x*. *if x* ∈ *dom*(*f*) ∧ *the*(*f*(*x*)) ∈ *B then f x else None*)
by (*auto simp add*: *ran-restrict-map-def fun-eq-iff bind-eq-None-conv*)

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

lemma *ran-restrict-ran* [*simp*]: $f⇂_{ran(f)}$ = *f*
proof
fix *x*
show ($f⇂_{ran(f)}$) *x* = *f x*
proof (*cases f*(*x*))
case *None*
then show *?thesis* **by** (*simp add*: *ran-restrict-map-def ran-def*)
next
case (*Some a*)
then show *?thesis* **by** (*auto simp add*: *ran-restrict-map-def ran-def*)
qed
qed

lemma *ran-ran-restrict* [*simp*]: *ran*($f⇂_B$) = *ran*(*f*) ∩ *B*

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

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

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

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

lemma *map-dres-rres-commute*: $f|_B\ |`\ A = (f\ |`\ A)|_B$
by (*auto simp add*: *restrict-map-def ran-restrict-map-def*)

lemma *ran-restrict-map-twice* [*simp*]: $(f|_A)|_B = f|_{(A \cap B)}$
proof
fix *x*
show $((f|_A)|_B)\ x = (f|_{(A \cap B)})\ x$
proof (*cases f x*)
case *None*
then show *?thesis* **by** (*simp add*: *ran-restrict-map-def*)
next
case (*Some a*)
then show *?thesis* **by** (*simp add*: *ran-restrict-map-def fun-eq-iff option.case-eq-if*)
qed
qed

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*]: $[\![\ x \notin dom\ g;\ \ x \in dom\ f\ ]\!] \Longrightarrow (f\ ++\ g)\ x = f\ x$
by (*auto simp add*:*map-add-def dom-def*)

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

13.9 Map Inverse and Identity

definition *map-inv* :: $('a \rightharpoonup 'b) \Rightarrow ('b \rightharpoonup 'a)$ **where**
map-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-id-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-id-on xs x* $=$ *Some x*

by (*simp add*:*map-id-on-def*)

lemma *map-id-on-out* [*simp*]:
x ∉ *xs* ⟹ *map-id-on xs x* = *None*
by (*simp add*:*map-id-on-def*)

lemma *map-id-dom* [*simp*]: *dom* (*map-id-on xs*) = *xs*
by (*simp add*:*dom-def map-id-on-def*)

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

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

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

lemma *restrict-map-inj-on*:
inj-on f (*dom f*) ⟹ *inj-on* (*f* |‘ *A*) (*dom f* ∩ *A*)
by (*auto simp add*:*inj-on-def*)

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

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

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

lemma *map-inv-f-f* [*simp*]:
⟦ *inj-on f* (*dom f*); *f x* = *Some y* ⟧ ⟹ *map-inv f y* = *Some x*
apply (*simp add*: *map-inv-def*, *safe*)
apply (*rule some-equality*)
apply (*auto simp add*:*inj-on-def dom-def ran-def*)
done

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

lemma *ran-map-inv* [*simp*]:
inj-on f (*dom f*) ⟹ *ran* (*map-inv f*) = *dom f*
apply (*simp add*:*map-inv-def ran-def*, *safe*)
apply (*metis* (*mono-tags*, *lifting*) *verit-sko-ex*′)
apply (*metis* (*mono-tags*, *lifting*) *domI domIff map-inv-def map-inv-f-f option.inject*)

done

lemma *dom-image-ran*: *f ‘ dom f = Some ‘ ran f*
by (*auto simp add*:*dom-def ran-def image-def*)

lemma *inj-map-inv* [*intro*]:
inj-on f (*dom f*) ⟹ *inj-on* (*map-inv f*) (*ran f*)
apply (*simp add*:*map-inv-def inj-on-def dom-def ran-def*, *safe*)
apply (*metis* (*mono-tags*, *lifting*) *option.sel someI-ex*)
done

lemma *inj-map-bij*: *inj-on f* (*dom f*) ⟹ *bij-betw f* (*dom f*) (*Some ‘ ran f*)
by (*auto simp add*:*inj-on-def dom-def ran-def image-def bij-betw-def*)

lemma *map-inv-map-inv* [*simp*]:
assumes *inj-on f* (*dom f*)
shows *map-inv* (*map-inv f*) = *f*
proof −

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

thus *?thesis*
by (*metis* (*no-types*, *lifting*) *ext assms domIff dom-map-inv map-inv-f-f option.collapse*
ran-map-inv)
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*)
proof (*rule inj-onI*)
fix *x y*
assume *x*:*x* ∈ *dom f* ∪ *ran f* **and** *y*:*y* ∈ *dom f* ∪ *ran f*
and *f*:(*map-inv f* ++ *f*) *x* = (*map-inv f* ++ *f*) *y*

show *x* = *y*
proof (*cases x* ∈ *dom f*)
case *True*
then show *?thesis*
by (*metis assms*(*1*,*2*) *disjoint-iff-not-equal domD dom-left-map-add f inj-on-def*
map-add-dom-app-simps(*3*) *ranI ran-map-inv*)
next
case *False*
then show *?thesis*
by (*metis* (*full-types*) *UnE assms*(*1*,*2*) *disjoint-iff domIff dom-left-map-add dom-map-inv*
f inj-map-inv inj-on-def map-add-dom-app-simps(*3*) *ran-map-inv ran-restrict-alt-def*
ran-restrict-ran x)

```
  qed
qed

lemma inj-completed-map [intro]:
  assumes dom f = ran f inj-on f (dom f)
  shows inj (Some ++ f)
proof (rule injI)
  fix x y
  assume f:(Some ++ f) x = (Some ++ f) y
  have bb: bij-betw f (dom f) (Some ` ran f)
    using assms(2) inj-map-bij by blast
  thus x = y
  proof (cases x ∈ dom f)
    case True
    then show ?thesis
      by (metis assms(1,2) f inj-on-contraD map-add-dom-app-simps(1,3) ranI)
  next
    case False
    then show ?thesis
    by (metis assms(1) dom-left-map-add f map-add-dom-app-simps(3) option.inject
        ranI)
  qed
qed

lemma bij-completed-map [intro]:
  fixes f :: 'a ⇀ 'a
  assumes dom f = ran f inj-on f (dom f)
  shows bij-betw (Some ++ f) UNIV (range Some)
proof −
  have range (Some ++ f) = range Some
  proof (rule set-eqI, rule iffI)
    fix x
    assume x ∈ range (Some ++ f)
    thus x ∈ range Some
      using image-iff by fastforce
  next
    fix x :: 'a option
    assume x ∈ range Some
    thus x ∈ range (Some ++ f)
      by (metis assms(1) dom-image-ran[of f] image-iff[of x f dom f] image-iff[of
Some - Some dom f] image-iff[of x Some UNIV]
        map-add-dom-app-simps(1)[of - f Some] map-add-dom-app-simps(3)[of - f
Some] rangeI[of Some ++ f])
  qed
  thus ?thesis
    by (metis assms(1,2) inj-completed-map inj-on-imp-bij-betw)
qed
```

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* (*opaque-lifting*, *no-types*) *comp-inj-on-iff f-the-inv-into-f inj-on-inverseI option.sel*)
apply (*metis* (*opaque-lifting*, *no-types*) *image-iff option.sel*)
apply (*metis Option.these-def Some-image-these-eq image-image these-image-Some-eq*)
done

lemma *ran-map-add* [*simp*]:
m‘(*dom m* ∩ *dom n*) = *n*‘(*dom m* ∩ *dom n*) ⟹
ran(*m*++*n*) = *ran n* ∪ *ran m*
apply (*simp add*:*ran-def*, *safe*, *simp-all*)
apply (*metis*)
apply (*metis domI map-add-dom-app-simps*(*1*))
apply (*metis* (*no-types*, *opaque-lifting*) *IntI domI dom-left-map-add image-iff map-add-dom-app-simps*(*3*))
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 (*simp add*:*inj-on-def*, *safe*, *simp-all*)
apply (*metis* (*full-types*) *disjoint-iff-not-equal domI dom-left-map-add map-add-dom-app-simps*(*3*) *ranI*)
apply (*metis disjoint-iff-not-equal domI map-add-SomeD ranI*)
apply (*metis disjoint-iff-not-equal domIff map-add-Some-iff ranI*)
apply (*metis domI*)
done

lemma *map-inv-add′*:
assumes *inj-on f* (*dom f*) *inj-on g* (*dom g*)
dom f ∩ *dom g* = {} *ran f* ∩ *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* ∈ *ran g* ⟹ *map-inv* (*f* ++ *g*) *x* = (*map-inv f* ++ *map-inv g*) *x*
proof −
assume *ran*:*x* ∈ *ran g*
then obtain *y* **where** *dom*:*g y* = *Some x y* ∈ *dom g*

by (*auto simp add*:*ran-def*)

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

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

moreover have $[\![x \notin ran\ g;\ x \in ran\ f]\!] \Longrightarrow map\text{-}inv\ (f ++ g)\ x = (map\text{-}inv\ f ++ map\text{-}inv\ g)\ x$
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* $(f ++ g)\ x = (map\text{-}inv\ f ++ map\text{-}inv\ g)\ x$
by *simp*
qed

moreover from *assms minj* **have** $[\![x \notin ran\ g;\ x \notin ran\ f]\!] \Longrightarrow map\text{-}inv\ (f ++ g)\ x = (map\text{-}inv\ f ++ map\text{-}inv\ g)\ x$
apply (*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* $(f ++ g)\ x = (map\text{-}inv\ f ++ map\text{-}inv\ g)\ x$
by *blast*
qed

lemma *map-inv-dom-res*:
assumes *inj-on f* (*dom f*)
shows *map-inv* $(f\ |`\ A) = (map\text{-}inv\ f)\ \restriction_A$
using *assms*
apply (*simp add*: *map-inv-def restrict-map-def ran-restrict-map-def dom-def ran-def fun-eq-iff inj-on-def*)
apply (*safe intro*!: *some-equality*)
apply (*metis* (*mono-tags*, *lifting*) *someI-ex*)+
done

lemma *map-inv-ran-res*:
assumes *inj-on f* (*dom f*)
shows *map-inv* $(f\ \restriction_A) = (map\text{-}inv\ f)\ |`\ A$
using *assms someI-ex* **by** (*force intro*!: *some-equality simp add*: *map-inv-def*

restrict-map-def ran-restrict-map-def dom-def ran-def fun-eq-iff inj-on-def)

lemma *map-update-as-add*: $f(x \mapsto y) = f ++ [x \mapsto y]$
by (*auto simp add*: *map-add-def*)

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*: $Some ++ f = map\text{-}id\text{-}on\ (-\ dom\ f) ++ f$
proof
fix *x*
show $(Some ++ f)\ x = (map\text{-}id\text{-}on\ (-\ dom\ f) ++ f)\ x$
proof (*cases* $x \in dom\ f$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis* **by** *simp*
qed
qed

lemma *distinct-map-dom*:
$x \notin set\ xs \Longrightarrow x \notin dom\ [xs\ [\mapsto]\ ys]$
by (*simp add*:*dom-def*)

lemma *distinct-map-ran*:
$[\![\ distinct\ xs;\ \ y \notin set\ ys;\ \ length\ xs = length\ ys\ ]\!] \Longrightarrow$
$y \notin ran\ ([xs\ [\mapsto]\ ys])$
apply (*simp add*:*map-upds-def*)
apply (*subgoal-tac distinct* (*map fst* (*rev* (*zip xs ys*))))
apply (*simp add*:*ran-distinct*)
apply (*metis* (*opaque-lifting*, *no-types*) *image-iff set-zip-rightD surjective-pairing*)
apply (*simp add*:*zip-rev*[*THEN sym*])
done

lemma *maplets-lookup* [*dest*]:
$[\![\ length\ xs = length\ ys;\ \ distinct\ xs;\ \ \forall\ y.\ [xs\ [\mapsto]\ ys]\ x = Some\ y\ ]\!] \Longrightarrow y \in set\ ys$
using *ranI* **by** *fastforce*

lemma *maplets-distinct-inj* [*intro*]:
$[\![\ length\ xs = length\ ys;\ \ distinct\ xs;\ \ distinct\ ys;\ \ set\ xs \cap set\ ys = \{\}\ ]\!] \Longrightarrow$
$inj\text{-}on\ [xs\ [\mapsto]\ ys]\ (set\ xs)$
apply (*induct rule*:*list-induct2*)
apply (*simp-all*)
apply (*rule conjI*)
apply (*rule inj-onI*)
apply (*metis fun-upd-def inj-on-contraD*)

apply (*metis image-iff ranI ran-maplets*)
done

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

lemma *map-inv-add*:
assumes *inj-on f* (*dom f*) *inj-on g* (*dom g*)
$ran\ f \cap ran\ g = \{\}$
shows *map-inv* $(f +\!\!+ g) = map\text{-}inv\ f\restriction_{(-\ dom\ g)} +\!\!+ map\text{-}inv\ g$
proof −
have *map-inv* $(f +\!\!+ g) = map\text{-}inv\ (f\ |`\ (-\ dom(g))\ +\!\!+ g)$
by (*metis map-add-restrict*)
also have ... = *map-inv* $(f\ |`\ (-\ dom\ g)) +\!\!+ map\text{-}inv\ g$
by (*rule map-inv-add′*, *simp-all add*: *assms restrict-map-inj-on*)
(*metis assms*(*3*) *disjoint-iff ranI ran-restrictD*)
also have ... = *map-inv* $f\restriction_{(-\ dom\ g)} +\!\!+ map\text{-}inv\ g$
by (*simp add*: *map-inv-dom-res assms*)
finally show *?thesis* **.**
qed

lemma *map-inv-upd*:
assumes *inj-on f* (*dom f*) *inj-on g* (*dom g*) $v \notin ran\ f$
shows *map-inv* $(f(k \mapsto v)) = (map\text{-}inv\ (f\ |`\ (-\ \{k\})))(v \mapsto k)$
proof −
have *map-inv* $(f(k \mapsto v)) = map\text{-}inv\ (f +\!\!+ [k \mapsto v])$
by (*auto*)
also have ... = *map-inv* $f\restriction_{(-\ dom\ [k \mapsto v])} +\!\!+ map\text{-}inv\ [k \mapsto v]$
by (*rule map-inv-add*, *simp-all add*: *assms*)
also have ... = $(map\text{-}inv\ f\restriction_{(-\ \{k\})})(v \mapsto k)$
by (*simp*)
also have ... = $(map\text{-}inv\ (f\ |`\ (-\ \{k\})))(v \mapsto k)$
by (*simp add*: *assms*(*1*) *map-inv-dom-res*)
finally show *?thesis* **.**
qed

lemma *map-inv-maplets* [*simp*]:
⟦ *length xs* = *length ys*; *distinct xs*; *distinct ys*; *set xs* ∩ *set ys* = {} ⟧ ⟹
map-inv [*xs* [↦] *ys*] = [*ys* [↦] *xs*]
proof (*induct rule*:*list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs y ys*)
have *map-inv* ([*xs* [↦] *ys*] ++ [*x* ↦ *y*]) = *map-inv* [*xs* [↦] *ys*] ++ *map-inv* [*x* ↦ *y*]
proof (*rule map-inv-add′*)
from *Cons* **show** *inj-on* [*xs* [↦] *ys*] (*dom* [*xs* [↦] *ys*]) **by** *auto*
from *Cons* **show** *inj-on* [*x* ↦ *y*] (*dom* [*x* ↦ *y*]) **by** *auto*

from *Cons* **show** *dom* [*xs* [↦] *ys*] ∩ *dom* [*x* ↦ *y*] = {} **by** *auto*
from *Cons* **show** *ran* [*xs* [↦] *ys*] ∩ *ran* [*x* ↦ *y*] = {} **by** *auto*
qed
with *Cons* **show** *?case*
by (*metis disjoint-iff distinct.simps*(2) *list.set-intros*(2) *map-inv-maplet map-update-as-add map-upds-Cons map-upds-twist*)
qed

lemma *maplets-lookup-nth* [*simp*]:
⟦ *length xs* = *length ys*; *distinct xs*; *i* < *length ys* ⟧ ⟹
[*xs* [↦] *ys*] (*xs* ! *i*) = *Some* (*ys* ! *i*)
apply (*induct arbitrary*: *i rule*:*list-induct2*)
apply *simp*
using *less-Suc-eq-0-disj* **apply** *auto*
done

theorem *inv-map-inv*:
assumes *inj-on f* (*dom f*) *ran f* = *dom f*
shows *inv* (*the* ∘ (*Some* ++ *f*)) = *the* ∘ *map-inv* (*Some* ++ *f*)
proof
fix *x*
show (*inv* (*the* ∘ (*Some* ++ *f*))) *x* = (*the* ∘ *map-inv* (*Some* ++ *f*)) *x*
proof (*cases x* ∈ *ran f*)
case *True*
then obtain *y* **where** *y*:*f y* = *Some x*
by (*metis dom-image-ran image-iff*)
with *assms* **show** *?thesis*
apply (*simp add*:*map-add-Some map-inv-add′ inv-def*)
apply (*rule some-equality*)
apply *simp*
apply (*metis* (*full-types*) *Compl-iff domIff inj-on-def map-add-Some map-add-dom-app-simps*(2,3) *map-id-dom option.exhaust-sel ranI*)
done
next
case *False*
then show *?thesis*
apply (*simp add*:*map-add-Some map-inv-add′ inv-def*)
apply (*rule some-equality*)
apply (*simp add*: *assms*(1,2) *map-inv-add′*)
apply (*metis* (*no-types*, *opaque-lifting*) *Un-UNIV-right assms*(1,2) *dom-map-add inj-completed-map map-add-None map-add-Some map-id-dom map-id-on-UNIV map-inv-f-f option.exhaust-sel*
option.sel)
done
qed
qed

lemma *map-comp-dom*: *dom* (*g* ∘$_m$ *f*) ⊆ *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*)

lemma *map-graph-map-inv*: *inj-on f* (*dom f*) $\Longrightarrow$ *map-graph* (*map-inv f*) = (*map-graph f*)$^{-1}$
  by (*simp add*: *map-graph-def*, *safe*, *simp-all*, *metis dom-map-inv inj-map-inv map-inv-f-f map-inv-map-inv*)

13.10 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*: $f \parallel_m g \Longrightarrow g \parallel_m f$
by (*simp add*: *comp-map-def*)

definition *merge* :: $('a \rightharpoonup 'b)\ set \Rightarrow 'a \rightharpoonup 'b$ **where**
merge fs =
(λ *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*
unfolding *merge-def*
by (*auto simp add*: *fun-eq-iff domIff*)

13.11 Conversion between lists and maps

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

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

lemma *dom-map-of-list* [*simp*]: *dom* (*map-of-list xs*) = {*0*..<*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* ≠ *Map.empty*
by (*metis ran-empty ran-map-of-list set-empty*)
ultimately show *?thesis*
by (*auto intro*!:*nth-equalityI simp add*: *list-of-map-def map-of-list-def fun-eq-iff*)
qed

13.12 Map Comprehension

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

syntax
-Mapcompr :: *'a* ⇒ *'b* ⇒ *idts* ⇒ *bool* ⇒ *'a* ⇀ *'b* (*(1[- ↦ - |/-./ -])*)

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

lemma *map-compr-eta*:
[*x* ↦ *y* | *x y*. (*x*, *y*) ∈*m f*] = *f*
apply (*rule ext*)
apply (*simp add*: *graph-map-def*, *safe*, *simp-all*)
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* ↦ *F x y* | *x y*. (*x*, *y*) ∈*m f*] = (λ *x*. *do* { *y* ← *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* ↦ *f x* | *x*. *P x*] = {*x*. *P x*}
by (*force simp add*: *graph-map-dom image-Collect*)

lemma *map-compr-ran-simple* [*simp*]:
ran [*x* ↦ *f x* | *x*. *P x*] = {*f x* | *x*. *P x*}
apply (*simp add*: *graph-map-def ran-def*, *safe*, *simp-all*)
apply *blast*
apply (*metis* (*mono-tags*, *lifting*) *fst-eqD image-eqI mem-Collect-eq someI*)
done

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

13.13 Sorted lists from maps

definition *sorted-list-of-map* :: (*'a*::*linorder* ⇀ *'b*) ⇒ (*'a* × *'b*) *list* **where**

sorted-list-of-map f = map (λ k. (k, the (f k))) (sorted-list-of-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*: *assms sorted-list-of-map-def map-of-map-keys*)
qed
qed

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

13.14 Extra map lemmas

lemma *map-eqI*:
⟦ *dom f = dom g*; ∀ *x*∈*dom*(*f*). *the*(*f x*) = *the*(*g x*) ⟧ ⟹ *f = g*
by (*metis domIff map-le-antisym map-le-def option.expand*)

lemma *map-restrict-dom* [*simp*]: *f* |‘ *dom f = f*
by (*simp add*: *map-eqI*)

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*) ∩ *A* = {} ⟹ *f* |‘ (− *A*) = *f*
by (*simp add*: *restrict-map-def*, *rule ext*, *metis IntI domIff empty-iff*)

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*: $f \mid` (- \ dom \ h) = (f \ ++ \ h) \mid` (- \ dom \ h)$
by (*simp add*: *map-plus-restrict-dist*)
have $h \mid` (- \ dom \ h) = Map.empty$
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*:
$dom(f) = A \cup B \Longrightarrow (f \mid` A) \ ++ \ (f \mid` B) = f$
by (*rule ext*, *auto simp add*: *map-add-def restrict-map-def option.case-eq-if*)

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

lemma *map-add-cancel*:
$f \subseteq_m g \Longrightarrow f \ ++ \ (g \ -- \ f) = g$
by (*simp add*: *map-le-def map-add-def map-minus-def fun-eq-iff option.case-eq-if*)
(*metis domIff*)

lemma *map-le-iff-add*: $f \subseteq_m g \longleftrightarrow (\exists \ h. \ dom(f) \cap dom(h) = \{\} \wedge f \ ++ \ h = g)$
proof
assume $f \subseteq_m g$
hence $dom \ f \cap dom \ (g \ -- \ f) = \{\} \wedge f \ ++ \ (g \ -- \ f) = g$
by (*metis* (*no-types*, *lifting*) *Int-emptyI domIff map-add-cancel map-le-def map-minus-def*)
thus $\exists h. \ dom \ f \cap dom \ h = \{\} \wedge f \ ++ \ h = g$ **by** *blast*
next
assume $\exists h. \ dom \ f \cap dom \ h = \{\} \wedge f \ ++ \ h = g$
thus $f \subseteq_m g$
by (*auto simp add*: *map-add-comm*)
qed

lemma *map-add-comm-weak*: $(\forall \ k \in dom \ m1 \cap dom \ m2. \ m1(k) = m2(k)) \Longrightarrow m1 \ ++ \ m2 = m2 \ ++ \ m1$
by (*simp add*: *map-add-def option.case-eq-if fun-eq-iff*)
(*metis IntI domI*)

lemma *map-add-comm-weak'*: $Q \mid` dom \ P = P \mid` dom \ Q \Longrightarrow P \ ++ \ Q = Q \ ++ \ P$
by (*metis IntD1 IntD2 map-add-comm-weak restrict-in*)

lemma *map-compat-add*: $Q \mid` dom \ P = P \mid` dom \ Q \Longrightarrow R \mid` (dom \ Q \cup dom \ P) = (P \ ++ \ Q) \mid` dom \ R \Longrightarrow R \mid` dom \ P = P \mid` dom \ R$
by (*metis Int-commute Map.restrict-restrict Un-Int-eq*(*2*) *map-add-comm-weak' map-le-iff-map-add-commute map-le-via-restrict*)

abbreviation *rel-map R* ≡ *rel-fun* (=) (*rel-option R*)

lemma *rel-map-iff*:
rel-map R f g ⟷ (*dom*(*f*) = *dom*(*g*) ∧ (∀ *x*∈*dom*(*f*). *R* (*the* (*f x*)) (*the* (*g x*))))
apply (*simp add*: *rel-fun-def*, *safe*)
apply (*metis not-None-eq option.rel-distinct*(*2*))
apply (*metis not-None-eq option.rel-distinct*(*1*))
apply (*metis option.rel-sel option.sel option.simps*(*3*))
apply (*metis domIff option.rel-sel*)
done

end

14 Partial Functions

theory *Partial-Fun*
imports *Optics.Optics Map-Extra HOL−Library.Mapping*
begin

no-notation *Stream.stream.SCons* (**infixr** ‹##› *65*)

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.

