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 aformentioned 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*
⟨*proof*⟩

lemma *Fpow-as-Pow*: *finite A* ⟹ *Fpow A* = *Pow A*
⟨*proof*⟩

lemma *Fpow-set* [*code*]:
Fpow (*set* []) = {{}}
Fpow (*set* (*x* # *xs*)) = (*let A* = *Fpow* (*set xs*) *in A* ∪ *insert x* ‘ *A*)
⟨*proof*⟩

2.3 Folds

context *abel-semigroup*
begin

lemma *foldr-snoc*: *foldr* (∗) (*xs* • [*x*]) *k* = (*foldr* (∗) *xs k*) ∗ *x*
⟨*proof*⟩

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*⟩

lemma *nth-el-appendr*[*simp*]: *length xs* ≤ *i* ⟹ (*xs* • *ys*)⟨*i*⟩$_l$ = *ys*⟨*i* − *length xs*⟩$_l$
⟨*proof*⟩

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*⟩

2.5.2 Sorted Lists

lemma *sorted-last* [*simp*]: $[\![\ x \in set\ xs;\ \ sorted\ xs\ ]\!] \Longrightarrow x \leq last\ xs$
⟨*proof*⟩

lemma *sorted-prefix*:
assumes $xs \leq ys$ *sorted ys*
shows *sorted xs*
⟨*proof*⟩

lemma *sorted-map*: $[\![\ sorted\ xs;\ \ mono\ f\ ]\!] \Longrightarrow sorted\ (map\ f\ xs)$
⟨*proof*⟩

lemma *sorted-distinct* [*intro*]: $[\![\ sorted\ (xs);\ \ distinct(xs)\ ]\!] \Longrightarrow (\forall\ \ i{<}length\ xs - 1.\ xs!i < xs!(i + 1))$
⟨*proof*⟩

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*:
$sorted(xs{\bullet}ys) = (sorted\ xs \wedge sorted\ ys \wedge (xs \neq [] \wedge ys \neq [] \longrightarrow xs!(length\ xs - 1) \leq ys!0))$
⟨*proof*⟩

Is the given list a permutation of the given set?

definition *is-sorted-list-of-set* :: $('a{::}ord)\ set \Rightarrow 'a\ list \Rightarrow bool$ **where**
$is\text{-}sorted\text{-}list\text{-}of\text{-}set\ A\ xs = ((\forall\ \ i{<}length(xs) - 1.\ xs!i < xs!(i + 1)) \wedge set(xs) = A)$

lemma *sorted-is-sorted-list-of-set*:
assumes *is-sorted-list-of-set A xs*
shows *sorted*(*xs*) **and** *distinct*(*xs*)
⟨*proof*⟩

lemma *is-sorted-list-of-set-alt-def*:
$is\text{-}sorted\text{-}list\text{-}of\text{-}set\ A\ xs \longleftrightarrow sorted\ (xs) \wedge distinct\ (xs) \wedge set(xs) = A$
⟨*proof*⟩

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

lemma *is-sorted-list-of-set*:
$finite\ A \Longrightarrow is\text{-}sorted\text{-}list\text{-}of\text{-}set\ A\ (sorted\text{-}list\text{-}of\text{-}set\ A)$
$\langle proof \rangle$

lemma *sorted-list-of-set-other-def*:
$finite\ A \Longrightarrow sorted\text{-}list\text{-}of\text{-}set(A) = (THE\ xs.\ sorted(xs) \wedge distinct(xs) \wedge set\ xs = A)$
$\langle proof \rangle$

lemma *sorted-list-of-set-alt* [*simp*]:
$finite\ A \Longrightarrow sorted\text{-}list\text{-}of\text{-}set\text{-}alt(A) = sorted\text{-}list\text{-}of\text{-}set(A)$
$\langle proof \rangle$

Sorting lists according to a relation

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

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

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

lemma *is-sorted-list-of-set-by-mono*:
$[\![\ R \subseteq S;\ \ is\text{-}sorted\text{-}list\text{-}of\text{-}set\text{-}by\ R\ A\ xs\ ]\!] \Longrightarrow is\text{-}sorted\text{-}list\text{-}of\text{-}set\text{-}by\ S\ A\ xs$
$\langle proof \rangle$

lemma *lexord-mono'*:
$[\![\ (\bigwedge\ x\ y.\ f\ x\ y \longrightarrow g\ x\ y);\ \ (xs,\ ys) \in lexord\ \{(x,\ y).\ f\ x\ y\}\ ]\!] \Longrightarrow (xs,\ ys) \in lexord\ \{(x,\ y).\ g\ x\ y\}$
$\langle proof \rangle$

lemma *fin-set-lexord-mono* [*mono*]:
$(\bigwedge\ x\ y.\ f\ x\ y \longrightarrow g\ x\ y) \Longrightarrow (xs,\ ys) \in fin\text{-}set\text{-}lexord\ \{(x,\ y).\ f\ x\ y\} \longrightarrow (xs,\ ys) \in fin\text{-}set\text{-}lexord\ \{(x,\ y).\ g\ x\ y\}$
$\langle proof \rangle$

definition *distincts* :: $'a\ set \Rightarrow 'a\ list\ set$ **where**
$distincts\ A = \{xs \in lists\ A.\ distinct(xs)\}$

lemma *tl-element*:
$[\![\ x \in set\ xs;\ \ x \neq hd(xs)\ ]\!] \Longrightarrow x \in set(tl(xs))$
$\langle proof \rangle$

2.5.3 List Update

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

2.5.4 Drop While and Take While

lemma *dropWhile-sorted-le-above*:
  $[\![\ sorted\ xs;\ \ x \in set\ (dropWhile\ (\lambda\ x.\ x \leq n)\ xs)\ ]\!] \Longrightarrow x > n$
$\langle proof \rangle$

lemma *set-dropWhile-le*:
  $sorted\ xs \Longrightarrow set\ (dropWhile\ (\lambda\ x.\ x \leq n)\ xs) = \{x{\in}set\ xs.\ x > n\}$
$\langle proof \rangle$

lemma *set-takeWhile-less-sorted*:
  $[\![\ sorted\ I;\ \ x \in set\ I;\ \ x < n\ ]\!] \Longrightarrow x \in set\ (takeWhile\ (\lambda x.\ x < n)\ I)$
$\langle proof \rangle$

lemma *nth-le-takeWhile-ord*: $[\![\ sorted\ xs;\ \ i \geq length\ (takeWhile\ (\lambda\ x.\ x \leq n)\ xs);$
$i < length\ xs\ ]\!] \Longrightarrow n \leq xs\ !\ i$
$\langle proof \rangle$

lemma *length-takeWhile-less*:
  $[\![\ a \in set\ xs;\ \ \neg\ P\ a\ ]