14.1 Partial function type and operations

typedef ('*a*, '*b*) *pfun* = *UNIV* :: ('*a* ⇀ '*b*) *set*
morphisms *pfun-lookup pfun-of-map* **..**

type-notation *pfun* (**infixr** ↦ *1*)

setup-lifting *type-definition-pfun*

lemma *pfun-lookup-map* [*simp*]: *pfun-lookup* (*pfun-of-map f*) = *f*
by (*simp add*: *pfun-of-map-inverse*)

lift-bnf ('*k*, *pran*: '*v*) *pfun* [*wits*: *Map.empty* :: '*k* ⇒ '*v option*] **for** *map*: *map-pfun rel*: *relt-pfun*
by *auto*

declare *pfun.map-transfer* [*transfer-rule*]

instantiation *pfun* :: (*type*, *type*) *equal*
begin

definition *HOL.equal m1 m2* ⟷ (∀ *k*. *pfun-lookup m1 k* = *pfun-lookup m2 k*)

instance
by (*intro-classes*, *simp add*: *equal-pfun-def*, *transfer*, *auto*)

end

lift-definition *pfun-app* :: ('*a*, '*b*) *pfun* ⇒ '*a* ⇒ '*b* (-'(-')$_p$ [*999*,*0*] *999*) **is**
λ *f x*. *if* (*x* ∈ *dom f*) *then the* (*f x*) *else undefined* **.**

lift-definition *pfun-upd* :: ('*a*, '*b*) *pfun* ⇒ '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *pfun*
is λ *f k v*. *f*(*k* := *Some v*) **.**

lift-definition *pdom* :: ('*a*, '*b*) *pfun* ⇒ '*a set* **is** *dom* **.**

lemma *pran-rep-eq* [*transfer-rule*]: *pran f* = *ran* (*pfun-lookup f*)
by (*transfer*, *auto simp add*: *ran-def*)

lift-definition *pfun-comp* :: ('*b*, '*c*) *pfun* ⇒ ('*a*, '*b*) *pfun* ⇒ ('*a*, '*c*) *pfun* (**infixl** ∘$_p$ *55*) **is**
λ *f g*. *f* ∘$_m$ *g* **.**

lift-definition *map-pfun'* :: ('*c* ⇒ '*a*) ⇒ ('*b* ⇒ '*d*) ⇒ ('*a*, '*b*) *pfun* ⇒ ('*c*, '*d*) *pfun*
is λ*f g m*. (*map-option g* ∘ *m* ∘ *f*) **parametric** *map-parametric* **.**

functor *map-pfun'*
by (*transfer*, *auto simp add*: *fun-eq-iff option.map-comp option.map-id*)+

lift-definition *pfun-member* :: '*a* × '*b* ⇒ ('*a*, '*b*) *pfun* ⇒ *bool* (**infix** ∈$_p$ *50*) **is**
(∈$_m$) **.**

lift-definition *pfun-inj* :: ('*a*, '*b*) *pfun* ⇒ *bool* **is** λ *f*. *inj-on f* (*dom f*) **.**

lift-definition *pfun-inv* :: ('*a*, '*b*) *pfun* ⇒ ('*b*, '*a*) *pfun* **is** *map-inv* **.**

lift-definition *pId-on* :: '*a set* ⇒ ('*a*, '*a*) *pfun* **is** λ *A x*. *if* (*x* ∈ *A*) *then Some x*
else None **.**

abbreviation *pId* :: ('*a*, '*a*) *pfun* **where**
pId ≡ *pId-on UNIV*

lift-definition *pdom-res* :: '*a set* ⇒ ('*a*, '*b*) *pfun* ⇒ ('*a*, '*b*) *pfun* (**infixr** ◁$_p$ *85*)
is λ *A f*. *restrict-map f A* **.**

abbreviation *pdom-nres* (**infixr** −◁$_p$ *85*) **where** *pdom-nres A P* ≡ (− *A*) ◁$_p$ *P*

lift-definition *pran-res* :: ('*a*, '*b*) *pfun* ⇒ '*b set* ⇒ ('*a*, '*b*) *pfun* (**infixl** ▷$_p$ *86*)
is *ran-restrict-map* **.**

abbreviation *pran-nres* (**infixr** ▷$_p$− *66*) **where** *pran-nres P A* ≡ *P* ▷$_p$ (− *A*)

definition *pfun-image* :: $'a \rightharpoonup 'b \Rightarrow 'a\ set \Rightarrow 'b\ set$ **where**
[*simp*]: $pfun\text{-}image\ f\ A = pran\ (A \lhd_p f)$

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** $graph\text{-}map \circ mk\text{-}functional$
.

definition *pfun-pfun* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \rightharpoonup 'b)\ set$ **where**
$pfun\text{-}pfun\ A\ B = \{f :: 'a \rightharpoonup 'b.\ pdom(f) \subseteq A \wedge pran(f) \subseteq B\}$

definition *pfun-tfun* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \rightharpoonup 'b)\ set$ **where**
$pfun\text{-}tfun\ A\ B = \{f \in pfun\text{-}pfun\ A\ B.\ pdom(f) = UNIV\}$

definition *pfun-ffun* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \rightharpoonup 'b)\ set$ **where**
$pfun\text{-}ffun\ A\ B = \{f \in pfun\text{-}pfun\ A\ B.\ finite(pdom(f))\}$

definition *pfun-pinj* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \rightharpoonup 'b)\ set$ **where**
$pfun\text{-}pinj\ A\ B = \{f \in pfun\text{-}pfun\ A\ B.\ pfun\text{-}inj\ f\}$

definition *pfun-psurj* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \rightharpoonup 'b)\ set$ **where**
$pfun\text{-}psurj\ A\ B = \{f \in pfun\text{-}pfun\ A\ B.\ pran(f) = UNIV\}$

definition $pfun\text{-}finj\ A\ B = pfun\text{-}ffun\ A\ B \cap pfun\text{-}pinj\ A\ B$
definition $pfun\text{-}tinj\ A\ B = pfun\text{-}tfun\ A\ B \cap pfun\text{-}pinj\ A\ B$
definition $pfun\text{-}tsurj\ A\ B = pfun\text{-}tfun\ A\ B \cap pfun\text{-}psurj\ A\ B$
definition $pfun\text{-}bij\ A\ B = pfun\text{-}tfun\ A\ B \cap pfun\text{-}pinj\ A\ B \cap pfun\text{-}psurj\ A\ B$

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 *pfuse* :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'c) \Rightarrow ('a \rightharpoonup 'b \times 'c)$
where $pfuse\ f\ g = pfun\text{-}entries\ (pdom(f) \cap pdom(g))\ (\lambda\ x.\ (pfun\text{-}app\ f\ x, pfun\text{-}app\ g\ x))$

lift-definition *ptabulate* :: $'a\ list \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a, 'b)\ pfun$
is $\lambda ks\ f.\ (map\text{-}of\ (List.map\ (\lambda k.\ (k, f\ k))\ ks))$ **.**

lift-definition *pcombine* ::
$('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)\ pfun \Rightarrow ('a, 'b)\ pfun \Rightarrow ('a, 'b)\ pfun$
is $\lambda f\ m1\ m2\ x.\ combine\text{-}options\ f\ (m1\ x)\ (m2\ x)$ **.**

abbreviation $fun\text{-}pfun \equiv pfun\text{-}entries\ UNIV$

definition *pfun-disjoint* :: $'a \rightharpoonup 'b\ set \Rightarrow bool$ **where**
$pfun\text{-}disjoint\ S = (\forall\ i \in pdom\ S.\ \forall\ j \in pdom\ S.\ i \neq j \longrightarrow pfun\text{-}app\ S\ i \cap pfun\text{-}app\ S\ j = \{\})$

definition *pfun-partitions* :: $'a \nrightarrow 'b\ set \Rightarrow 'b\ set \Rightarrow bool$ **where**
pfun-partitions S T = (*pfun-disjoint S* $\wedge$ $\bigcup$ (*pran S*) = *T*)

no-notation *disj* (**infixr** | *30*)

definition *pabs* :: $'a\ set \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \nrightarrow 'b$ **where**
pabs A P f = (*A* $\cap$ *Collect P*) $\lhd_p$ *fun-pfun f*

definition *pcard* :: ('*a*, '*b*) *pfun* $\Rightarrow$ *nat*
where *pcard f* = *card* (*pdom f*)

unbundle *lattice-syntax*

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

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

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

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

instantiation *pfun* :: (*type*, *type*) *inf*
begin
lift-definition *inf-pfun* :: ('*a*, '*b*) *pfun* $\Rightarrow$ ('*a*, '*b*) *pfun* $\Rightarrow$ ('*a*, '*b*) *pfun* **is**
λ *f g x. if* (*x* $\in$ *dom*(*f*) $\cap$ *dom*(*g*) $\wedge$ *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* ⇒ ('*a*, '*b*) *pfun* ⇒ *bool* **is**
λ *f g*. *f* ⊆$_m$ *g* .
lift-definition *less-pfun* :: ('*a*, '*b*) *pfun* ⇒ ('*a*, '*b*) *pfun* ⇒ *bool* **is**
λ *f g*. *f* ⊆$_m$ *g* ∧ *f* ≠ *g* .
instance
by (*intro-classes*, (*transfer*, *auto intro*: *map-le-trans simp add*: *map-le-antisym*)+)
end

abbreviation *pfun-subset* :: ('*a*, '*b*) *pfun* ⇒ ('*a*, '*b*) *pfun* ⇒ *bool* (**infix** ⊂$_p$ *50*)
where *pfun-subset* ≡ *less*

abbreviation *pfun-subset-eq* :: ('*a*, '*b*) *pfun* ⇒ ('*a*, '*b*) *pfun* ⇒ *bool* (**infix** ⊆$_p$ *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*]:
{}$_p$ ⊆$_p$ *f*
by (*transfer*, *auto*)

syntax
-PfunUpd :: [('*a*, '*b*) *pfun*, *maplets*] => ('*a*, '*b*) *pfun* (-'(-')$_p$ [*900*,*0*]*900*)
-Pfun :: *maplets* => ('*a*, '*b*) *pfun* ((*1*{-}$_p$))
-pabs :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ⇒ *logic* (λ - ∈ - | - • - [*0*, *0*, *0*, *10*] *10*)
-pabs-mem :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ⇒ *logic* (λ - ∈ - • - [*0*, *0*, *10*] *10*)
-pabs-pred :: *pttrn* ⇒ *logic* ⇒ *logic* ⇒ *logic* ⇒ *logic* (λ - | - • - [*0*, *0*, *10*] *10*)
-pabs-tot :: *pttrn* ⇒ *logic* ⇒ *logic* (λ - • - [*0*, *10*] *10*)

translations
-PfunUpd m (*-Maplets xy ms*) == *-PfunUpd* (*-PfunUpd m xy*) *ms*
-PfunUpd m (*-maplet x y*) == *CONST pfun-upd m x y*
-Pfun ms => *-PfunUpd* (*CONST pempty*) *ms*
-Pfun (*-Maplets ms1 ms2*) <= *-PfunUpd* (*-Pfun ms1*) *ms2*
-Pfun ms <= *-PfunUpd* (*CONST pempty*) *ms*
-pabs x A P f => *CONST pabs A* (λ *x*. *P*) (λ *x*. *f*)
-pabs x A P f <= *CONST pabs A* (λ *y*. *P*) (λ *x*. *f*)
-pabs x A P (*f x*) <= *CONST pabs A* (λ *x*. *P*) *f*
-pabs-mem x A f == *-pabs x A* (*CONST True*) *f*
-pabs-pred x P f == *-pabs x* (*CONST UNIV*) *P f*
-pabs-tot x f == *-pabs-pred x* (*CONST True*) *f*
-pabs-tot x f <= *-pabs-mem x* (*CONST UNIV*) *f*

14.2 Algebraic laws

lemma *pfun-comp-assoc*: *f* ∘$_p$ (*g* ∘$_p$ *h*) = (*f* ∘$_p$ *g*) ∘$_p$ *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-comp-left-zero* [*simp*]: $\{\}_p \circ_p f = \{\}_p$
by (*transfer*, *auto*)

lemma *pfun-comp-right-zero* [*simp*]: $f \circ_p \{\}_p = \{\}_p$
by (*transfer*, *auto*)

lemma *pfun-override-dist-comp*:
$(f \oplus g) \circ_p h = (f \circ_p h) \oplus (g \circ_p h)$
apply (*transfer*)
apply (*rule ext*)
apply (*simp add*: *map-add-def*)
apply (*metis* (*no-types*, *lifting*) *bind.bind-lunit bind-eq-None-conv map-comp-def option.case-eq-if option.collapse*)
done

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

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

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

instantiation *pfun* :: (*type*, *type*) *override*
begin
definition *compatible-pfun* :: $'a \rightharpoonup 'b \Rightarrow 'a \rightharpoonup 'b \Rightarrow bool$ **where**
compatible-pfun R S = $((pdom\ R) \lhd_p S = (pdom\ S) \lhd_p R)$

lemma *pfun-compat-add*: $(P :: 'a \rightharpoonup 'b)\ \#\#\ Q \Longrightarrow P \oplus Q\ \#\#\ R \Longrightarrow P\ \#\#\ R$
apply (*simp add*: *compatible-pfun-def oplus-pfun-def*)
apply (*transfer*)
using *map-compat-add* **apply** *auto*
done

lemma *pfun-compat-addI*: $[\![(P :: 'a \rightharpoonup 'b)\ \#\#\ Q;\ \ P\ \#\#\ R;\ \ Q\ \#\#\ R]\!] \Longrightarrow P$

$\oplus$ Q ## R
apply (*simp add*: *compatible-pfun-def oplus-pfun-def*)
apply (*transfer*)
apply (*simp add*: *restrict-map-def fun-eq-iff dom-def map-add-def option.case-eq-if*)
apply *metis*
done

instance proof
fix P Q R :: $'a \rightharpoonup 'b$
show P ## $Q \Longrightarrow P \oplus Q$ ## $R \Longrightarrow P$ ## R
using *pfun-compat-add* **by** *blast*
show P ## $Q \Longrightarrow P$ ## $R \Longrightarrow Q$ ## $R \Longrightarrow P \oplus Q$ ## R
by (*simp add*: *pfun-compat-addI*)
qed (*simp-all add*: *compatible-pfun-def oplus-pfun-def*,
(*transfer*, *auto simp add*: *map-add-subsumed2 map-add-comm-weak'*)+)

end

lemma *pfun-indep-compat*: $pdom(f) \cap pdom(g) = \{\} \Longrightarrow f$ ## g
unfolding *compatible-pfun-def*
by (*transfer*, *auto simp add*: *restrict-map-def fun-eq-iff*)

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

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

lemma *pfun-override-fully*: $pdom\ f \subseteq pdom\ g \Longrightarrow f \oplus g = g$
by (*transfer*, *auto simp add*: *map-add-def option.case-eq-if fun-eq-iff*)

lemma *pfun-override-res*: $pdom\ g -\lhd_p f \oplus g = f \oplus g$
by (*transfer*, *auto simp add*: *map-add-restrict*[*THEN sym*])

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

lemma *pfun-override-minus*:
$f \subseteq_p g \Longrightarrow (g - f) \oplus f = g$
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-override*:
pdom(*f*) ∩ *pdom*(*g*) = {} ⟹ (*f* ⊕ *g*) − *g* = *f*
apply (*transfer*)
apply (*simp add*: *map-add-def map-minus-def option.case-eq-if fun-eq-iff*)
apply (*metis disjoint-iff domI domIff*)
done

lemma *pfun-override-pos*: *x* ⊕ *y* = {}$_p$ ⟹ *x* = {}$_p$
by (*transfer*, *simp*)

lemma *pfun-le-override*: *pdom x* ∩ *pdom y* = {} ⟹ *x* ≤ *x* ⊕ *y*
by (*transfer*, *auto simp add*: *map-le-iff-add*)

14.3 Membership, application, and update

lemma *pfun-ext*: ⟦ ⋀ *x y*. (*x*, *y*) ∈$_p$ *f* ⟷ (*x*, *y*) ∈$_p$ *g* ⟧ ⟹ *f* = *g*
by (*transfer*, *simp add*: *map-ext*)

lemma *pfun-member-alt-def*:
(*x*, *y*) ∈$_p$ *f* ⟷ (*x* ∈ *pdom f* ∧ *f*(*x*)$_p$ = *y*)
by (*transfer*, *auto simp add*: *map-member-alt-def map-apply-def*)

lemma *pfun-member-override*:
(*x*, *y*) ∈$_p$ *f* ⊕ *g* ⟷ ((*x* ∉ *pdom*(*g*) ∧ (*x*, *y*) ∈$_p$ *f*) ∨ (*x*, *y*) ∈$_p$ *g*)
by (*transfer*, *simp add*: *map-member-plus*)

lemma *pfun-member-minus*:
(*x*, *y*) ∈$_p$ *f* − *g* ⟷ (*x*, *y*) ∈$_p$ *f* ∧ (¬ (*x*, *y*) ∈$_p$ *g*)
by (*transfer*, *simp add*: *map-member-minus*)

lemma *pfun-app-in-ran* [*simp*]: *x* ∈ *pdom f* ⟹ *f*(*x*)$_p$ ∈ *pran f*
by (*transfer*, *auto*)

lemma *pfun-app-map* [*simp*]: (*pfun-of-map f*)(*x*)$_p$ = (*if* (*x* ∈ *dom*(*f*)) *then the* (*f x*) *else undefined*)
by (*transfer*, *simp*)

lemma *pfun-app-upd-1*: *x* = *y* ⟹ (*f*(*x* ↦ *v*)$_p$)(*y*)$_p$ = *v*
by (*transfer*, *simp*)

lemma *pfun-app-upd-2*: *x* ≠ *y* ⟹ (*f*(*x* ↦ *v*)$_p$)(*y*)$_p$ = *f*(*y*)$_p$
by (*transfer*, *simp*)

lemma *pfun-app-upd* [*simp*]: (*f*(*x* ↦ *e*)$_p$)(*y*)$_p$ = (*if* (*x* = *y*) *then e else f*(*y*)$_p$)
by (*metis pfun-app-upd-1 pfun-app-upd-2*)

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

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

lemma *pfun-app-add* [*simp*]: $x \in pdom(g) \Longrightarrow (f \oplus g)(x)_p = g(x)_p$
by (*transfer*, *auto*)

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

lemma *pfun-upd-add-left* [*simp*]: $x \notin pdom(g) \Longrightarrow f(x \mapsto v)_p \oplus g = (f \oplus g)(x \mapsto v)_p$
by (*transfer*, *safe*, *metis domD map-add-upd-left*)

lemma *pfun-app-add'* [*simp*]: $e \notin pdom\ g \Longrightarrow (f \oplus g)(e)_p = f(e)_p$
by (*transfer*, *auto*)

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-upd-as-ovrd*: $f(k \mapsto v)_p = f \oplus \{k \mapsto v\}_p$
by (*transfer*, *simp*)

lemma *pfun-ovrd-single-upd*: $x \in pdom(g) \Longrightarrow f \oplus (\{x\} \lhd_p g) = f(x \mapsto g(x)_p)_p$
by (*transfer*, *auto simp add*: *map-add-def restrict-map-def fun-eq-iff*)

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*]:
$[\![f \subseteq_p g;\ x \notin pdom(g)]\!] \Longrightarrow f \subseteq_p g(x \mapsto v)_p$
by (*transfer*, *auto simp add*: *map-le-def dom-def*)

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

lemma *psubseteq-pfun-upd3* [*intro*]:
$[\![f \subseteq_p g;\ g(x)_p = v]\!] \Longrightarrow f \subseteq_p g(x \mapsto v)_p$
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*)

lemma *pfun-eq-iff*: $f = g \longleftrightarrow (pdom(f) = pdom(g) \wedge (\forall\ x \in pdom(f).\ f(x)_p = g(x)_p))$
by (*safe*, *transfer*, *simp add*: *map-eq-iff*, *metis domD option.sel*)

lemma *pfun-leI*: $[\![pdom\ f \subseteq pdom\ g;\ \forall x{\in}pdom\ f.\ f(x)_p = g(x)_p]\!] \Longrightarrow f \subseteq_p g$
by (*transfer*, *simp add*: *map-le-def*, *safe*)
(*metis domD domI option.sel subsetD*)

lemma *pfun-le-iff*: $(f \subseteq_p g) = (pdom\ f \subseteq pdom\ g \wedge (\forall x{\in}pdom\ f.\ f(x)_p = g(x)_p))$
by (*metis pfun-app-add pfun-leI pfun-override-minus psubseteq-dom-subset*)

14.4 Map laws

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

lemma *map-pfun′-empty* [*simp*]: *map-pfun′ f g* $\{\}_p = \{\}_p$
unfolding *map-pfun′-def* **by** (*transfer*, *simp add*: *comp-def*)

lemma *map-pfun-upd* [*simp*]: *map-pfun f* $(g(x \mapsto v)_p) = (map\text{-}pfun\ f\ g)(x \mapsto f\ v)_p$
by (*simp add*: *map-pfun-def pfun-upd.rep-eq pfun-upd.abs-eq*)

lemma *map-pfun-apply* [*simp*]: $x \in pdom\ G \Longrightarrow (map\text{-}pfun\ F\ G)(x)_p = F(G(x)_p)$
unfolding *map-pfun-def* **by** (*auto simp add*: *pfun-app.rep-eq domD pdom.rep-eq*)

lemma *map-pfun-as-pabs*: *map-pfun f g* = (λ *x* ∈ *pdom*(*g*) · *f*(*g*(*x*)$_p$))
by (*simp add*: *pabs-def*, *transfer*, *auto simp add*: *fun-eq-iff restrict-map-def*)

lemma *map-pfun-ovrd* [*simp*]: *map-pfun f* (*g* ⊕ *h*) = (*map-pfun f g*) ⊕ (*map-pfun f h*)
by (*simp add*: *map-pfun-def*, *transfer*, *simp add*: *map-add-def fun-eq-iff*)
(*metis bind.bind-lunit comp-apply map-conv-bind-option option.case-eq-if*)

lemma *map-pfun-dres* [*simp*]: *map-pfun f* (*A* ◁$_p$ *g*) = *A* ◁$_p$ *map-pfun f g*
by (*simp add*: *map-pfun-def*, *transfer*, *auto simp add*: *restrict-map-def*)

14.5 Domain laws

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

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

lemma *pdom-plus* [*simp*]: *pdom* (*f* ⊕ *g*) = *pdom f* ∪ *pdom g*
by (*transfer*, *auto*)

lemma *pdom-minus* [*simp*]: *g* ≤ *f* ⟹ *pdom* (*f* − *g*) = *pdom f* − *pdom g*
apply (*transfer*, *simp add*: *map-minus-def*, *safe*)
apply (*meson option.distinct*(*1*))
apply (*metis domIff map-le-def option.simps*(*3*))
apply *metis*
done

lemma *pdom-inter*: *pdom* (*f* ∩$_p$ *g*) ⊆ *pdom f* ∩ *pdom g*
by (*transfer*, *auto simp add*: *dom-def*)

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

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

lemma *pdom-pdom-res* [*simp*]: *pdom* (*A* ◁$_p$ *f*) = *A* ∩ *pdom*(*f*)
by (*transfer*, *auto*)

lemma *pdom-graph-pfun*: *pdom* (*graph-pfun R*) ⊆ *Domain R*
by (*transfer*, *simp add*: *graph-map-dom fst-eq-Domain Domain-mk-functional*)

lemma *pdom-functional-graph-pfun* [*simp*]:
functional R ⟹ *pdom* (*graph-pfun R*) = *Domain R*
by (*transfer*, *simp add*: *dom-map-graph mk-functional-fp*)

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-graph f*)
by (*transfer*, *simp add*: *finite-dom-graph*)

lemma *pdom-map-pfun* [*simp*]: *pdom* (*map-pfun F G*) = *pdom G*
unfolding *map-pfun-def* **by** (*safe*, *simp-all*; *metis dom-map-option-comp pdom.abs-eq pdom.rep-eq*)

lemma *rel-comp-pfun*: *R O pfun-graph f* = (λ *p*. (*fst p*, *pfun-app f* (*snd p*))) ‘ (*R* $\rhd_r$ *pdom*(*f*))
by (*transfer*, *auto simp add*: *rel-comp-map rel-ranres-def*)

lemma *pdom-empty-iff-dom-empty*: *f* = $\{\}_p$ $\longleftrightarrow$ *pdom f* = {}
by (*transfer*, *simp*)

lemma *empty-map-pfunD* [*dest*!]: $\{\}_p$ = *map-pfun f F* $\Longrightarrow$ *F* = $\{\}_p$
by (*metis pdom-empty-iff-dom-empty pdom-map-pfun*)

14.6 Range laws

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

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

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

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

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 (*simp add*: *pdom.rep-eq pran-rep-eq*)

lemma *pran-pdom*: *pran F* = *pfun-app F* ‘ *pdom F*
by (*transfer*, *force simp add*: *dom-def*)

lemma *pran-override* [*simp*]: *pran* (*f* ⊕ *g*) = *pran*(*g*) ∪ *pran*(*pdom*(*g*) $-\lhd_p$ *f*)
by (*transfer*, *auto simp add*: *restrict-map-def dom-def map-add-def option.case-eq-if*)

14.7 Graph laws

lemma *pfun-graph-inv* [*code-unfold*]: *graph-pfun* (*pfun-graph f*) = *f*
by (*transfer*, *simp add*: *mk-functional-fp*)

lemma *pfun-eq-graph*: *f* = *g* ⟷ *pfun-graph f* = *pfun-graph g*
by (*metis pfun-graph-inv*)

lemma *Dom-pfun-graph*: *Domain* (*pfun-graph f*) = *pdom f*
by (*transfer*, *simp add*: *dom-map-graph*)

lemma *Range-pfun-graph*: *Range* (*pfun-graph f*) = *pran f*
by (*transfer*, *auto simp add*: *ran-map-graph*[*THEN sym*] *ran-def*)

lemma *card-pfun-graph*: *finite* (*pdom f*) ⟹ *card* (*pfun-graph f*) = *card* (*pdom f*)
by (*transfer*, *simp add*: *card-map-graph dom-map-graph finite-dom-graph*)

lemma *functional-pfun-graph* [*simp*]: *functional* (*pfun-graph f*)
by (*transfer*, *simp*)

lemma *pfun-graph-zero*: *pfun-graph* ⊥ = {}
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* ∩$_p$ *g*) = *pfun-graph f* ∩ *pfun-graph g*
apply (*transfer*, *simp add*: *map-graph-def*, *safe*, *simp-all add*: *domIff*)
apply (*metis option.discI*)
apply (*metis ifSomeE*)
done

lemma *pfun-graph-domres*: *pfun-graph* (*A* ◁$_p$ *f*) = (*A* ◁$_r$ *pfun-graph f*)
by (*transfer*, *simp add*: *rel-domres-math-def map-graph-def restrict-map-def*, *metis option.simps*(*3*))

lemma *pfun-graph-override*: *pfun-graph* (*f* ⊕ *g*) = *pfun-graph f* ⊕ *pfun-graph g*
by (*transfer*, *simp add*: *map-add-def oplus-set-def rel-domres-def map-graph-def option.case-eq-if*, *safe*, *simp-all*)
(*metis option.collapse*)+

lemma *pfun-graph-update*: *pfun-graph* (*f*(*k* ↦ *v*)$_p$) = *insert* (*k*, *v*) ((− {*k*}) ◁$_r$ *pfun-graph f*)
by (*transfer*, *simp add*: *map-graph-update*)

lemma *pfun-graph-comp*: *pfun-graph* (*f* ∘$_p$ *g*) = *pfun-graph g O pfun-graph f*
by (*transfer*, *simp add*: *map-graph-comp*)

lemma *comp-pfun-graph*: *pfun-graph f O pfun-graph g* = *pfun-graph* ($g \circ_p f$)
by (*simp add*: *pfun-graph-comp*)

lemma *pfun-graph-pfun-inv*: *pfun-inj f* $\Longrightarrow$ *pfun-graph* (*pfun-inv f*) = (*pfun-graph f*)$^{-1}$
by (*transfer*, *simp add*: *map-graph-map-inv*)

lemma *pfun-graph-pabs*: *pfun-graph* ($\lambda\ x \in A \mid P\ x \cdot f\ x$) = $\{(k, v).\ k \in A \wedge P\ k \wedge v = f\ k\}$
unfolding *pabs-def* **by** (*transfer*, *auto simp add*: *map-graph-def restrict-map-def*)

lemma *pfun-graph-le-iff*:
pfun-graph f $\subseteq$ *pfun-graph g* $\longleftrightarrow$ $f \subseteq_p g$
by (*simp add*: *inf.order-iff pfun-eq-graph pfun-graph-inter*)

lemma *pfun-member-iff* [*simp*]: $(k, v) \in$ *pfun-graph f* $\longleftrightarrow$ ($k \in$ *pdom*(f) $\wedge$ *pfun-app f k* = v)
by (*transfer*, *auto simp add*: *map-graph-def*)

lemma *pfun-graph-rres*: *pfun-graph* ($f \rhd_p A$) = *pfun-graph f* $\rhd_r A$
by (*transfer*, *auto simp add*: *map-graph-def rel-ranres-def ran-restrict-map-def*)

14.8 Graph Transfer Setup

definition *cr-pfung* :: ($'a \leftrightarrow 'b$) $\Rightarrow$ $'a \nrightarrow 'b \Rightarrow$ *bool* **where**
cr-pfung f g = (*f* = *pfun-graph g*)

lemma *Domainp-cr-pfung* [*transfer-domain-rule*]: *Domainp cr-pfung* = *functional*
unfolding *cr-pfung-def Domainp-iff*[*abs-def*]
by (*auto simp add*: *fun-eq-iff*, *metis graph-map-inv pfun-graph.abs-eq*)

bundle *pfun-graph-lifting*
begin

unbundle *lifting-syntax*

lemma *bi-unique-cr-pfung* [*transfer-rule*]: *bi-unique cr-pfung*
unfolding *cr-pfung-def bi-unique-def* **by** (*auto simp add*: *pfun-eq-graph*)

lemma *right-total-cr-pfung* [*transfer-rule*]: *right-total cr-pfung*
unfolding *cr-pfung-def right-total-def* **by** *simp*

lemma *cr-pfung-empty* [*transfer-rule*]: *cr-pfung* {} $\{\}_p$
unfolding *cr-pfung-def* **by** (*simp add*: *pfun-graph-zero*)

lemma *cr-pfung-dom* [*transfer-rule*]: (*cr-pfung* ===> (=)) *Domain pdom*
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *Dom-pfun-graph*)

lemma *cr-pfung-ran* [*transfer-rule*]: (*cr-pfung* ===> (=)) *Range pran*
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *Range-pfun-graph*)

lemma *cr-pfung-id* [*transfer-rule*]: ((=) ===> *cr-pfung*) *Id-on pId-on*
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *pfun-graph-pId-on*)

lemma *cr-pfung-ovrd* [*transfer-rule*]: (*cr-pfung* ===> *cr-pfung* ===> *cr-pfung*) (⊕) (⊕)
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *pfun-graph-override*)

lemma *cr-pfung-ovrd* [*transfer-rule*]: (*cr-pfung* ===> *cr-pfung* ===> *cr-pfung*) (*O*) (λ *x y*. *y* ∘$_p$ *x*)
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *pfun-graph-comp*)

lemma *cr-pfung-dres* [*transfer-rule*]: ((=) ===> *cr-pfung* ===> *cr-pfung*) (◁$_r$) (◁$_p$)
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *pfun-graph-domres*)

lemma *cr-pfung-rres* [*transfer-rule*]: (*cr-pfung* ===> (=) ===> *cr-pfung*) (▷$_r$) (▷$_p$)
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *pfun-graph-rres*)

lemma *cr-pfung-le* [*transfer-rule*]: (*cr-pfung* ===> *cr-pfung* ===> (=)) (≤) (≤)
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *pfun-graph-le-iff*)

lemma *cr-pfung-update* [*transfer-rule*]: (*cr-pfung* ===> (=) ===> (=) ===> *cr-pfung*) (λ *f k v*. *insert* (*k*, *v*) ((− {*k*}) ◁$_r$ *f*)) *pfun-upd*
unfolding *rel-fun-def cr-pfung-def* **by** (*simp add*: *pfun-graph-update*)

end

14.9 Partial Injections

lemma *pfun-inj-empty* [*simp*]: *pfun-inj* {}$_p$
by (*transfer*, *simp*)

lemma *pinj-pId-on* [*simp*]: *pfun-inj* (*pId-on A*)
by (*transfer*, *auto simp add*: *inj-on-def*)

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

lemma *pfun-inj-inv*: *pfun-inj f* ⟹ *pfun-inj* (*pfun-inv f*)
by (*transfer*, *simp add*: *inj-map-inv*)

lemma *f-pfun-inv-f-apply*: ⟦ *pfun-inj f*; *x* ∈ *pran f* ⟧ ⟹ *f*(*pfun-inv f*(*x*)$_p$)$_p$ = *x*
by (*transfer*, *auto simp add*: *ranI*)

lemma *pfun-inv-f-f-apply*: ⟦ *pfun-inj f*; *x* ∈ *pdom f* ⟧ ⟹ *pfun-inv f*(*f*(*x*)$_p$)$_p$ = *x*

by (*transfer*, *auto simp add*: *ranI*)

lemma *pfun-inj-upd*: ⟦ *pfun-inj f*; $v \notin$ *pran f* ⟧ ⟹ *pfun-inj* $(f(k \mapsto v)_p)$
apply (*transfer*, *simp-all*, *safe*)
apply (*meson f-the-inv-into-f inj-on-fun-updI*)
apply *fastforce*
done

lemma *pfun-inj-dres*: *pfun-inj f* ⟹ *pfun-inj* $(A \lhd_p f)$
by (*transfer*, *auto simp add*: *inj-on-def*)

lemma *pfun-inj-rres*: *pfun-inj f* ⟹ *pfun-inj* $(f \rhd_p A)$
by (*transfer*, *metis dom-map-inv inj-map-inv map-inv-dom-res map-inv-map-inv map-inv-ran-res ran-ran-restrict restrict-map-inj-on*)

lemma *pfun-inj-comp*: ⟦ *pfun-inj f*; *pfun-inj g* ⟧ ⟹ *pfun-inj* $(f \circ_p g)$
by (*transfer*, *auto simp add*: *inj-on-def map-comp-def option.case-eq-if dom-def*)

lemma *pfun-inj-ovrd*: ⟦ *pfun-inj f*; *pfun-inj g*; *pran f* ∩ *pran g* = {} ⟧ ⟹ *pfun-inj* $(f \oplus g)$
by (*transfer*, *force simp add*: *inj-on-def map-add-def option.case-eq-if dom-def*)

lemma *pfun-inv-dres*: *pfun-inj f* ⟹ *pfun-inv* $(A \lhd_p f) = (\textit{pfun-inv}\ f) \rhd_p A$
by (*transfer*, *simp add*: *map-inv-dom-res*)

lemma *pfun-inv-rres*: *pfun-inj f* ⟹ *pfun-inv* $(f \rhd_p A) = A \lhd_p (\textit{pfun-inv}\ f)$
by (*transfer*, *simp add*: *map-inv-ran-res*)

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

lemma *pdom-pfun-inv* [*simp*]: *pdom* (*pfun-inv f*) = *pran f*
by (*simp add*: *pran-rep-eq*, *transfer*, *simp*)

lemma *pfun-inv-add*:
assumes *pfun-inj f pfun-inj g pran f* ∩ *pran g* = {}
shows *pfun-inv* $(f \oplus g) = (\textit{pfun-inv}\ f \rhd_p (- \textit{pdom}\ g)) \oplus \textit{pfun-inv}\ g$
using *assms* **by** (*simp add*: *pran-rep-eq*, *transfer*, *safe*, *meson map-inv-add*)

lemma *pfun-inv-upd*:
assumes *pfun-inj f v* $\notin$ *pran f*
shows *pfun-inv* $(f(k \mapsto v)_p) = (\textit{pfun-inv}\ ((- \{k\}) \lhd_p f))(v \mapsto k)_p$
using *assms* **by** (*simp add*: *pran-rep-eq*, *transfer*, *meson map-inv-upd*)

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

lemma *pdom-res-override* [*simp*]: $A \lhd_p (f \oplus g) = (A \lhd_p f) \oplus (A \lhd_p g)$
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*)

lemma *pdom-res-frame-update* [*simp*]:
⟦ $x \in pdom(f)$; $(-\{x\}) \lhd_p f = (-\{x\}) \lhd_p g$ ⟧ ⟹ $g(x \mapsto f(x)_p)_p = f$
by *transfer* (*metis* (*mono-tags*, *opaque-lifting*) *domIff fun-upd-triv fun-upd-upd option.exhaust-sel*
restrict-complement-singleton-eq)

lemma *pdres-rres-commute*: $A \lhd_p (P \rhd_p B) = (A \lhd_p P) \rhd_p B$
by (*transfer*, *simp add*: *map-dres-rres-commute*)

lemma *pdom-nres-disjoint*: $pdom(f) \cap A = \{\} \Longrightarrow (- A) \lhd_p f = f$
by (*metis disjoint-eq-subset-Compl inf.absorb2 pdom-res-pdom pdom-res-twice*)

lemma *pranres-pdom* [*simp*]: $pdom\ (f \rhd_p A) \lhd_p f = f \rhd_p A$
by (*transfer*, *simp add*: *restrict-map-def fun-eq-iff ran-restrict-map-def option.case-eq-if*)
(*metis* (*full-types*, *lifting*) *bind.bind-lunit bind.bind-lzero domIff not-None-eq*)

lemma *pdom-pranres* [*simp*]: $pdom\ (f \rhd_p A) \subseteq pdom\ f$
by (*metis inf.absorb-iff1 inf.commute pdom-pdom-res pdom-res-pdom pdom-res-swap*)

lemma *pfun-split-domain*: $A \lhd_p f \oplus (- A) \lhd_p f = f$
by (*transfer*, *auto simp add*: *restrict-map-def map-add-def fun-eq-iff option.case-eq-if*)

14.11 Range restriction laws

lemma *pran-res-UNIV* [*simp*]: $f \rhd_p UNIV = f$
by (*transfer*, *simp add*: *ran-restrict-map-def*)

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

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-twice* [*simp*]: $f \rhd_p A \rhd_p B = f \rhd_p (A \cap B)$
by (*transfer*, *simp*)

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*: $(f \oplus g) \rhd_p A \subseteq_p (f \rhd_p A) \oplus (g \rhd_p A)$
by (*transfer*, *auto simp add*: *map-add-def ran-restrict-map-def map-le-def option.case-eq-if*)

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

lemma *pranres-le*: $A \subseteq B \Longrightarrow f \rhd_p A \leq f \rhd_p B$
by (*simp add*: *pfun-graph-le-iff*[*THEN sym*] *pfun-graph-comp pfun-graph-rres rel-comp-mono rel-ranres-le*)

lemma *pranres-neg-ran* [*simp*]: $P \rhd_p - \ pran\ P = \{\}_p$
by (*transfer*, *simp add*: *ran-restrict-map-def fun-eq-iff option.case-eq-if bind-eq-None-conv*, *meson option.exhaust-sel*)

14.12 Preimage Laws

lemma *ppreimageI* [*intro*!]: $[\![\ x \in pdom(f);\ f(x)_p \in A\ ]\!] \Longrightarrow x \in pdom\ (f \rhd_p A)$
by (*metis* (*full-types*) *insertI1 pdom-upd pfun-upd-ext pran-res-upd-1*)

lemma *ppreimageD*: $x \in pdom\ (f \rhd_p A) \Longrightarrow \exists\ y \in A.\ f(x)_p = y$
by (*transfer*, *auto simp add*: *ran-restrict-map-def*)

lemma *ppreimageE* [*elim*!]: $[\![\ x \in pdom\ (f \rhd_p A);\ \bigwedge y.\ [\![\ x \in pdom(f);\ y \in A;\ f(x)_p = y\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*metis* (*no-types*) *pdom-pranres ppreimageD subsetD*)

lemma *mem-pimage-iff*: $x \in pran\ (A \lhd_p f) \longleftrightarrow (\exists\ y \in A \cap pdom(f).\ f(y)_p = x)$
by (*auto simp add*: *pran-pdom*)

lemma *ppreimage-inter* [*simp*]: $pdom\ (f \rhd_p (A \cap B)) = pdom\ (f \rhd_p A) \cap pdom\ (f \rhd_p B)$
by *fastforce*

14.13 Composition

lemma *pcomp-apply* [*simp*]: $[\![\ x \in pdom(g)\ ]\!] \Longrightarrow (f \circ_p g)(x)_p = f(g(x)_p)_p$
by (*transfer*, *auto*)

lemma *pcomp-mono*: $[\![\ f \leq f';\ g \leq g'\ ]\!] \Longrightarrow f \circ_p g \leq f' \circ_p g'$
by (*simp add*: *pfun-graph-le-iff*[*THEN sym*] *pfun-graph-comp relcomp-mono*)

lemma *pdom-UNIV-comp*: $pdom\ f = UNIV \Longrightarrow pdom\ (f \circ_p g) = pdom\ g$
by *simp*

14.14 Entries

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

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

lemma *pran-pfun-entries* [*simp*]: *pran* (*pfun-entries A f*) = *f* ‘ *A*
by (*transfer*, *simp add*: *ran-def*, *auto*)

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

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

lemma *pdom-res-entries*: *A* ◁$_p$ *pfun-entries B f* = *pfun-entries* (*A* ∩ *B*) *f*
by (*transfer*, *auto simp add*: *fun-eq-iff restrict-map-def*)

lemma *pfuse-empty* [*simp*]: *pfuse* {}$_p$ *g* = {}$_p$
by (*simp add*: *pfuse-def*)

lemma *pfuse-app* [*simp*]:
⟦ *e* ∈ *pdom F*; *e* ∈ *pdom G* ⟧ ⟹ (*pfuse F G*)(*e*)$_p$ = (*F*(*e*)$_p$, *G*(*e*)$_p$)
by (*metis* (*no-types*, *lifting*) *IntI pfun-entries-apply-1 pfuse-def*)

lemma *pdom-pfuse* [*simp*]: *pdom* (*pfuse f g*) = *pdom*(*f*) ∩ *pdom*(*g*)
by (*auto simp add*: *pfuse-def*)

lemma *pfuse-upd*:
pfuse (*f*(*k* ↦ *v*)$_p$) *g* =
(*if k* ∈ *pdom g then* (*pfuse* ((−{*k*}) ◁$_p$ *f*) *g*)(*k* ↦ (*v*, *pfun-app g k*))$_p$ *else pfuse f g*)
by (*simp add*: *pfuse-def*, *transfer*, *auto simp add*: *fun-eq-iff*)

14.15 Lambda abstraction

lemma *pabs-cong*:
assumes *A* = *B* ⋀ *x*. *x* ∈ *A* ⟹ *P*(*x*) = *Q*(*x*) ⋀ *x*. ⟦ *x* ∈ *A*; *P x* ⟧ ⟹ *F*(*x*) = *G*(*x*)
shows (λ *x* ∈ *A* | *P x* · *F*(*x*)) = (λ *x* ∈ *B* | *Q x* · *G*(*x*))
using *assms* **unfolding** *pabs-def*
by (*transfer*, *auto simp add*: *restrict-map-def fun-eq-iff*)

lemma *pabs-apply* [*simp*]: ⟦ *y* ∈ *A*; *P y* ⟧ ⟹ (λ *x* ∈ *A* | *P x* · *f x*) (*y*)$_p$ = *f y*
by (*simp add*: *pabs-def*)

lemma *pdom-pabs* [*simp*]: *pdom* (λ *x* ∈ *A* | *P x* · *f x*) = *A* ∩ *Collect P*
by (*simp add*: *pabs-def*)

lemma *pran-pabs* [*simp*]: *pran* (λ *x* ∈ *A* | *P x* · *f x*) = {*f x* | *x*. *x* ∈ *A* ∧ *P x*}
unfolding *pabs-def*
by (*transfer*, *auto simp add*: *ran-def restrict-map-def*)

lemma *pabs-eta* [*simp*]: $(\lambda\ x \in pdom(f) \cdot f(x)_p) = f$
by (*simp add*: *pabs-def*, *transfer*, *auto simp add*: *fun-eq-iff domIff restrict-map-def*)

lemma *pabs-id* [*simp*]: $(\lambda\ x \in A \mid P\ x \cdot x) = pId\text{-}on\ \{x{\in}A.\ P\ x\}$
unfolding *pabs-def* **by** (*transfer*, *simp add*: *restrict-map-def*)

lemma *pfun-entries-pabs*: *pfun-entries A f* $= (\lambda\ x \in A \cdot f\ x)$
by (*simp add*: *pabs-def*, *transfer*, *auto*)

lemma *pabs-empty* [*simp*]: $(\lambda\ x{\in}\{\} \cdot f(x)) = \{\}_p$
by (*simp add*: *pabs-def*)

lemma *pabs-insert-maplet*: $(\lambda\ x{\in}insert\ y\ A \cdot f(x)) = (\lambda\ x{\in}A \cdot f(x)) \oplus \{y \mapsto f(y)\}_p$
by (*simp add*: *pabs-def*, *transfer*, *auto simp add*: *restrict-map-insert*)

This rule can perhaps be simplified

lemma *pcomp-pabs*:
$(\lambda\ x \in A \mid P\ x \cdot f\ x) \circ_p (\lambda\ x \in B \mid Q\ x \cdot g\ x)$
$= (\lambda\ x \in pdom\ (pabs\ B\ Q\ g \rhd_p (A \cap Collect\ P)) \cdot (f\ (g\ x)))$
proof −
have *pabs A P f* $\circ_p$ *pabs B Q g* $= (\lambda\ x \in pdom\ (pabs\ A\ P\ f \circ_p pabs\ B\ Q\ g) \cdot (pfun\text{-}app\ (pabs\ A\ P\ f \circ_p pabs\ B\ Q\ g))\ x)$
by (*rule pabs-eta*[*THEN sym*, *of* $(\lambda\ x \in A \mid P\ x \cdot f\ x) \circ_p (\lambda\ x \in B \mid Q\ x \cdot g\ x)$])
also have ... $= (\lambda\ x \in pdom\ (pabs\ B\ Q\ g \rhd_p (A \cap Collect\ P)) \cdot (f\ (g\ x)))$
unfolding *pabs-def*
by (*transfer*, *auto simp add*: *restrict-map-def map-comp-def ran-restrict-map-def fun-eq-iff*)
finally show *?thesis* **.**
qed

lemma *pabs-rres* [*simp*]: *pabs A P f* $\rhd_p$ *B* = *pabs A* $(\lambda\ x.\ P\ x \wedge f\ x \in B)$ *f*
by (*simp add*: *pabs-def*, *transfer*, *auto simp add*: *ran-restrict-map-def restrict-map-def*)

lemma *pabs-simple-comp* [*simp*]: $(\lambda\ x \cdot f\ x) \circ_p g(k \mapsto v)_p = ((\lambda\ x \cdot f\ x) \circ_p g)(k \mapsto f\ v)_p$
by (*simp add*: *pabs-def*, *transfer*, *auto*)

lemma *pabs-comp*: $(\lambda\ x \in A \cdot f\ x) \circ_p g = (\lambda\ x \in pdom\ (g \rhd_p A) \cdot f\ (pfun\text{-}app\ g\ x))$
by (*metis pabs-eta pcomp-pabs pdom-pId-on pdom-pabs*)

lemma *map-pfun-pabs* [*simp*]: *map-pfun f* $(\lambda\ x{\in}A \mid B(x) \cdot g(x)) = (\lambda\ x{\in}A \mid B(x) \cdot f(g(x)))$
by (*simp add*: *pfun-eq-iff*)

14.16 Singleton Partial Functions

definition *pfun-singleton* :: $('a \rightharpoonup 'b) \Rightarrow bool$ **where**
$pfun\text{-}singleton\ f = (\exists\ k\ v.\ f = \{k \mapsto v\}_p)$

lemma *pfun-singleton-dom*: $pfun\text{-}singleton\ f \longleftrightarrow (\exists\ k.\ pdom(f) = \{k\})$
by (*simp add*: *pfun-singleton-def*, *safe*, *simp-all*)
(*metis insertI1 override-lzero pdom-res-pdom pfun-ovrd-single-upd*)

lemma *pfun-singleton-maplet* [*simp*]:
$pfun\text{-}singleton\ \{k \mapsto v\}_p$
by (*auto simp add*: *pfun-singleton-def*)

definition *dest-pfsingle* :: $('a \rightharpoonup 'b) \Rightarrow 'a \times 'b$ **where**
$dest\text{-}pfsingle\ f = (THE\ (k, v).\ f = \{k \mapsto v\}_p)$

lemma *dest-pfsingle-maplet* [*simp*]: $dest\text{-}pfsingle\ \{k \mapsto v\}_p = (k, v)$
unfolding *dest-pfsingle-def*
by (*rule the-equality*, *simp-all add*: *prod.case-eq-if*)
(*metis fst-eqD pdom-res-zero pdom-upd pdom-zero pran-upd pran-zero prod.expand singleton-insert-inj-eq sndI*)

14.17 Summation

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

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

lemma *pfun-sum-upd-1*:
assumes $finite(pdom(m))$ $k \notin pdom(m)$
shows $pfun\text{-}sum\ (m(k \mapsto v)_p) = pfun\text{-}sum\ m + v$
proof −
from *assms*(*2*) **have** $(\sum x{\in}pdom\ m.\ if\ k = x\ then\ v\ else\ m(x)_p) = sum\ (pfun\text{-}app\ m)\ (pdom\ m)$
by (*auto intro*!: *sum.cong*)
thus *?thesis*
by (*simp-all add*: *pfun-sum-def assms add.commute cong*: *sum.cong*)
qed

lemma *pfun-sums-upd-2*:
assumes $finite(pdom(m))$
shows $pfun\text{-}sum\ (m(k \mapsto v)_p) = pfun\text{-}sum\ ((-\ \{k\}) \lhd_p m) + v$
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*}) ◁$_p$ *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*) ◁$_p$ *f*) = $f(x)_p$ + *pfun-sum* (*A* ◁$_p$ *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* ◁$_p$ *f*) = *pfun-sum f* − (*pfun-sum* ((− *A*) ◁$_p$ *f*))
proof −
have *1*:*A* ∩ *pdom*(*f*) = *pdom*(*f*) − (*pdom*(*f*) − *A*)
by (*auto*)
have *2*: *sum* (*pfun-app f*) (*pdom f*) − *sum* (*pfun-app f*) (*pdom f* − *A*) =
sum (*pfun-app f*) (*pdom f*) − *sum* (*pfun-app f*) (− *A* ∩ *pdom f*)
by (*auto simp add*: *sum-diff Int-commute boolean-algebra-class.diff-eq assms*)
show *?thesis*
by (*simp add*: *pfun-sum-def 1 2 sum-diff assms*)
qed

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

14.18 Conversions

definition *list-pfun* :: '*a list* ⇒ *nat* ⇸ '*a* **where**
list-pfun xs = (λ *i* | *0* < *i* ∧ *i* ≤ *length xs* · *xs* ! (*i*−*1*))

lemma *pdom-list-pfun* [*simp*]: *pdom* (*list-pfun xs*) = {*1*..*length xs*}
by (*auto simp add*: *list-pfun-def*)

lemma *pran-list-pfun* [*simp*]: *pran* (*list-pfun xs*) = *set xs*
by (*simp add*: *list-pfun-def*, *safe*, *simp-all*)
(*metis One-nat-def Suc-leI diff-Suc-1 in-set-conv-nth zero-less-Suc*)

lemma *pfun-app-list-pfun*: ⟦ *0* < *i*; *i* ≤ *length xs* ⟧ ⟹ (*list-pfun xs*)(i)$_p$ = *xs* !
(*i* − *1*)
by (*simp add*: *list-pfun-def*)

lemma *pfun-graph-list-pfun*: *pfun-graph* (*list-pfun xs*) = (λ *i*. (*i*, *xs* ! (*i* − *1*))) ‘
{*1*..*length xs*}

by (*simp add*: *list-pfun-def pfun-graph-pabs*, *auto*)

lemma *range-list-pfun*:
range list-pfun = {*f* :: *nat* ⇸ *'a*. ∃ *i*. *pdom*(*f*) = {*1..i*}}
proof (*rule set-eqI*, *rule iffI*)
fix *f* :: *nat* ⇸ *'a*
assume *f* ∈ *range list-pfun*
thus *f* ∈ {*f*. ∃ *i*. *pdom f* = {*1..i*}}
by *auto*
next
fix *f* :: *nat* ⇸ *'a*
assume *f* ∈ {*f*. ∃ *i*. *pdom f* = {*1..i*}}
thus *f* ∈ *range list-pfun*
proof (*unfold list-pfun-def pabs-def image-def*, *transfer*)
fix *f* :: *nat* ⇒ *'a option*
assume *f* ∈ {*f*. ∃ *i*. *dom f* = {*1..i*}}
then obtain *i* **where** *i*:*dom f* = {*1..i*}
by *blast*
hence *1*: ⋀*x*. *dom f* = {*Suc 0..i*} ⟹ *0* < *x* ⟹ *x* ≤ *i* ⟹ *f x* = *Some* (*the* (*f x*))
by (*metis Suc-leI atLeastAtMost-iff domIff option.exhaust-sel*)
with *i* **have** *2*:*f 0* = *None*
using *atLeastAtMost-iff not-one-le-zero* **by** *blast*
from *i 1 2* **have** *f*: *f* = (λ*xa*. *Some* (*map* (*the* ∘ *f* ∘ *nat*) [*1..int i*] ! (*xa* − *Suc 0*))) |‘ {*ia*. *0* < *ia* ∧ *ia* ≤ *length* (*map* (*the* ∘ *f* ∘ *nat*) [*1..int i*])}
by (*auto simp add*: *fun-eq-iff restrict-map-def*)
have *3*: (λ*xa*. *Some* (*map* (*the* ∘ *f* ∘ *nat*) [*1..int i*] ! (*xa* − *Suc 0*))) |‘ {*ia*. *0* < *ia* ∧ *ia* ≤ *length* (*map* (*the* ∘ *f* ∘ *nat*) [*1..int i*])} ∈ {*y*. ∃ *x*∈*UNIV*. *y* = (λ*xa*. *if xa* ∈ *UNIV then Some* (*x* ! (*xa* − *1*)) *else None*) |‘ (*UNIV* ∩ {*i*. *0* < *i* ∧ *i* ≤ *length x*})}
by (*auto simp add*: *fun-eq-iff restrict-map-def*)
show *f* ∈ {*y*. ∃ *x*∈*UNIV*. *y* = (λ*xa*. *if xa* ∈ *UNIV then Some* (*x* ! (*xa* − *1*)) *else None*) |‘ (*UNIV* ∩ {*i*. *0* < *i* ∧ *i* ≤ *length x*})}
using *3 f* **by** *auto*
qed
qed

lemma *list-pfun-le-iff-prefix* [*simp*]: *list-pfun xs* ≤ *list-pfun ys* ⟷ *xs* ≤ *ys*
apply (*simp add*: *pfun-le-iff*, *safe*, *simp-all add*: *pfun-app-list-pfun list-le-prefix-iff*)
apply (*metis Suc-leI Suc-le-mono atLeastAtMost-iff diff-Suc-Suc le0 minus-nat.diff-0*)
apply (*metis Suc-le-D Suc-le-eq diff-Suc-Suc diff-zero*)
done

lemma *pfun-upd-le-iff*: (*f*(*k* ↦ *v*)$_p$ ⊆$_p$ *g*) = (*k* ∈ *pdom g* ∧ *g*(*k*)$_p$ = *v* ∧ (− {*k*}) ◁$_p$ *f* ⊆$_p$ *g*)
by (*auto simp add*: *pfun-le-iff*)

lemma *pfun-upd-le-pfun-upd*: (*f*(*k* ↦ *v*)$_p$ ⊆$_p$ *g*(*k* ↦ *v*)$_p$) = ((− {*k*}) ◁$_p$ *f* ⊆$_p$ (− {*k*}) ◁$_p$ *g*)

by (*auto simp add*: *pfun-le-iff*)

14.19 Partial Function Lens

definition *pfun-lens* :: $'a \Rightarrow ('b \Longrightarrow ('a, 'b)\ pfun)$ **where**
[*lens-defs*]: *pfun-lens i* = (| *lens-get* = λ *s*. $s(i)_p$, *lens-put* = λ *s v*. $s(i \mapsto v)_p$ |)

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

lemma *pfun-lens-src*: $\mathcal{S}_{pfun\text{-}lens\ i} = \{f.\ i \in pdom(f)\}$
by (*simp add*: *lens-defs lens-source-def*, *transfer*, *force*)

lemma *lens-override-pfun-lens*:
$x \in pdom(g) \Longrightarrow f \oplus_L g$ *on pfun-lens x* $= f \oplus (\{x\} \lhd_p g)$
by (*simp add*: *lens-defs pfun-ovrd-single-upd*)

14.20 Prism Functions

We can use prisms to index a type and construct partial functions.

definition *prism-fun* :: $('a \Longrightarrow_\triangle 'e) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow bool \times 'b) \Rightarrow ('e \nrightarrow 'b)$
where [*code-unfold*]: *prism-fun c A PB* = (λ $x{\in}build_c$ ‘ *A* | *fst* (*PB* (*the* ($match_c$ *x*))) $\cdot$ *snd* (*PB* (*the* ($match_c$ *x*))))

definition *prism-fun-upd* :: $('e \nrightarrow 'b) \Rightarrow ('a \Longrightarrow_\triangle 'e) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow bool \times 'b) \Rightarrow ('e \nrightarrow 'b)$
where [*code-unfold*]: *prism-fun-upd F c A PB* = *F* $\oplus$ *prism-fun c A PB*

nonterminal *prism-maplet* **and** *prism-maplets*

syntax
-prism-maplet :: *id* $\Rightarrow$ *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *prism-maplet* (-{- ∈ -./ -} $\Rightarrow$ -)
-prism-maplet-mem :: *id* $\Rightarrow$ *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *logic* $\Rightarrow$ *prism-maplet* (-{- ∈ -} $\Rightarrow$ -)
-prism-maplet-simple :: *id* $\Rightarrow$ *pttrn* $\Rightarrow$ *logic* $\Rightarrow$ *prism-maplet* (- - $\Rightarrow$ -)
:: *prism-maplet* $\Rightarrow$ *prism-maplets* (-)
-prism-Maplets :: [*prism-maplet*, *prism-maplets*] $\Rightarrow$ *prism-maplets* (- |/ -)
-prism-fun-upd :: *logic* $\Rightarrow$ *prism-maplets* $\Rightarrow$ *logic* (-'(-') [*900*, *0*] *900*)
-prism-fun :: *prism-maplets* $\Rightarrow$ *logic* ($\{\text{-}\}_p$)

translations
f(*c*{*v* ∈ *A*. *P*} $\Rightarrow$ *B*) == *CONST prism-fun-upd f c A* (λ *v*. (*P*, *B*))
f(*c*{*v* ∈ *A*} $\Rightarrow$ *B*) == *f*(*c*{*v* ∈ *A*. *CONST True*} $\Rightarrow$ *B*)
f(*c v* $\Rightarrow$ *B*) == *f*(*c*{*v* ∈ *CONST UNIV*} $\Rightarrow$ *B*)
-prism-fun-upd m (*-prism-Maplets xy ms*) $\rightleftharpoons$ *-prism-fun-upd* (*-prism-fun-upd m xy*) *ms*
-prism-fun ms $\rightleftharpoons$ *-prism-fun-upd* $\{\}_p$ *ms*

-prism-fun (*-prism-Maplets ms1 ms2*) ↼ *-prism-fun-upd* (*-prism-fun ms1*) *ms2*
-prism-Maplets ms1 (*-prism-Maplets ms2 ms3*) ↼ *-prism-Maplets* (*-prism-Maplets ms1 ms2*) *ms3*

lemma *dom-prism-fun*: *wb-prism c* ⟹ *pdom*(*prism-fun c A PB*) = {*build*$_c$ *v* | *v*. *v* ∈ *A* ∧ *fst* (*PB v*)}
by (*simp add*: *prism-fun-def*, *auto*)

lemma *prism-fun-compat*: *c* ▽ *d* ⟹ *prism-fun c A PB* ## *prism-fun d B QB*
by (*auto intro*!: *pfun-indep-compat simp add*: *prism-fun-def prism-diff-build*)

lemma *prism-fun-commute*: *c* ▽ *d* ⟹ *prism-fun c A PB* ⊕ *prism-fun d B QB* = *prism-fun d B QB* ⊕ *prism-fun c A PB*
by (*meson override-comm prism-fun-compat*)

lemma *prism-fun-apply*: ⟦ *wb-prism c*; *v* ∈ *A*; *fst* (*PB v*) ⟧ ⟹ (*prism-fun c A PB*)(*build*$_c$ *v*)$_p$ = *snd* (*PB v*)
by (*simp add*: *prism-fun-def*)

lemma *prism-fun-update-app-1* [*simp*]: ⟦ *wb-prism c*; *v* ∈ *A*; *P v* ⟧ ⟹ (*f*(*c*{*x* ∈ *A*. *P*(*x*)} ⇒ *B*(*x*)))(*build*$_c$ *v*)$_p$ = *B v*
by (*simp add*: *prism-fun-def prism-fun-upd-def*)

lemma *prism-fun-update-app-2* [*simp*]: ⟦ *wb-prism c*; *wb-prism d*; *d* ▽ *c* ⟧ ⟹ (*f*(*c*{*x* ∈ *A*. *P*(*x*)} ⇒ *B*(*x*)))(*build*$_d$ *v*)$_p$ = *f*(*build*$_d$ *v*)$_p$
by (*simp add*: *prism-fun-def prism-fun-upd-def image-iff prism-diff-build*)

lemma *prism-fun-update-cancel* [*simp*]: *f*(*c*{*x* ∈ *A*. *P*(*x*)} ⇒ *g*(*x*) | *c*{*x* ∈ *A*. *P*(*x*)} ⇒ *h*(*x*)) = *f*(*c*{*x* ∈ *A*. *P*(*x*)} ⇒ *h*(*x*))
by (*simp add*: *prism-fun-def prism-fun-upd-def override-assoc*[*THEN sym*] *pfun-override-fully*)

lemma *prism-fun-update-commute*:
c ▽ *d* ⟹ *f*(*c*{*x* ∈ *A*. *P*(*x*)} ⇒ *g*(*x*) | *d*{*y* ∈ *B*. *Q*(*y*)} ⇒ *h*(*y*))
= *f*(*d*{*y* ∈ *B*. *Q*(*y*)} ⇒ *h*(*y*) | *c*{*x* ∈ *A*. *P*(*x*)} ⇒ *g*(*x*))
by (*simp add*: *prism-fun-upd-def override-assoc*[*THEN sym*] *prism-fun-commute*)

lemma *case-sum-Plus*: *case-sum f g* ‘ (*A* <+> *B*) = (*f*‘*A*) ∪ (*g*‘*B*)
by (*simp add*: *image-iff Plus-def*, *metis* (*no-types*, *lifting*) *image-Un image-cong image-image sum.case*(*1*) *sum.case*(*2*))

lemma *build-in-dom-prism-fun*: ⟦ *wb-prism c*; *x* ∈ *A*; *fst* (*PB x*) ⟧ ⟹ *build*$_c$ *x* ∈ *pdom* (*prism-fun c A PB*)
by (*auto simp add*: *dom-prism-fun*)

lemma *prism-fun-combine*:
assumes *wb-prism c wb-prism d c* ▽ *d*
shows *prism-fun c A PB* ⊕ *prism-fun d B QB* = *prism-fun* (*c* +$_\triangle$ *d*) (*A* <+> *B*) (*case-sum PB QB*)

using *assms*
apply (*simp add*: *pfun-eq-iff dom-prism-fun sum.case-eq-if prism-diff-build build-in-dom-prism-fun*)
apply *safe*
apply (*simp-all add*: *add*: *build-in-dom-prism-fun prism-diff-build prism-fun-apply*)
apply (*metis InlI build-plus-Inl sum.disc*(*1*) *sum.sel*(*1*))
apply (*metis InrI build-plus-Inr sum.disc*(*2*) *sum.sel*(*2*))
apply *metis*
apply (*metis InlI build-in-dom-prism-fun build-plus-Inl old.sum.simps*(*5*) *pfun-app-add prism-fun-apply prism-fun-commute prism-plus-wb*)
apply (*metis InrI build-plus-Inr old.sum.simps*(*6*) *prism-fun-apply prism-plus-wb*)
done

lemma *prism-diff-implies-indep-funs*:
⟦ *wb-prism c*; *wb-prism d*; *c* ▽ *d* ⟧ ⟹ *pdom*(*prism-fun c A Pσ*) ∩ *pdom*(*prism-fun d B Qϱ*) = {}
by (*auto simp add*: *dom-prism-fun prism-diff-build*)

lemma *prism-fun-cong*: ⟦ *c* = *d*; *A* = *B*; *PB* = *QB* ⟧ ⟹ *prism-fun c A PB* = *prism-fun d B QB*
by *blast*

lemma *prism-fun-cong2*:
assumes
wb-prism c_1 *wb-prism* c_2
$c_1 = c_2$ $A_1 = A_2$
⋀ *i*. *i* ∈ A_1 ⟹ P_1 *i* ⟷ P_2 *i*
⋀ *i*. ⟦ *i* ∈ A_1; P_1 *i* ⟧ ⟹ B_1 *i* = B_2 *i*
shows *prism-fun* c_1 A_1 (λ *x*. (P_1 *x*, B_1 *x*)) = *prism-fun* c_2 A_2 (λ *y*. (P_2 *y*, B_2 *y*))
using *assms*
by (*auto intro*!: *pabs-cong simp add*: *prism-fun-def*)

lemma *map-pfun-prism-fun* [*simp*]: *map-pfun f* (*prism-fun a A* (λ *x*. (*B x*, *C x*))) = *prism-fun a A* (λ *x*. (*B x*, *f* (*C x*)))
by (*simp add*: *prism-fun-def*)

lemma *prism-fun-as-map*:
wb-prism b ⟹
prism-fun b A PB = *pfun-of-map* (λ *x*. *case* $match_b$ *x of None* ⇒ *None* | *Some x* ⇒ *if x* ∈ *A* ∧ *fst* (*PB x*) *then Some* (*snd* (*PB x*)) *else None*)
by (*simp add*: *prism-fun-def pfun-eq-iff domIff pdom.abs-eq option.case-eq-if*, *safe*, *simp-all*)
(*metis* (*no-types*, *lifting*) *image-iff option.collapse option.distinct*(*1*) *wb-prism.build-match*, *metis option.discI*)

14.21 Code Generator

14.21.1 Associative Lists

lemma *relt-pfun-iff*:
relt-pfun R f g $\longleftrightarrow$ (*pdom*(*f*) = *pdom*(*g*) $\wedge$ ($\forall$ *x*$\in$*pdom*(*f*). *R* (*f*(*x*)$_p$) (*g*(*x*)$_p$)))
by (*transfer*, *auto simp add*: *rel-map-iff*)

lift-definition *pfun-of-alist* :: ('*a* $\times$ '*b*) *list* $\Rightarrow$ '*a* $\rightharpoonup$ '*b* **is** *map-of* **.**

lemma *pfun-of-alist-clearjunk*: *pfun-of-alist xs* = *pfun-of-alist* (*AList.clearjunk xs*)
by (*transfer*, *simp add*: *map-of-clearjunk*)

lemma *pfun-of-alist-Nil* [*simp*]: *pfun-of-alist* [] = {}$_p$
by (*transfer*, *simp*)

lemma *pfun-of-alist-Cons* [*simp*]: *pfun-of-alist* (*p* # *ps*) = *pfun-of-alist ps*(*fst p* $\mapsto$ *snd p*)$_p$
by (*transfer*, *metis* (*full-types*) *map-of.simps*(*2*))

lemma *dom-pfun-alist* [*simp*, *code*]: *pdom* (*pfun-of-alist xs*) = *set* (*map fst xs*)
by (*transfer*, *simp add*: *dom-map-of-conv-image-fst*)

lemma *ran-pfun-alist* [*simp*, *code*]: *pran* (*pfun-of-alist xs*) = *set* (*remdups* (*map snd* (*AList.clearjunk xs*)))
apply (*transfer*, *safe*, *simp-all*)
apply (*safe*, *simp-all*)
apply (*metis ranI ran-map-of*)
apply (*metis distinct-clearjunk map-of-clearjunk map-of-eq-Some-iff*)
done

lemma *map-graph-map-of*: *map-graph* (*map-of xs*) = *set* (*AList.clearjunk xs*)
by (*metis graph-def graph-map-of map-graph-def*)

lemma *pfun-graph-alist* [*code*]: *pfun-graph* (*pfun-of-alist xs*) = *set* (*AList.clearjunk xs*)
by (*transfer*, *meson map-graph-map-of*)

lemma *empty-pfun-alist* [*code*]: {}$_p$ = *pfun-of-alist* []
by (*transfer*, *simp*)

lemma *update-pfun-alist* [*code*]: *pfun-upd* (*pfun-of-alist xs*) *k v* = *pfun-of-alist* (*AList.update k v xs*)
by *transfer* (*simp add*: *update-conv'*)

lemma *apply-pfun-alist* [*code*]:
pfun-app (*pfun-of-alist xs*) *k* = (*if k* $\in$ *set* (*map fst xs*) *then the* (*map-of xs k*) *else undefined*)
apply (*transfer*, *simp*, *safe*)
apply (*metis map-of-eq-None-iff option.distinct*(*1*))

apply (*metis eq-fst-iff weak-map-of-SomeI*)
done

lemma *map-of-Cons-code* [*code*]:
pfun-lookup (*pfun-of-alist* []) *k* = *None*
pfun-lookup (*pfun-of-alist* ((*l*, *v*) # *ps*)) *k* = (*if l* = *k then Some v else map-of ps k*)
by (*transfer*, *simp*)+

lemma *map-pfun-alist* [*code*]:
map-pfun f (*pfun-of-alist m*) = *pfun-of-alist* (*map* (λ (*k*, *v*). (*k*, *f v*)) *m*)
by (*transfer*, *simp add*: *map-of-map*)

lemma *map-pfun-of-map* [*code*]: *map-pfun f* (*pfun-of-map g*) = *pfun-of-map* (λ *x*. *map-option f* (*g x*))
by (*auto simp add*: *map-pfun-def pfun-of-map-inject fun-eq-iff*)

lemma *pdom-res-alist* [*code*]:
A ◁$_p$ (*pfun-of-alist m*) = *pfun-of-alist* (*AList.restrict A m*)
by (*transfer*, *simp add*: *restr-conv'*)

lemma *pran-res-alist-distinct*:
distinct (*map fst xs*) ⟹ *pfun-of-alist xs* ▷$_p$ *A* = *pfun-of-alist* (*filter* (λ(*k*, *v*). *v* ∈ *A*) *xs*)
by (*induct xs*, *auto*)

lemma *pran-res-alist* [*code*]: *pfun-of-alist xs* ▷$_p$ *A* = *pfun-of-alist* (*filter* (λ(*k*, *v*). *v* ∈ *A*) (*AList.clearjunk xs*))
by (*metis distinct-clearjunk pfun-of-alist-clearjunk pran-res-alist-distinct*)

lemma *pdom-res-set-map* [*code*]:
set xs ◁$_p$ (*pfun-of-map m*) = *pfun-of-alist* (*map* (λ *x*. (*x*, *the* (*m x*))) (*filter* (λ *x*. *m x* ≠ *None*) *xs*))
proof (*induct xs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
then show *?case*
by (*simp*, *safe*; *transfer*)
(*simp add*: *restrict-map-insert*, *metis Int-insert-right-if0 Map.restrict-restrict domIff map-restrict-dom*)
qed

lemma *plus-pfun-alist* [*code*]: *pfun-of-alist f* ⊕ *pfun-of-alist g* = *pfun-of-alist* (*g* • *f*)
by (*transfer*, *simp*)

lemma *pfun-entries-alist* [*code*]: *pfun-entries* (*set ks*) *f* = *pfun-of-alist* (*map* (λ *k*.

$(k, f\ k))\ ks$)
by (*auto simp add*: *pfun-eq-iff apply-pfun-alist map-of-map prod.case-eq-if image-iff map-of-map-restrict*)

lemma *pdom-res-entries-alist* [*code*]:
$A \lhd_p$ *pfun-entries* (*set bs*) *f* =
pfun-of-alist (*map* (λ *k*. (*k*, *f k*)) (*filter* (λx. $x \in A$) *bs*))
by (*metis inter-set-filter pdom-res-entries pfun-entries-alist*)

lemma *pfun-alist-oplus-map* [*code*]:
pfun-of-alist xs $\oplus$ *pfun-of-map f* = *pfun-of-map* (λ *k*. *case f k of None* $\Rightarrow$ *map-of xs k* | *Some v* $\Rightarrow$ *Some v*)
by (*simp add*: *map-add-def oplus-pfun.abs-eq pfun-of-alist.abs-eq*)

lemma *pfun-map-oplus-alist* [*code*]:
pfun-of-map f $\oplus$ *pfun-of-alist xs* = *pfun-of-map* (λ *k*. *if k* $\in$ *set* (*map fst xs*) *then map-of xs k else f k*)
by (*simp add*: *map-add-def oplus-pfun.abs-eq pfun-of-alist.abs-eq*)
(*metis map-of-eq-None-iff option.case-eq-if option.exhaust option.sel*)

lemma *pfun-singleton-alist* [*code*]: *pfun-singleton* (*pfun-of-alist* [(*k*, *v*)]) = *True*
by *simp*

lemma *dest-pfsingle-alist* [*code*]: *dest-pfsingle* (*pfun-of-alist* [(*k*, *v*)]) = (*k*, *v*)
by *simp*

Adapted from Mapping theory

lemma *ptabulate-alist* [*code*]: *ptabulate ks f* = *pfun-of-alist* (*map* (λk. (*k*, *f k*)) *ks*)
by *transfer* (*simp add*: *map-of-map-restrict*)

lemma *pcombine-alist* [*code*]:
pcombine f (*pfun-of-alist xs*) (*pfun-of-alist ys*) =
ptabulate (*remdups* (*map fst xs* • *map fst ys*))
(λx. *the* (*combine-options f* (*map-of xs x*) (*map-of ys x*)))
apply *transfer*
apply (*rule ext*)
apply (*rule sym*)
subgoal for *f xs ys x*
apply (*cases map-of xs x*; *cases map-of ys x*; *simp*)
apply (*force simp*: *map-of-eq-None-iff combine-options-def option.the-def o-def image-iff*
dest: *map-of-SomeD split*: *option.splits*)+
done
done

lemma *pfun-comp-alist* [*code*]: *pfun-of-alist ys* $\circ_p$ *pfun-of-alist xs* = *pfun-of-alist* (*AList.compose xs ys*)
by (*transfer*, *simp add*: *compose-conv*′)

lemma *equal-pfun* [*code*]:
HOL.equal (*pfun-of-alist xs*) (*pfun-of-alist ys*) ⟷
(*let ks* = *map fst xs*; *ls* = *map fst ys*
in (∀ *l*∈*set ls*. *l* ∈ *set ks*) ∧ (∀ *k*∈*set ks*. *k* ∈ *set ls* ∧ *map-of xs k* = *map-of ys k*))
apply (*simp add*: *equal-pfun-def*, *transfer*, *safe*, *simp-all*)
apply (*metis map-of-eq-None-iff option.distinct*(*1*) *weak-map-of-SomeI*)
apply (*metis domI domIff map-of-eq-None-iff weak-map-of-SomeI*)
apply (*metis* (*no-types*, *lifting*) *image-iff map-of-eq-None-iff*)
done

lemma *set-inter-Collect*: *set xs* ∩ *Collect P* = *set* (*filter P xs*)
by (*auto*)

Partial abstractions can either be modelled finitely, as lists, or infinitely as total functions. We therefore allow both of these as possibilities. If an abstraction is over a finite set, then it is compiled to an associative list. Otherwise, it becomes an enriched total function via *pfun-entries*.

lemma *pabs-set* [*code*]: *pabs* (*set xs*) *P f* = *pfun-of-alist* (*map* (λ*k*. (*k*, *f k*)) (*filter P xs*))
by (*auto simp add*: *pfun-eq-iff apply-pfun-alist map-of-map prod.case-eq-if image-iff map-of-map-restrict*)

lemma *pabs-coset* [*code*]:
pabs (*List.coset A*) *P f* = *pfun-of-map* (λ *x*. *if x* ∈ *List.coset A* ∧ *P x then Some* (*f x*) *else None*)
by (*simp add*: *pabs-def*, *transfer*, *auto*)

lemma *pfun-app-of-map* [*code*]: *pfun-app* (*pfun-of-map f*) *x* = *the* (*f x*)
by (*simp add*: *domIff option.the-def*)

lemma *graph-pfun-set* [*code*]:
graph-pfun (*set xs*) = *pfun-of-alist* (*filter* (λ(*x*, *y*). *length* (*remdups* (*map snd* (*AList.restrict* {*x*} *xs*))) = *1*) *xs*)
by (*transfer*, *simp only*: *comp-def mk-functional-alist*)
(*metis graph-map-set mk-functional mk-functional-alist*)

lemma *pabs-basic-pfun-entries* [*code-unfold*]: (λ *x* · *f x*) = *pfun-entries* (*List.coset* []) *f*
by (*metis UNIV-coset pfun-entries-pabs*)

declare *pdom-pfun-entries* [*code*]

lemma *pfun-app-entries* [*code*]: *pfun-app* (*pfun-entries A f*) *x* = (*if x* ∈ *A then f x else undefined*)
by *auto*

Useful for optimising relational compositions containing partial functions

declare *rel-comp-pfun* [*code-unfold*]

Fusing associative lists

fun *pfuse-alist* :: $('k \times 'a)$ *list* $\Rightarrow$ $('k \rightarrow\!\!\!\!\!\!+\ 'b) \Rightarrow ('k \times ('a \times 'b))$ *list* **where**
pfuse-alist [] *f* = [] |
pfuse-alist ((*k*, *v*) # *ps*) *f* =
(*if k* ∈ *pdom f then* (*k*, (*v*, *pfun-app f k*)) # *pfuse-alist ps f else pfuse-alist ps f*)

lemma *pfuse-pfun-of-alist-aux*:
pfuse (*pfun-of-alist xs*) *g* = *pfun-of-alist* (*pfuse-alist xs g*)
apply (*induct xs*)
apply (*simp-all add*: *pfuse-upd*, *safe*, *simp-all*)
apply (*metis* (*no-types*, *lifting*) *disjoint-iff-not-equal pdom-nres-disjoint pfun-upd-ext pfun-upd-twice pfuse-upd singletonD*)
done

lemma *pfuse-pfun-of-alist* [*code*]:
pfuse (*pfun-of-alist xs*) *g* = *pfun-of-alist* (*pfuse-alist* (*AList.clearjunk xs*) *g*)
by (*metis pfun-of-alist-clearjunk pfuse-pfun-of-alist-aux*)

14.22 Notation

bundle *Z-Pfun-Notation*
begin

no-notation *Stream.stream.SCons* (**infixr** ‹##› *65*)

no-notation *funcset* (**infixr** → *60*)

notation *pfun-tfun* (**infixr** → *60*)
notation *pfun-pfun* (**infixr** ⇸ *60*)
notation *pfun-ffun* (**infixr** ⇻ *60*)
notation *pfun-pinj* (**infixr** ↣ *60*)
notation *pfun-finj* (**infixr** ⇿ *60*)
notation *pfun-psurj* (**infixr** ⤖ *60*)
notation *pfun-tinj* (**infixr** ↣ *60*)
notation *pfun-bij* (**infixr** ⤖ *60*)

notation *pdom-res* (**infixr** ◁ *86*)
notation *pdom-nres* (**infixr** ⋪ *86*)

notation *pran-res* (**infixl** ▷ *86*)
notation *pran-nres* (**infixl** ⋫ *86*)

notation *pempty* ({↦})

end

Hide implementation details for partial functions

lifting-update *pfun.lifting*

lifting-forget *pfun.lifting*

end

15 Partial Injections

theory *Partial-Inj*
  imports *Partial-Fun*
begin

typedef $('a, 'b)$ *pinj* $= \{f :: ('a, 'b)$ *pfun*. *pfun-inj f*$\}$
  morphisms *pfun-of-pinj pinj-of-pfun*
  by (*auto intro*: *pfun-inj-empty*)

lemma *pinj-eq-pfun*: $f = g \longleftrightarrow$ *pfun-of-pinj f* $=$ *pfun-of-pinj g*
  by (*simp add*: *pfun-of-pinj-inject*)

lemma *pfun-inj-pinj* [*simp*]: *pfun-inj* (*pfun-of-pinj f*)
  using *pfun-of-pinj* **by** *auto*

type-notation *pinj* (**infixr** $\rightarrowtail$ *1*)

setup-lifting *type-definition-pinj*

lift-definition *pinv* :: $'a \rightarrowtail 'b \Rightarrow 'b \rightarrowtail 'a$ **is** *pfun-inv*
  by (*simp add*: *pfun-inj-inv*)

unbundle *lattice-syntax*

instantiation *pinj* :: (*type*, *type*) *bot*
begin
  lift-definition *bot-pinj* :: $('a, 'b)$ *pinj* **is** $\bot$
    by *simp*
instance ..
end

abbreviation *pinj-empty* :: $('a, 'b)$ *pinj* $(\{\}_\varrho)$ **where** $\{\}_\varrho \equiv \bot$

lift-definition *pinj-app* :: $('a, 'b)$ *pinj* $\Rightarrow 'a \Rightarrow 'b$ $(\text{-}'(\text{-}')_\varrho$ [*999*,*0*] *999*)
is *pfun-app* .

Adding a maplet to a partial injection requires that we remove any other maplet that points to the value v, to preserve injectivity.

lift-definition *pinj-upd* :: $('a, 'b)$ *pinj* $\Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *pinj*
is λ *f k v*. *pfun-upd* $(f \rhd_p (- \{v\}))$ *k v*
  by (*simp add*: *pfun-inj-rres pfun-inj-upd*)

lift-definition *pidom* :: $'a \rightarrowtail 'b \Rightarrow 'a$ *set* **is** *pdom* .

lift-definition *piran* :: $'a \rightarrowtail 'b \Rightarrow 'b$ *set* **is** *pran* **.**

lift-definition *pinj-dres* :: $'a$ *set* $\Rightarrow$ $('a, 'b)$ *pinj* $\Rightarrow$ $('a, 'b)$ *pinj* (**infixr** $\lhd_\varrho$ *85*) **is** *pdom-res*
by (*simp add*: *pfun-inj-dres*)

lift-definition *pinj-rres* :: $('a, 'b)$ *pinj* $\Rightarrow$ $'b$ *set* $\Rightarrow$ $('a, 'b)$ *pinj* (**infixl** $\rhd_\varrho$ *86*) **is** *pran-res*
by (*simp add*: *pfun-inj-rres*)

lift-definition *pinj-comp* :: $'b \rightarrowtail 'c \Rightarrow 'a \rightarrowtail 'b \Rightarrow 'a \rightarrowtail 'c$ (**infixl** $\circ_\varrho$ *55*) **is** $(\circ_p)$
by (*simp add*: *pfun-inj-comp*)

syntax
-PinjUpd :: $[('a, 'b)$ *pinj*, *maplets*$] \Rightarrow ('a, 'b)$ *pinj* $(\text{-}'(\text{-}')_\varrho$ $[900,0]900)$
-Pinj :: *maplets* $\Rightarrow ('a, 'b)$ *pinj* $((1\{\text{-}\}_\varrho))$

translations
-PinjUpd m (*-Maplets xy ms*) == *-PinjUpd* (*-PinjUpd m xy*) *ms*
-PinjUpd m (*-maplet x y*) == *CONST pinj-upd m x y*
-Pinj ms => *-PinjUpd* (*CONST pempty*) *ms*
-Pinj (*-Maplets ms1 ms2*) <= *-PinjUpd* (*-Pinj ms1*) *ms2*
-Pinj ms <= *-PinjUpd* (*CONST pempty*) *ms*

lemma *pinj-app-upd* [*simp*]: $(f(k \mapsto v)_\varrho)(x)_\varrho = (\mathit{if}\ (k = x)\ \mathit{then}\ v\ \mathit{else}\ (f \rhd_\varrho (-\{v\}))\ (x)_\varrho)$
by (*transfer*, *simp*)

lemma *pinj-eq-iff*: $f = g \longleftrightarrow (\mathit{pidom}(f) = \mathit{pidom}(g) \wedge (\forall\ x{\in}\mathit{pidom}(f).\ f(x)_\varrho = g(x)_\varrho))$
by (*transfer*, *simp add*: *pfun-eq-iff*)

lemma *pinv-pempty* [*simp*]: *pinv* $\{\}_\varrho = \{\}_\varrho$
by (*transfer*, *simp*)

lemma *pinv-pinj-upd* [*simp*]: *pinv* $(f(x \mapsto y)_\varrho) = (\mathit{pinv}\ ((-\{x\}) \lhd_\varrho f))(y \mapsto x)_\varrho$
by (*transfer*, *subst pfun-inv-upd*, *simp-all add*: *pfun-inj-dres pfun-inj-rres pfun-inv-rres pdres-rres-commute*, *simp add*: *pfun-inv-dres*)

lemma *pinv-pinv*: *pinv* (*pinv f*) = *f*
by (*transfer*, *simp add*: *pfun-inj-inv-inv*)

lemma *pinv-pcomp*: *pinv* $(f \circ_\varrho g) =$ *pinv g* $\circ_\varrho$ *pinv f*
by (*transfer*, *simp add*: *pfun-eq-graph pfun-graph-pfun-inv pfun-graph-comp pfun-inj-comp converse-relcomp*)

lemmas *pidom-empty* [*simp*] = *pdom-zero*[*Transfer.transferred*]
lemma *piran-zero* [*simp*]: *piran* $\{\}_\varrho = \{\}$ **by** (*transfer*, *simp*)

lemmas *pinj-dres-empty* [*simp*] = *pdom-res-zero*[*Transfer.transferred*]
lemmas *pinj-rres-empty* [*simp*] = *pran-res-zero*[*Transfer.transferred*]

lemmas *pidom-res-empty* [*simp*] = *pdom-res-empty*[*Transfer.transferred*]
lemmas *piran-res-empty* [*simp*] = *pran-res-empty*[*Transfer.transferred*]

lemma *pidom-res-upd*: $A \lhd_\varrho f(k \mapsto v)_\varrho = (\mathit{if}\ k \in A\ \mathit{then}\ (A \lhd_\varrho f)(k \mapsto v)_\varrho\ \mathit{else}\ A \lhd_\varrho (f \rhd_\varrho (- \{v\})))$
by (*transfer*, *simp*, *metis pdom-res-swap*)

lemma *piran-res-upd*: $f(x \mapsto v)_\varrho \rhd_\varrho A = (\mathit{if}\ v \in A\ \mathit{then}\ (f \rhd_\varrho A)(x \mapsto v)_\varrho\ \mathit{else}\ ((- \{x\}) \lhd_\varrho f) \rhd_\varrho A)$
by (*transfer*, *simp add*: *inf.commute*)
(*metis* (*no-types*, *opaque-lifting*) *ComplI Compl-Un double-compl insert-absorb insert-is-Un pdom-res-swap pran-res-twice*)

lemma *pinj-upd-with-dres-rres*: $((-\{x\}) \lhd_\varrho f \rhd_\varrho (-\{y\}))(x \mapsto y)_\varrho = f(x \mapsto y)_\varrho$
by (*transfer*, *simp add*: *pdom-res-swap*)

lemma *pidres-twice*: $A \lhd_\varrho B \lhd_\varrho f = (A \cap B) \lhd_\varrho f$
by (*transfer*, *metis pdom-res-twice*)

lemma *pidres-commute*: $A \lhd_\varrho B \lhd_\varrho f = B \lhd_\varrho A \lhd_\varrho f$
by (*metis* (*no-types*, *opaque-lifting*) *inf-commute pidres-twice*)

lemma *pidres-rres-commute*: $A \lhd_\varrho (P \rhd_\varrho B) = (A \lhd_\varrho P) \rhd_\varrho B$
by (*transfer*, *simp*, *metis* (*mono-tags*, *opaque-lifting*) *pdres-rres-commute*)

lemma *pirres-twice*: $f \rhd_\varrho A \rhd_\varrho B = f \rhd_\varrho (A \cap B)$
by (*transfer*, *metis* (*no-types*, *opaque-lifting*) *pran-res-twice*)

lemma *pirres-commute*: $f \rhd_\varrho A \rhd_\varrho B = f \rhd_\varrho B \rhd_\varrho A$
by (*metis inf-commute pirres-twice*)

lemma *pidom-upd*: $\mathit{pidom}\ (f(k \mapsto v)_\varrho) = \mathit{insert}\ k\ (\mathit{pidom}\ (f \rhd_\varrho (- \{v\})))$
by (*transfer*, *simp*)

lemma *f-pinv-f-apply*: $x \in \mathit{pran}\ (\mathit{pfun\text{-}of\text{-}pinj}\ f) \Longrightarrow (\mathit{pfun\text{-}of\text{-}pinj}\ f)(\mathit{pfun\text{-}of\text{-}pinj}\ (\mathit{pinv}\ f)\ (x)_p)_p = x$
by (*transfer*, *simp add*: *f-pfun-inv-f-apply*)

fun *pinj-of-alist* :: $('a \times 'b)\ \mathit{list} \Rightarrow 'a \rightarrowtail 'b$ **where**
$\mathit{pinj\text{-}of\text{-}alist}\ [] = \{\}_\varrho\ |$
$\mathit{pinj\text{-}of\text{-}alist}\ (p \# ps) = (\mathit{pinj\text{-}of\text{-}alist}\ ps)(\mathit{fst}\ p \mapsto \mathit{snd}\ p)_\varrho$

lemma *pinj-empty-alist* [*code*]: $\{\}_\varrho = \mathit{pinj\text{-}of\text{-}alist}\ []$
by *simp*

lemma *pinj-upd-alist* [*code*]: $(pinj\text{-}of\text{-}alist\ xs)(k \mapsto v)_\varrho = pinj\text{-}of\text{-}alist\ ((k, v) \# xs)$
by *simp*

context begin

Injective associative lists

definition *ialist* :: $('a \times 'b)\ list \Rightarrow bool$ **where**
$ialist\ xs = (distinct\ (map\ fst\ xs) \wedge distinct\ (map\ snd\ xs))$

Remove pairs where either the key or value appeared in a previous pair

qualified fun *clearjunk* :: $('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$ **where**
$clearjunk\ [] = [] \mid$
$clearjunk\ (p\#ps) = p \# filter\ (\lambda\ (k', v').\ k' \neq fst\ p \wedge v' \neq snd\ p)\ (clearjunk\ ps)$

lemma *ialist-clearjunk*: $ialist\ (clearjunk\ xs)$
by (*induct xs rule*:*clearjunk.induct*, *auto simp add*: *ialist-def*, (*meson distinct-map-filter*)+)

lemma *ialist-clearjunk-fp*: $ialist\ xs \Longrightarrow clearjunk\ xs = xs$
by (*induct xs*, *auto simp add*: *ialist-def filter-id-conv rev-image-eqI*)

lemma *clearjunk-idem* [*simp*]: $clearjunk\ (clearjunk\ xs) = clearjunk\ xs$
using *ialist-clearjunk ialist-clearjunk-fp* **by** *blast*

lemma *pinj-of-alist-ndres*: $k \notin fst\ `\ set\ xs \Longrightarrow (-\{k\}) \lhd_\varrho (pinj\text{-}of\text{-}alist\ xs) = pinj\text{-}of\text{-}alist\ xs$
by (*induct xs*, *auto simp add*: *pidom-res-upd*)

lemma *pinj-of-alist-nrres*: $v \notin snd\ `\ set\ xs \Longrightarrow (pinj\text{-}of\text{-}alist\ xs) \rhd_\varrho (-\ \{v\}) = pinj\text{-}of\text{-}alist\ xs$
by (*induct xs*, *auto simp add*: *piran-res-upd*)

lemma *pidom-ialist*: $ialist\ xs \Longrightarrow pidom\ (pinj\text{-}of\text{-}alist\ xs) = set\ (map\ fst\ xs)$
by (*induct xs*, *auto simp add*: *ialist-def pidom-upd*)
(*metis* (*no-types*, *lifting*) *fst-conv image-eqI pinj-of-alist-nrres*)+

lemma *pinj-of-alist-filter-as-dres-rres*:
$ialist\ xs \Longrightarrow pinj\text{-}of\text{-}alist\ (filter\ (\lambda(k', v').\ k' \neq fst\ p \wedge v' \neq snd\ p)\ xs) = (-\{fst\ p\}) \lhd_\varrho pinj\text{-}of\text{-}alist\ xs \rhd_\varrho (-\{snd\ p\})$
by (*induct xs rule*: *pinj-of-alist.induct*)
(*auto simp add*: *ialist-def piran-res-upd pinj-of-alist-ndres pidom-res-upd*
,*metis* (*no-types*, *lifting*) *pinj-of-alist-nrres pirres-commute*)

lemma *pinj-of-alist-clearjunk*: $pinj\text{-}of\text{-}alist\ (clearjunk\ xs) = pinj\text{-}of\text{-}alist\ xs$
by (*induct xs rule*:*clearjunk.induct*, *simp add*: *pinj-eq-iff*)
(*simp add*: *ialist-clearjunk pinj-of-alist-filter-as-dres-rres pinj-upd-with-dres-rres*)

lemma *pinv-pinj-of-ialist*:
$ialist\ xs \Longrightarrow pinv\ (pinj\text{-}of\text{-}alist\ xs) = pinj\text{-}of\text{-}alist\ (map\ (\lambda\ (x, y).\ (y, x))\ xs)$

by (*induct xs rule*: *pinj-of-alist.induct*, *auto simp add*: *ialist-def simp add*: *pinj-of-alist-ndres*)

lemma *pfun-of-ialist*: *ialist xs* $\Longrightarrow$ *pfun-of-pinj* (*pinj-of-alist xs*) = *pfun-of-alist xs*
by (*induct xs rule*: *pinj-of-alist.induct*, *auto simp add*: *bot-pinj.rep-eq ialist-def pinj-upd.rep-eq*)
(*metis pinj-of-alist-nrres pinj-rres.rep-eq*)

declare *clearjunk.simps* [*simp del*]

end

lemma *pinv-pinj-of-alist* [*code*]: *pinv* (*pinj-of-alist xs*) = *pinj-of-alist* (*map* (λ (*x*, *y*). (*y*, *x*)) (*Partial-Inj.clearjunk xs*))
by (*metis ialist-clearjunk pinj-of-alist-clearjunk pinv-pinj-of-ialist*)

lemma *pfun-of-pinj-of-alist* [*code*]:
pfun-of-pinj (*pinj-of-alist xs*) = *pfun-of-alist* (*Partial-Inj.clearjunk xs*)
by (*metis ialist-clearjunk pfun-of-ialist pinj-of-alist-clearjunk*)

declare *pinj-of-alist.simps* [*simp del*]

end

16 Finite Functions

theory *Finite-Fun*
imports *Partial-Fun*
begin

16.1 Finite function type and operations

typedef ('*a*, '*b*) *ffun* = {*f* :: ('*a*, '*b*) *pfun*. *finite*(*pdom*(*f*))}
morphisms *pfun-of Abs-pfun*
using *infinite-imp-nonempty* **by** *auto*

type-notation *ffun* (**infixr** $\nrightarrow$ *1*)

setup-lifting *type-definition-ffun*

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

lift-definition *ffun-upd* :: '*a* $\nrightarrow$ '*b* $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ '*a* $\nrightarrow$ '*b* **is** *pfun-upd* **by** *simp*

lift-definition *fdom* :: '*a* $\nrightarrow$ '*b* $\Rightarrow$ '*a set* **is** *pdom* **.**

lift-definition *fran* :: '*a* $\nrightarrow$ '*b* $\Rightarrow$ '*b set* **is** *pran* **.**

lift-definition *ffun-comp* :: ('*b*, '*c*) *ffun* ⇒ '*a* ↠ '*b* ⇒ ('*a*, '*c*) *ffun* (**infixl** ∘$_f$ *55*)
is *pfun-comp* **by** *auto*

lift-definition *ffun-member* :: '*a* × '*b* ⇒ '*a* ↠ '*b* ⇒ *bool* (**infix** ∈$_f$ *50*) **is** (∈$_p$) **.**

lift-definition *fdom-res* :: '*a set* ⇒ '*a* ↠ '*b* ⇒ '*a* ↠ '*b* (**infixr** ◁$_f$ *85*)
is *pdom-res* **by** *simp*

abbreviation *fdom-nres* (**infixr** −◁$_f$ *85*) **where** *fdom-nres A P* ≡ (− *A*) ◁$_f$ *P*

lift-definition *fran-res* :: '*a* ↠ '*b* ⇒ '*b set* ⇒ '*a* ↠ '*b* (**infixl** ▷$_f$ *85*)
is *pran-res* **by** *simp*

abbreviation *fran-nres* (**infixr** ▷$_f$− *66*) **where** *fran-nres P A* ≡ *P* ▷$_f$ (− *A*)

lift-definition *ffun-graph* :: '*a* ↠ '*b* ⇒ ('*a* × '*b*) *set* **is** *pfun-graph* **.**

lift-definition *graph-ffun* :: ('*a* × '*b*) *set* ⇒ '*a* ↠ '*b* **is**
  λ *R*. *if* (*finite* (*Domain R*)) *then graph-pfun R else pempty*
  by (*simp add*: *finite-Domain*) (*meson pdom-graph-pfun rev-finite-subset*)

unbundle *lattice-syntax*

lift-definition *ffun-entries* :: '*a set* ⇒ ('*a* ⇒ '*b*) ⇒ '*a* ↠ '*b*
  is λ *A f*. *if* (*finite A*) *then pfun-entries A f else* ⊥ **by** *simp*

instantiation *ffun* :: (*type*, *type*) *bot*
begin
lift-definition *bot-ffun* :: '*a* ↠ '*b* **is** ⊥ **by** *simp*
instance ..
end

abbreviation *fempty* :: '*a* ↠ '*b* ({}$_f$)
where *fempty* ≡ ⊥

instantiation *ffun* :: (*type*, *type*) *oplus*
begin
lift-definition *oplus-ffun* :: '*a* ↠ '*b* ⇒ '*a* ↠ '*b* ⇒ '*a* ↠ '*b* **is** (⊕) **by** *simp*
instance ..
end

instantiation *ffun* :: (*type*, *type*) *minus*
begin
lift-definition *minus-ffun* :: '*a* ↠ '*b* ⇒ '*a* ↠ '*b* ⇒ '*a* ↠ '*b* **is** (−)
  by (*metis Dom-pfun-graph finite-Diff finite-Domain pdom-pfun-graph-finite pfun-graph-minus*)
instance ..
end

instantiation *ffun* :: (*type*, *type*) *inf*

begin
lift-definition *inf-ffun* :: $'a \nrightarrow 'b \Rightarrow 'a \nrightarrow 'b \Rightarrow 'a \nrightarrow 'b$ **is** *inf*
by (*meson finite-Int infinite-super pdom-inter*)
instance ..
end

abbreviation *ffun-inter* :: $'a \nrightarrow 'b \Rightarrow 'a \nrightarrow 'b \Rightarrow 'a \nrightarrow 'b$ (**infixl** $\cap_f$ *80*)
where *ffun-inter* $\equiv$ *inf*

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

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

abbreviation *ffun-subset-eq* :: $'a \nrightarrow 'b \Rightarrow 'a \nrightarrow 'b \Rightarrow$ *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*)

instantiation *ffun* :: (*type*, *type*) *size*
begin

definition *size-ffun* :: $'a \nrightarrow 'b \Rightarrow$ *nat* **where**
[*simp*]: *size-ffun f* = *card* (*fdom f*)

instance ..

end

syntax
-FfunUpd :: [$'a \nrightarrow 'b$, *maplets*] => $'a \nrightarrow 'b$ (*-*$'$(*-*$')_f$ [*900*,*0*]*900*)
-Ffun :: *maplets* => $'a \nrightarrow 'b$ (($1\{$*-*$\}_f$))

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*

16.2 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 *ffun-override-dist-comp*:
$(f \oplus g) \circ_f h = (f \circ_f h) \oplus (g \circ_f h)$
by (*transfer*, *simp add*: *pfun-override-dist-comp*)

lemma *ffun-minus-unit* [*simp*]:
fixes $f :: 'a \nrightarrow 'b$
shows $f - \bot = f$
by (*transfer*, *simp*)

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

lemma *ffun-minus-self* [*simp*]:
fixes $f :: 'a \nrightarrow 'b$
shows $f - f = \bot$
by (*transfer*, *simp*)

instantiation *ffun* :: (*type*, *type*) *override*
begin
lift-definition *compatible-ffun* :: $'a \nrightarrow 'b \Rightarrow 'a \nrightarrow 'b \Rightarrow$ *bool* **is** *compatible* **.**

instance
by (*intro-classes*; *transfer*, *simp-all add*: *compatible-sym override-assoc override-comm*)
(*transfer*, *simp add*: *override-compat-iff*)+
end

lemma *compatible-ffun-alt-def*: $R \mathbin{\#\#} S = ((fdom\ R) \lhd_f S = (fdom\ S) \lhd_f R)$
by (*transfer*, *simp add*: *compatible-pfun-def*)

lemma *ffun-indep-compat*: $fdom(f) \cap fdom(g) = \{\} \Longrightarrow f \mathbin{\#\#} g$
by (*transfer*, *simp add*: *pfun-indep-compat*)

lemma *ffun-override-commute*:
$fdom(f) \cap fdom(g) = \{\} \Longrightarrow f \oplus g = g \oplus f$
by (*meson ffun-indep-compat override-comm*)

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

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

lemma *ffun-override-pos*: $x \oplus y = \{\}_f \Longrightarrow x = \{\}_f$
by (*transfer*, *simp add*: *pfun-override-pos*)

lemma *ffun-le-override*: $fdom\ x \cap fdom\ y = \{\} \Longrightarrow x \leq x \oplus y$
by (*transfer*, *simp add*: *pfun-le-override*)

16.3 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-override*:
$(x, y) \in_f f \oplus g \longleftrightarrow ((x \notin fdom(g) \wedge (x, y) \in_f f) \vee (x, y) \in_f g)$
by (*transfer*, *simp add*: *pfun-member-override*)

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-ext* [*simp*]: $x \in fdom(f) \Longrightarrow f(x \mapsto f(x)_f)_f = f$
by (*transfer*, *simp*)

lemma *ffun-app-add* [*simp*]: $x \in fdom(g) \Longrightarrow (f \oplus g)(x)_f = g(x)_f$

by (*transfer*, *simp*)

lemma *ffun-upd-add* [*simp*]: $f \oplus g(x \mapsto v)_f = (f \oplus g)(x \mapsto v)_f$
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-add-left* [*simp*]: $x \notin fdom(g) \Longrightarrow f(x \mapsto v)_f \oplus g = (f \oplus g)(x \mapsto v)_f$
by (*transfer*, *simp*)

lemma *ffun-app-add'* [*simp*]: $[\![\ e \in fdom\ f;\ \ e \notin fdom\ g\ ]\!] \Longrightarrow (f \oplus g)(e)_f = f(e)_f$
by (*transfer*, *simp*)

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*]: $x \notin fdom\ g \Longrightarrow (f - g)(x)_f = f(x)_f$
by (*transfer*, *auto*)

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

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

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

lemma *fsubseteq-ffun-upd2* [*intro*]:
$[\![\ f \subseteq_f g;\ \ x \notin fdom(f)\ ]\!] \Longrightarrow f \subseteq_f g(x \mapsto v)_f$
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*:
$f \subseteq_f g \Longrightarrow fdom(f) \subseteq fdom(g)$
by (*transfer*, *auto simp add*: *psubseteq-dom-subset*)

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

lemma *fdom-res-apply* [*simp*]:
$x \in A \Longrightarrow (A \lhd_f f)(x)_f = f(x)_f$
by (*transfer*, *simp*)

16.4 Domain laws

lemma *fdom-finite* [*simp*]: *finite*(*fdom*(*f*))
by (*transfer*, *simp*)

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

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

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

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

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

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

lemma *ffun-fdom-antires-upd* [*simp*]:
$k \in A \Longrightarrow ((-\ A) \lhd_f m)(k \mapsto v)_f = ((-\ (A - \{k\})) \lhd_f m)(k \mapsto v)_f$
by (*transfer*, *simp*)

lemma *fdom-res-UNIV* [*simp*]: $UNIV \lhd_f f = f$
by (*transfer*, *simp*)

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

lemma *pdom-pfun-of* [*simp*]: $pdom\ (pfun\text{-}of\ f) = fdom\ f$
by (*transfer*, *simp*)

lemma *finite-pdom-ffun* [*simp*]: *finite* (*pdom* (*pfun-of f*))
by (*transfer*, *simp*)

16.5 Range laws

lemma *fran-zero* [*simp*]: *fran* $\bot = \{\}$
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*)

16.6 Domain restriction laws

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

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

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

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

lemma *ffun-split-domain*: $A \lhd_f f \oplus (- A) \lhd_f f = f$
by (*transfer*, *simp add*: *pfun-split-domain*)

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

16.8 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* $\bot = \{\}$
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*)

16.9 Conversions

lift-definition *list-ffun* :: *'a list* ⇒ *nat* ↠ *'a* **is**
list-pfun **by** *simp*

lemma *fdom-list-ffun* [*simp*]: *fdom* (*list-ffun xs*) = {*1..length xs*}
by (*transfer*, *auto*)

lemma *fran-list-ffun* [*simp*]: *fran* (*list-ffun xs*) = *set xs*
by (*transfer*, *simp*)

lemma *ffun-app-list-ffun*: ⟦ *0* < *i*; *i* < *length xs* ⟧ ⟹ (*list-ffun xs*)(*i*)$_f$ = *xs* ! (*i* − *1*)
by (*transfer*, *simp add*: *pfun-app-list-pfun*)

lemma *range-list-ffun*: *range list-ffun* = {*f*. ∃ *i*. *fdom*(*f*) = {*1..i*}}

by (*transfer*, *auto simp add*: *range-list-pfun*)

16.10 Finite Function Lens

definition *ffun-lens* :: $'a \Rightarrow ('b \Longrightarrow 'a \twoheadrightarrow 'b)$ **where**
[*lens-defs*]: *ffun-lens i* = (| *lens-get* = λ *s*. $s(i)_f$, *lens-put* = λ *s v*. $s(i \mapsto v)_f$ |)

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

lemma *ffun-lens-src*: $\mathcal{S}_{\mathit{ffun\text{-}lens}\ i} = \{f.\ i \in \mathit{fdom}(f)\}$
by (*auto simp add*: *lens-defs lens-source-def*, *metis ffun-upd-ext*)

16.11 Notation

bundle *Z-Ffun-Notation*
begin

no-notation *Stream.stream.SCons* (**infixr** ‹##› *65*)

no-notation *funcset* (**infixr** $\rightarrow$ *60*)

notation *fdom-res* (**infixr** $\lhd$ *86*)
notation *fdom-nres* (**infixr** $\notin$ *86*)

notation *fran-res* (**infixl** $\rhd$ *86*)
notation *fran-nres* (**infixl** $\ntriangleright$ *86*)

notation *fempty* ($\{\mapsto\}$)

syntax *-Ffun* :: *maplets* => *logic* ((*1*{-}))

end

Hide implementation details for finite functions

lifting-update *ffun.lifting*
lifting-forget *ffun.lifting*

end

16.12 Finite Injections

theory *Finite-Inj*
imports *Partial-Inj Finite-Fun*
begin

typedef ('*a*, '*b*) *finj* = $\{f :: 'a \rightarrowtail 'b.\ \mathit{finite}(\mathit{pidom}(f))\}$
morphisms *pinj-of-finj finj-of-pinj*
by (*metis CollectI infinite-imp-nonempty pidom-empty*)

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

type-notation *finj* (**infixr** ⤀ *1*)

lift-definition *ffun-of-finj* :: $'a$ ⤀ $'b \Rightarrow 'a$ ⇻ $'b$ **is** λ *x*. *pfun-of-pinj* (*pinj-of-finj* *x*)
  by (*transfer*, *simp*, *metis mem-Collect-eq pidom.rep-eq pinj-of-finj*)

Hide implementation details for finite functions and injections

lifting-update *ffun.lifting*
lifting-forget *ffun.lifting*

lifting-update *finj.lifting*
lifting-forget *finj.lifting*

end

17 Total functions

theory *Total-Fun*
  imports *Partial-Fun*
begin

17.1 Total function type and operations

It may seem a little strange to create this, given we already have *fun*, but it's necessary to implement Z's type hierarchy.

typedef ($'a$, $'b$) *tfun* = $\{f :: 'a \nrightarrow 'b.\ pdom(f) = UNIV\}$
  morphisms *pfun-of-tfun Abs-tfun*
  by (*meson mem-Collect-eq pdom-pfun-entries*)

type-notation *tfun* (**infixr** $\Rightarrow_t$ *0*)

setup-lifting *type-definition-tfun*

lift-definition *mk-tfun* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow_t 'b)$ **is**
λ *f*. *fun-pfun f* **by** *simp*

lemma *pfun-of-tfun-mk-tfun* [*simp*]: *pfun-of-tfun* (*mk-tfun f*) = *fun-pfun f*
  by (*transfer*, *simp*)

end

18 Bounded Lists

theory *Bounded-List*

imports *List-Extra HOL−Library.Numeral-Type*
begin

The term $CARD('n)$ retrieves the cardinality of a finite type $'n$. Examples include the types *1*, *2* and *3*.

typedef $('a, 'n::finite)$ *blist* = $\{xs :: 'a\ list.\ length\ xs \leq CARD('n)\}$
morphisms *list-of-blist blist-of-list*
proof
show $[] \in \{xs.\ length\ xs \leq CARD('n)\}$
by *simp*
qed

declare *list-of-blist-inverse* [*simp*]

syntax *-blist* :: *type* ⇒ *type* ⇒ *type* (*- blist*[-] [*100*, *0*] *100*)
translations (*type*) *'a blist*['*n*] == (*type*) (*'a*, *'n*) *blist*

We construct various functions using the lifting package to lift corresponding list functions.

setup-lifting *type-definition-blist*

lift-definition *blength* :: *'a blist*['*n*::*finite*] ⇒ *nat* **is** *length* **.**

lift-definition *bnth* :: *'a blist*['*n*::*finite*] ⇒ *nat* ⇒ *'a* **is** *nth* **.**

lift-definition *bappend* :: *'a blist*['*m*::*finite*] ⇒ *'a blist*['*n*::*finite*] ⇒ *'a blist*['*m* + '*n*] (**infixr** $\bullet_s$ *65*) **is** *append*
by *auto*

lift-definition *bmake* :: *'n itself* ⇒ *'a list* ⇒ *'a blist*['*n*::*finite*] **is** λ *-*. *take* $CARD('n)$
by *auto*

code-datatype *bmake*

lemma *bmake-length-card*:
blength (*bmake TYPE*('*n*::*finite*) *xs*) = (*if length xs* ≤ $CARD('n)$ *then length xs else* $CARD('n)$)
by (*simp add*: *blength-def bmake-def*, *auto simp add*: *blist-of-list-inverse*)

lemma *blist-always-bounded*:
length (*list-of-blist* (*bl*::*'a blist*['*n*::*finite*])) ≤ $CARD('n)$
using *list-of-blist* **by** *blast*

lemma *blist-remove-head*:
fixes *bl* :: *'a blist*['*n*::*finite*]
assumes *blength bl* > *0*
shows *blength* (*bmake TYPE*('*n*::*finite*) (*tl* (*list-of-blist* (*bl*::*'a blist*['*n*::*finite*])))) < *blength bl*

by (*metis One-nat-def Suc-pred assms blength.rep-eq bmake-length-card length-tl less-Suc-eq-le linear*)

This proof is performed by transfer

lemma *bappend-bmake* [*code*]:
bmake TYPE('a::finite) xs $\bullet_s$ *bmake TYPE('b::finite) ys*
= bmake TYPE('a + 'b) (take CARD('a) xs $\bullet$ *take CARD('b) ys)*
by (*transfer*, *simp*)

instantiation *blist* :: (*type*, *finite*) *equal*
begin

definition *equal-blist* :: *'a blist['b]* $\Rightarrow$ *'a blist['b]* $\Rightarrow$ *bool* **where**
equal-blist m1 m2 $\longleftrightarrow$ (*list-of-blist m1 = list-of-blist m2*)

instance proof
fix *x y* :: *'a blist['b]*
show *equal-class.equal x y* = (*x = y*)
by (*simp add*: *equal-blist-def*, *transfer*, *simp*)
qed

end

lemma *list-of-blist-code* [*code*]:
list-of-blist (*bmake TYPE('n::finite) xs*) = *take CARD('n) xs*
by (*transfer*, *simp*)+

definition *blists* :: *'n::finite itself* $\Rightarrow$ *'a list* $\Rightarrow$ (*'a blist['n]*) *list* **where**
blists n xs = map blist-of-list (*b-lists CARD('n) xs*)

lemma *n-blists-as-b-lists*:
fixes *n::'n::finite itself*
shows *map list-of-blist* (*blists n xs*) = *b-lists CARD('n) xs* (**is** *?lhs = ?rhs*)
proof −
have *?lhs = map* (*list-of-blist* $\circ$ (*blist-of-list* :: - $\Rightarrow$ - *blist['n]*)) (*b-lists CARD('n) xs*)
by (*simp add*: *blists-def*)
also have ... = *map id* (*b-lists CARD('n) xs*)
by (*rule map-cong*, *auto simp add*: *blist-of-list-inverse length-b-lists-elem*)
also have ... = *b-lists CARD('n) xs*
by *simp*
finally show *?thesis* **.**
qed

lemma *set-blists-enum-UNIV*: *set* (*blists TYPE('n::finite)* (*enum-class.enum* ::*'a::enum list*)) = *UNIV*
proof −
have $\bigwedge$*x*:: *'a blist['n]*. *list-of-blist x* $\in$ *set* (*b-lists CARD('n) enum-class.enum*)
proof (*rule in-blistsI*)

fix *x*::*'a blist*['*n*]
show *length* (*list-of-blist x*) ≤ *CARD*('*n*)
by (*simp add*: *blist-always-bounded*)
show *list-of-blist x* ∈ *lists* (*set enum-class.enum*)
by (*auto simp add*: *lists-eq-set enum-UNIV*)
qed
thus *?thesis*
by (*force simp add*: *blists-def image-iff*)
qed

lemma *distinct-blists*: *distinct xs* ⟹ *distinct* (*blists n xs*)
by (*metis distinct-b-lists distinct-map n-blists-as-b-lists*)

definition *all-blists* :: ((*'a*::*enum*) *blist*['*n*::*finite*] ⇒ *bool*) ⇒ *bool*
where
all-blists P ⟷ (∀ *xs* ∈ *set* (*blists TYPE*('*n*) *Enum.enum*). *P xs*)

definition *ex-blists* :: ((*'a* :: *enum*) *blist*['*n*::*finite*] ⇒ *bool*) ⇒ *bool*
where
ex-blists P ⟷ (∃ *xs* ∈ *set* (*blists TYPE*('*n*) *Enum.enum*). *P xs*)

instantiation *blist* :: (*enum*, *finite*) *enum*
begin

definition *enum-blist* :: (*'a blist*['*b*]) *list* **where**
enum-blist = *blists TYPE*('*b*) *Enum.enum*

definition *enum-all-blist* :: (*'a blist*['*b*] ⇒ *bool*) ⇒ *bool* **where**
enum-all-blist P = *all-blists P*

definition *enum-ex-blist* :: (*'a blist*['*b*] ⇒ *bool*) ⇒ *bool* **where**
enum-ex-blist P = *ex-blists P*

instance
by (*intro-classes*)
(*simp-all add*: *enum-blist-def set-blists-enum-UNIV distinct-blists enum-distinct enum-all-blist-def enum-ex-blist-def all-blists-def ex-blists-def*)

end

end

19 Tabulating terms

theory *Tabulate-Command*
imports *Main*
keywords *tabulate* :: *diag*
begin

The following little tool allows creation truth tables for predicates with a finite domain

definition *tabulate* :: ('*a*::*enum* ⇒ '*b*) ⇒ ('*a* × '*b*) *list* **where**
tabulate f = *map* (λ *x*. (*x*, *f x*)) (*rev Enum.enum*)

ML ‹
structure Tabulate-Command =
struct

fun space-out cwidth str =
str ^ *Library.replicate-string* ((*cwidth* − *length* (*Symbol.explode str*))) *Symbol.space*;

fun print-row cwidths row =
String.concat (*Library.separate* | (*map* (*fn* (*w*, *r*) => *space-out w r*) (*ListPair.zip* (*cwidths*, *row*))));

(∗ *Print an ASCII art table, given a heading and list of rows* ∗)
fun print-table heads rows =
let
val cols = *map* (*fn i* => *map* (*fn xs* => *nth xs i*) (*heads* :: *rows*)) (*0 upto length heads* − *1*)
val cwidths = *map* (*fn c* => *foldr1 Int.max* (*map length* (*map Symbol.explode c*))) *cols*
val llength = *foldr1* (*fn* (*c*, *x*) => (*c* + *4*) + *x*) *cwidths* + *2*
in Print-Mode.with-modes [] (*fn* () =>
Pretty.paragraph ([*Pretty.str* (*print-row cwidths heads*)
, *Pretty.fbrk*
, *Pretty.para* ((*Library.replicate-string llength* =))
, *Pretty.fbrk*]
• *Library.separate Pretty.fbrk* (*map* (*Pretty.str o print-row cwidths*) *rows*))) () *end*;

fun tabulate-cmd raw-t state =
let
open Syntax
open Pretty
open HOLogic
val ctx = *Toplevel.context-of state*;
(∗ *Parse the term we'd like to tabulate* ∗)
val t = *Syntax.read-term ctx raw-t*;
val ctx' = *Proof-Context.augment t ctx*;
(∗ *Extract the set of free variables from the term* ∗)
val xs = *Term.add-frees t* []
val xp = *foldr1 mk-prod* (*map Free* (*rev xs*))
val t' = *HOLogic.tupled-lambda xp t*
(∗ *Close the term, apply the tabulate function, and evaluate* ∗)
val ts' = *dest-list* (*Value-Command.value ctx* (*check-term ctx* (*const* •{*const-name tabulate*} $ *t'*)));
fun term-string t = *XML.content-of* (*YXML.parse-body* (*symbolic-string-of*

(*pretty-term ctx′ t*)));
val heads = (*map term-string* (*map Free* (*rev xs*)) • [*term-string t*])
val rows = (*map* ((*fn* (*x*, *y*) => (*map term-string* (*strip-tuple x*) • [*term-string y*])) *o dest-prod*) *ts′*);
in Pretty.writeln ((*print-table heads rows*)) *end*;

end

val - =
Outer-Syntax.command ***command-keyword*** ‹*tabulate*› *tabulate the values of a term*
(*Parse.term*
>> (*fn t* => *Toplevel.keep* (*Tabulate-Command.tabulate-cmd t*)))

›

end

20 Meta-theory for Relation Library

theory *Relation-Lib*
imports
Countable-Set-Extra Positive Infinity Enum-Type Record-Default-Instance Def-Const Relation-Extra Partial-Fun Partial-Inj Finite-Fun Finite-Inj Total-Fun List-Extra Bounded-List Tabulate-Command
begin

This theory marks the boundary between reusable library utilities and the creation of the Z notation. We avoid overriding any HOL syntax up until this point, but we do supply some optional bundles.

lemma *if-eqE* [*elim*!]: ⟦ (*if b then e else f*) = *v*; ⟦ *b*; *e* = *v* ⟧ ⟹ *P*; ⟦ ¬ *b*; *f* = *v* ⟧ ⟹ *P* ⟧ ⟹ *P*
by (*cases b*, *auto*)

bundle *Z-Type-Syntax*
begin

type-notation *bool* (𝔹)
type-notation *nat* (ℕ)
type-notation *int* (ℤ)
type-notation *rat* (ℚ)
type-notation *real* (ℝ)

type-notation *set* (ℙ - [*999*] *999*)

type-notation *tfun* (**infixr** → *0*)

notation *Pow* (ℙ)

notation *Fpow* ($\mathbb{F}$)

end

class *refine* =
  fixes *ref-by* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\sqsubseteq$ *50*)
  and *sref-by* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\sqsubset$ *50*)
  assumes *ref-by-order*: *class.preorder* ($\sqsubseteq$) ($\sqsubset$)

interpretation *ref-preorder*: *preorder* ($\sqsubseteq$) ($\sqsubset$)
  by (*fact ref-by-order*)

lemma *ref-by-trans* [*trans*]: $[\![P \sqsubseteq Q;\ Q \sqsubseteq R]\!] \Longrightarrow P \sqsubseteq R$
  using *ref-preorder.dual-order.trans* **by** *auto*

abbreviation (*input*) *refines* (**infix** $\sqsupseteq$ *50*) **where** $Q \sqsupseteq P \equiv P \sqsubseteq Q$
abbreviation (*input*) *srefines* (**infix** $\sqsupset$ *50*) **where** $Q \sqsupset P \equiv P \sqsubset Q$

instantiation *bool* :: *refine*
begin

definition *ref-by-bool* $P\ Q = (Q \longrightarrow P)$
definition *sref-by-bool* $P\ Q = (\neg\ Q \wedge P)$

instance by (*intro-classes*, *unfold-locales*, *auto simp add*: *ref-by-bool-def sref-by-bool-def*)

end

instantiation *fun* :: (*type*, *refine*) *refine*
begin

definition *ref-by-fun* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$ **where** *ref-by-fun f g* = ($\forall$ x. $f(x) \sqsubseteq g(x)$)
definition *sref-by-fun* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$ **where** *sref-by-fun f g* = (f $\sqsubseteq$ $g \wedge \neg\ (g \sqsubseteq f)$)

instance
  by (*intro-classes*, *unfold-locales*)
    (*auto simp add*: *ref-by-fun-def sref-by-fun-def ref-preorder.less-le-not-le intro*: *ref-preorder.order.trans*)
end

end

21 Set Toolkit

theory *Set-Toolkit*
  imports *HOL−Library.Multiset Relation-Lib*
begin

The majority of the Z set toolkit is implemented in the core libraries of HOL. We could prove all the axioms of ISO 13568 as theorems, but we omit this for now. The main thing we need is to map between finite sets and the normal set type.

declare [[*coercion-enabled*]]

unbundle *Z-Type-Syntax*

end

22 Relation Toolkit

theory *Relation-Toolkit*
  imports *Set-Toolkit Overriding*
begin

22.1 Conversions

The majority of the relation toolkit is also part of HOL. We just need to generalise some of the syntax.

declare [[*coercion rel-apply*]]
declare [[*coercion pfun-app*]]
declare [[*coercion pfun-of*]]

The following definition is semantically identical to *pfun-graph*, but is used to represent coercions with associated reasoning.

definition *rel-of-pfun* :: $'a \nrightarrow 'b \Rightarrow 'a \leftrightarrow 'b$ ($[\text{-}]_{\nrightarrow}$) **where**
[*code-unfold*]: *rel-of-pfun* = *pfun-graph*

declare [[*coercion rel-of-pfun*]]
declare [[*coercion pfun-of-pinj*]]

notation *pfun-of-pinj* ($[\text{-}]_{\rightarrowtail\!\!\!\!\!\!\!\!\nrightarrow}$)

22.2 First component projection

Z supports n-ary Cartesian products. We cannot support such structures directly in Isabelle/HOL, but instead add the projection notations for the first and second components. A homogeneous finite Cartesian product type also exists in the Multivariate Analysis package.

abbreviation (*input*) *first* ≡ *fst*
notation *first* (*-.1* [*999*] *999*)

22.3 Second component projection

abbreviation (*input*) *second* ≡ *snd*
notation *second* (*-.2* [*999*] *999*)

22.4 Maplet

22.5 Domain

hide-const (**open**) *dom*

consts *dom* :: *'f* ⇒ *'a set*

adhoc-overloading
dom ⇌ *Map.dom* **and**
dom ⇌ *Relation.Domain* **and**
dom ⇌ *Partial-Fun.pdom* **and**
dom ⇌ *Finite-Fun.fdom* **and**
dom ⇌ *Partial-Inj.pidom*

22.6 Range

hide-const (**open**) *ran*

consts *ran* :: *'f* ⇒ *'a set*

adhoc-overloading
ran ⇌ *Map.ran* **and**
ran ⇌ *Relation.Range* **and**
ran ⇌ *Partial-Fun.pran* **and**
ran ⇌ *Finite-Fun.fran* **and**
ran ⇌ *Partial-Inj.piran*

22.7 Identity relation

notation *Id-on* (*id*[-])

22.8 Relational composition

notation *relcomp* (**infixr ;** *75*)

22.9 Functional composition

Composition is probably the most difficult of the Z functions to implement correctly. Firstly, the notation (∘) is already defined for HOL functions, and we need to respect that in order to use the HOL library functions. Secondly, Z composition can be used to compose heterogeneous relations and functions, which is not easy to type infer. Consequently, we opt to use adhoc overloading here.

consts *zcomp* :: *'f* ⇒ *'g* ⇒ *'h*

adhoc-overloading
zcomp ⇌ *Fun.comp* **and**
zcomp ⇌ *pfun-comp* **and**

zcomp ⇌ *ffun-comp*

Once we overload *zcomp*, we need to at least have output syntax set up.

notation (**output**) *zcomp* (**infixl** ∘ *55*)

bundle *Z-Relation-Syntax*
begin

no-notation *Fun.comp* (**infixl** ∘ *55*)
notation *zcomp* (**infixl** ∘ *55*)

end

22.10 Domain restriction and subtraction

consts *dom-res* :: *'a set* ⇒ *'r* ⇒ *'r* (**infixr** ◁ *85*)

abbreviation *ndres* (**infixr** ⩤ *85*) **where** *ndres A P* ≡ *CONST dom-res* (− *A*) *P*

adhoc-overloading
dom-res ⇌ *rel-domres*
and *dom-res* ⇌ *pdom-res*
and *dom-res* ⇌ *fdom-res*
and *dom-res* ⇌ *pinj-dres*

syntax *-ndres* :: *logic* ⇒ *logic* ⇒ *logic*
translations *-ndres A P* == *CONST dom-res* (− *A*) *P*

22.11 Range restriction and subtraction

consts *ran-res* :: *'r* ⇒ *'a set* ⇒ *'r* (**infixl** ▷ *86*)

abbreviation *nrres* (**infixl** ⩥ *86*) **where** *nrres P A* ≡ *CONST ran-res P* (− *A*)

adhoc-overloading
ran-res ⇌ *rel-ranres*
and *ran-res* ⇌ *pran-res*
and *ran-res* ⇌ *fran-res*
and *ran-res* ⇌ *pinj-rres*

22.12 Relational inversion

notation *converse* ((-˘) [*1000*] *999*)

lemma *relational-inverse*: $r^{\sim} = \{(p.2, p.1) \mid p.\ p \in r\}$
by *auto*

22.13 Relational image

notation *Image* (-⦇-⦈ [*990*] *990*)

lemma *Image-eq*: *Image r a* = {*p.2* | *p*. *p* ∈ *r* ∧ *p.1* ∈ *a*}
by (*auto simp add*: *Image-def*)

22.14 Overriding

lemma *dom-override*: *dom* ((*Q* :: *'a* ↔ *'b*) ⊕ *R*) = (*dom Q*) ∪ (*dom R*)
by (*simp add*: *Un-Int-distrib2*)

lemma *override-Un*: *dom Q* ∩ *dom R* = {} ⟹ *Q* ⊕ *R* = *Q* ∪ *R*
by (*simp add*: *override-eq Int-commute rel-domres-compl-disj*)

22.15 Proof Support

The objective of these laws is to, as much as possible, convert relational constructions into functional ones. The benefit is better proof automation in the more type constrained setting.

lemma *rel-of-pfun-eq-iff* [*simp*]: [*f*]↠ = [*g*]↠ ⟷ *f* = *g*
by (*simp add*: *pfun-eq-graph rel-of-pfun-def*)

lemma *rel-of-pfun-le-iff* [*simp*]: [*f*]↠ ⊆ [*g*]↠ ⟷ *f* ≤ *g*
by (*simp add*: *pfun-graph-le-iff rel-of-pfun-def*)

lemma *rel-of-pfun-pabs*: [*pabs A P f*]↠ = {(*k*, *v*). *k* ∈ *A* ∧ *P k* ∧ *v* = *f k*}
by (*simp add*: *rel-of-pfun-def pfun-graph-pabs*)

lemma *rel-of-pfun-apply* [*simp*]: [*f*]↠ *x* = *f x*
by (*simp add*: *rel-of-pfun-def*)

lemma *rel-of-pfun-functional* [*simp*]: *functional* [*f*]↠
by (*simp add*: *rel-of-pfun-def*)

lemma *rel-of-pfun-override* [*simp*]: [*f*]↠ ⊕ [*g*]↠ = [*f* ⊕ *g*]↠
by (*simp add*: *pfun-graph-override rel-of-pfun-def*)

lemma *rel-of-pfun-comp* [*simp*]: [*f*]↠ *O* [*g*]↠ = [*g* ∘*p* *f*]↠
by (*simp add*: *pfun-graph-comp rel-of-pfun-def*)

lemma *pfun-comp-inv*: [*f* ∘*p* *g*]↠~ = [*f*]↠~ *O* [*g*]↠~
by (*metis converse-relcomp rel-of-pfun-comp*)

lemma *rel-of-pfun-dom* [*simp*]: *Domain* [*f*]↠ = *pdom f*
by (*simp add*: *Dom-pfun-graph rel-of-pfun-def*)

lemma *rel-of-pfun-ran* [*simp*]: *Range* [*f*]↠ = *pran f*
by (*simp add*: *Range-pfun-graph rel-of-pfun-def*)

lemma *rel-of-pfun-domres* [*simp*]: $A \lhd [f]_{\rightarrow} = [A \lhd f]_{\rightarrow}$
by (*simp add*: *pfun-graph-domres rel-of-pfun-def*)

lemma *rel-of-pfun-ranres* [*simp*]: $[f]_{\rightarrow} \rhd A = [f \rhd A]_{\rightarrow}$
by (*simp add*: *rel-of-pfun-def pfun-graph-rres*)

lemma *rel-of-pfun-image* [*simp*]: $[f]_{\rightarrow}\ (\!|\ A\ |\!) = \mathit{pfun\text{-}image}\ f\ A$
by (*simp add*: *Image-as-rel-domres*)

lemma *rel-of-pfun-member-iff* [*simp*]:
$(k, v) \in [f]_{\rightarrow} \longleftrightarrow (k \in \mathit{dom}\ f \wedge f\ k = v)$
by (*simp add*: *rel-of-pfun-def*)

lemma *rel-of-pinj-conv* [*simp*]: $[[f]_{\rightarrowtail}]_{\rightarrow}{}^{-1} = [[\mathit{pinv}\ f]_{\rightarrowtail}]_{\rightarrow}$
by (*simp add*: *pfun-graph-pfun-inv pinv.rep-eq rel-of-pfun-def*)

lemma *dom-pinv* [*simp*]: $\mathit{dom}\ [\mathit{pinv}\ f]_{\rightarrowtail} = \mathit{ran}\ [f]_{\rightarrowtail}$
by (*simp add*: *pinv.rep-eq*)

lemma *ran-pinv* [*simp*]: $\mathit{ran}\ [\mathit{pinv}\ f]_{\rightarrowtail} = \mathit{dom}\ [f]_{\rightarrowtail}$
by (*metis dom-pinv pinv-pinv*)

lemma *pfun-inj-rel-conv* [*simp*]: $\mathit{pfun\text{-}inj}\ f \Longrightarrow [f]_{\rightarrow}{}^{\sim} = [\mathit{pfun\text{-}inv}\ f]_{\rightarrow}$
by (*simp add*: *pfun-graph-pfun-inv rel-of-pfun-def*)

end

23 Function Toolkit

theory *Function-Toolkit*
imports *Relation-Toolkit*
begin

23.1 Partial Functions

lemma *partial-functions*: $X \rightarrow_p Y = \{f \in X \leftrightarrow Y.\ \forall\ p \in f.\ \forall\ q \in f.\ p.1 = q.1 \longrightarrow p.2 = q.2\}$
by (*auto simp add*: *rel-pfun-def single-valued-def*)

notation *size* (#- [*999*] *999*)

instantiation *set* :: (*type*) *size*
begin
definition [*simp*]: *size A* = *card A*
instance ..
end

instantiation *pfun* :: (*type*, *type*) *size*

begin

definition *size-pfun* :: $('a \nrightarrow 'b) \Rightarrow nat$ **where**
size-pfun f = *card* (*pfun-graph f*)

instance ..

end

lemma *size-finite-pfun*: *finite* (*pdom f*) $\Longrightarrow$ $\#f = \#(dom\ f)$
by (*simp add*: *card-pfun-graph size-pfun-def*)

lemma *card-pfun-empty* [*simp*]: $\#\{\}_p = 0$
by (*simp add*: *size-pfun-def pfun-graph-zero*)

lemma *card-pfun-update* [*simp*]: *finite* (*dom f*) $\Longrightarrow$ $\#(f(k \mapsto v)_p) = (if\ (k \in dom\ f)\ then\ \#f\ else\ \#f + 1)$
by (*auto simp add*: *size-pfun-def pfun-graph-update Dom-pfun-graph rel-domres-compl-disj*)
(*metis card-pfun-graph insert-absorb pdom-upd pfun-graph-update*)

23.2 Total Functions

One issue that emerges in this encoding is the treatment of total functions. In Z, a total function is a particular kind of partial function whose domain covers the type universe. In HOL, a total function is one of the basic types. Typically, one wishes to apply total functions, partial functions, and finite functions to values using the notation $f\ x$. In order to implement this, we need to coerce the given function f to a total function, since this is fundamental to HOL's application construct. However, that means that we can't also coerce a total function to a partial function, as expected by Z, since this would lead to a cycle. Consequently, we actually need to create a new "total function" type, different to the HOL one, to break the cycle. We therefore consider the HOL total function type to be meta-logical with respect to Z.

declare [[*coercion pfun-of-tfun*]]

23.3 Disjointness

consts
disjoint :: $'f \Rightarrow bool$

adhoc-overloading
disjoint $\rightleftharpoons$ *rel-disjoint* **and**
disjoint $\rightleftharpoons$ *pfun-disjoint* **and**
disjoint $\rightleftharpoons$ *list-disjoint*

23.4 Partitions

consts *partitions* :: $'f \Rightarrow 'a\ set \Rightarrow bool$ (**infix** *partitions 65*)

adhoc-overloading
partitions $\rightleftharpoons$ *rel-partitions* **and**
partitions $\rightleftharpoons$ *pfun-partitions* **and**
partitions $\rightleftharpoons$ *list-partitions*

end

24 Number Toolkit

theory *Number-Toolkit*
imports *Function-Toolkit*
begin

The numeric operators are all implemented in HOL ((+), (−), (∗), etc.), and there seems little to be gained by repackaging them. However, we do make some syntactic additions.

24.1 Successor

abbreviation (*input*) *succ* $n \equiv n + 1$

24.2 Integers

type-notation *int* ($\mathbb{Z}$)

24.3 Natural numbers

type-notation *nat* ($\mathbb{N}$)

24.4 Rational numbers

type-notation *rat* ($\mathbb{Q}$)

24.5 Real numbers

type-notation *real* ($\mathbb{R}$)

24.6 Strictly positive natural numbers

definition *Nats1* ($\mathbb{N}_1$) **where** $\mathbb{N}_1 = \{x \in \mathbb{N}.\ \neg\ x = 0\}$

24.7 Non-zero integers

definition *Ints1* ($\mathbb{Z}_1$) **where** $\mathbb{Z}_1 = \{x \in \mathbb{Z}.\ \neg\ x = 0\}$

end

25 Sequence Toolkit

theory *Sequence-Toolkit*
imports *Number-Toolkit*
begin

25.1 Conversion

We define a number of coercions for mapping a list to finite function.

abbreviation *rel-of-list* :: *'a list* ⇒ *nat* ↔ *'a* ($[\text{-}]_s$) **where**
rel-of-list xs ≡ $[$*list-pfun xs*$]_{\rightarrow\!\!\!\!\!+}$

abbreviation *seq-nth* ($\text{-}'(\text{-}')_s$ [*999*,*0*] *999*) **where**
seq-nth xs i ≡ *xs* ! (*i* − *1*)

declare [[*coercion list-ffun*]]
declare [[*coercion list-pfun*]]
declare [[*coercion rel-of-list*]]
declare [[*coercion seq-nth*]]

25.2 Number range

lemma *number-range*: {*i*..*j*} = {*k* :: ℤ. *i* ≤ *k* ∧ *k* ≤ *j*}
by (*auto*)

The number range from i to j is the set of all integers greater than or equal to i, which are also less than or equal to j.

25.3 Iteration

definition *iter* :: ℤ ⇒ (*'X* ↔ *'X*) ⇒ (*'X* ↔ *'X*) **where**
iter n R = (*if* (*n* ≥ *0*) *then R* ^^ (*nat n*) *else* (*R*~) ^^ (*nat n*))

lemma *iter-eqs*:
iter 0 r = *Id*
n ≥ *0* ⟹ *iter* (*n* + *1*) *r* = *r* ; (*iter n r*)
n < *0* ⟹ *iter* (*n* + *1*) *r* = *iter n* (*r*~)
by (*simp-all add*: *iter-def*, *metis Suc-nat-eq-nat-zadd1 add.commute relpow.simps*(*2*) *relpow-commute*)

25.4 Number of members of a set

lemma *size-rel-of-list*:
#*xs* = *length xs*
by *simp*

25.5 Minimum

Implemented by the function *Min*.

25.6 Maximum

Implemented by the function *Max*.

25.7 Finite sequences

definition $seq\ A = lists\ A$

lemma *seq-iff* [*simp*]: $xs \in seq\ A \longleftrightarrow set\ xs \subseteq A$
by (*simp add*: *in-lists-conv-set seq-def subset-code*(*1*))

lemma *seq-ffun-set*: $range\ list\text{-}ffun = \{f :: \mathbb{N} \twoheadrightarrow 'X.\ dom(f) = \{1..\#f\}\}$
by (*simp add*: *range-list-ffun*, *force*)

25.8 Non-empty finite sequences

definition $seq_1\ A = seq\ A - \{[]\}$

lemma seq_1*-iff* [*simp*]: $xs \in seq_1(A) \longleftrightarrow (xs \in seq\ A \wedge \#xs > 0)$
by (*simp add*: seq_1*-def*)

25.9 Injective sequences

definition $iseq\ A = seq\ A \cap Collect\ distinct$

lemma *iseq-iff* [*simp*]: $xs \in iseq(A) \longleftrightarrow (xs \in seq\ A \wedge distinct\ xs)$
by (*simp add*: *iseq-def*)

25.10 Bounded sequences

definition $bseq :: \mathbb{N} \Rightarrow 'a\ set \Rightarrow 'a\ list\ set\ (bseq[-])$ **where**
$bseq\ n\ A = bounded\text{-}lists\ n\ A$

25.11 Sequence brackets

Provided by the HOL list notation $[x, y, z]$.

25.12 Concatenation

Provided by the HOL concatenation operator ($\bullet$).

25.13 Reverse

Provided by the HOL function *rev*.

25.14 Head of a sequence

definition *head* :: *'a list ⇸ 'a* **where**
head = (*λ xs* :: *'a list* | *#xs > 0* · *hd xs*)

lemma *dom-head*: *dom head* = {*xs*. *#xs > 0*}
by (*simp add*: *head-def*)

lemma *head-app*: *#xs > 0 ⟹ head xs* = *hd xs*
by (*simp add*: *head-def*)

lemma *head-z-def*: *xs* ∈ *seq*$_1$(*A*) *⟹ head xs* = *xs 1*
by (*simp add*: *hd-conv-nth head-app seq*$_1$*-def*)

25.15 Last of a sequence

definition *slast* :: *'a list ⇸ 'a* **where**
slast = (*λ xs* :: *'a list* | *#xs > 0* · *List.last xs*)

lemma *dom-last*: *dom slast* = {*xs*. *#xs > 0*}
by (*simp add*: *slast-def*)

lemma *slast-app*: *#xs > 0 ⟹ slast xs* = *last xs*
by (*simp add*: *slast-def*)

lemma *slast-eq*: *#s > 0 ⟹ last s* = *s* (*#s*)
by (*simp add*: *slast-app last-conv-nth*)

25.16 Tail of a sequence

definition *tail* :: *'a list ⇸ 'a list* **where**
tail = (*λ xs* :: *'a list* | *#xs > 0* · *tl xs*)

lemma *dom-tail*: *dom tail* = {*xs*. *#xs > 0*}
by (*simp add*: *tail-def*)

lemma *tail-app*: *#xs > 0 ⟹ tail xs* = *tl xs*
by (*simp add*: *tail-def*)

25.17 Domain

definition *dom-seq* :: *'a list ⇒* **N** *set* **where**
[*simp*]: *dom-seq xs* = {*0..<#xs*}

adhoc-overloading *dom ⇌ dom-seq*

25.18 Range

definition *ran-seq* :: *'a list ⇒ 'a set* **where**
[*simp*]: *ran-seq xs* = *set xs*

adhoc-overloading *ran* $\rightleftharpoons$ *ran-seq*

25.19 Filter

notation *seq-filter* (**infix** $\upharpoonright$ *80*)

lemma *seq-filter-Nil*: $[] \upharpoonright V = []$ **by** *simp*

lemma *seq-filter-append*: $(s \bullet t) \upharpoonright V = (s \upharpoonright V) \bullet (t \upharpoonright V)$
by (*simp add*: *seq-filter-append*)

lemma *seq-filter-subset-iff*: *ran* $s \subseteq V \longleftrightarrow (s \upharpoonright V = s)$
by (*auto simp add*: *seq-filter-def subsetD*, *meson filter-id-conv*)

lemma *seq-filter-empty*: $s \upharpoonright \{\} = []$ **by** *simp*

lemma *seq-filter-size*: $\#(s \upharpoonright V) \leq \#s$
by (*simp add*: *seq-filter-def*)

lemma *seq-filter-twice*: $(s \upharpoonright V) \upharpoonright W = s \upharpoonright (V \cap W)$ **by** *simp*

25.20 Examples

lemma ([*1*,*2*,*3*] **;** $(\lambda\ x \cdot x + 1))\ 1 = 2$
by (*simp add*: *pfun-graph-comp*[*THEN sym*] *list-pfun-def pcomp-pabs*)

end

26 Pretty Notation for Z

theory *Z-Toolkit-Pretty*
imports *Relation-Toolkit Number-Toolkit*
abbrevs $+\!\!> = \nrightarrow$ **and** $+\!\!> = \twoheadrightarrow$ **and** $++\!\!> = \nrightarrow$
and $>\!\!-\!\!> = \rightarrowtail$ **and** $>\!\!-\!\!> = \twoheadrightarrow$ **and** $>\!\!+\!\!> = \rightarrowtail$ **and** $>\!\!++\!\!> = \rightarrowtail$
and $<\!| = \lhd$ **and** $<\!| = \lhd$ **and** $<\!| = \trianglelefteq$ **and** $<\!| = \langle\!|$
and $|\!> = \rhd$ **and** $|\!> = \rhd$ **and** $|\!> = \trianglerighteq$ **and** $|\!> = |\!\rangle$
and $|\!\text{`} = \upharpoonright$ **and** $|\!\text{`} = \restriction$ **and** $|\!\text{`} = \upharpoonright$
and $O+ = \oplus$
and $;; = \fatsemi$ **and** $;; = \fatsemi$
and $PP = \mathbb{P}$ **and** $FF = \mathbb{F}$
begin

declare [[*coercion-enabled*]]

Code generation set up

code-datatype *pfun-of-alist*

Allow partial functions to be written with braces

syntax
  -Pfun :: *maplets* => ('*a*, '*b*) *pfun* ((*1*{-}))

bundle *Z-Syntax*
begin

unbundle *Z-Type-Syntax*

notation *relcomp* (**infixr** ⨾ *75*)

end

Relation Function Syntax

bundle *Z-RFun-Syntax*
begin

unbundle *Z-Syntax*

no-notation *funcset* (**infixr** → *60*)

notation *rel-tfun* (**infixr** → *60*)
notation *rel-pfun* (**infixr** ⇸ *60*)
notation *rel-ffun* (**infixr** ⇻ *60*)

end

context
  includes *Z-RFun-Syntax*
begin

26.1 Examples

typ ℙ ℕ → ℕ
typ ℙ ℕ ⇸ 𝔹
term {}
term *P* ⨾ *Q*
term *A* ◁ *B* ⩤ (*P* :: ℙ(ℕ) ⇸ 𝔹)

term ℙ ℕ → ℕ
term ℕ ⇻ ℕ

end

end

27 Partial Function Command

theory *Partial-Function-Command*
  imports *Function-Toolkit*

keywords *zfun* :: *thy-decl-block* **and** *precondition postcondition*
begin

definition *pfun-spec* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \nrightarrow 'b$ **where**
pfun-spec P Q = $(\lambda\ x \mid P\ x \wedge (\exists\ y.\ Q\ x\ y) \cdot SOME\ y.\ Q\ x\ y)$

lemma *pfun-spec-app-eqI* [*intro*]: $[\![\ P\ x;\ \bigwedge y.\ Q\ x\ y \longleftrightarrow y = f\ x\ ]\!] \Longrightarrow (pfun\text{-}spec\ P\ Q)(x)_p = f\ x$
by (*simp add*: *pfun-spec-def*, *subst pabs-apply*, *auto*)

ML-file ‹*Partial-Function-Command.ML*›

ML ‹
val - =
Outer-Syntax.command •{*command-keyword zfun*} *define Z−like partial functions*
(*Zfun-Command.zfun-parser* >> (*Toplevel.theory o Zfun-Command.compile-zfun*));
›

end

28 Z Mathematical Toolkit Meta-Theory

theory *Z-Toolkit*
imports
Relation-Lib
Set-Toolkit
Relation-Toolkit
Function-Toolkit
Number-Toolkit
Sequence-Toolkit
Z-Toolkit-Pretty
Haskell-Show
Enum-Type
Partial-Function-Command
begin end

References

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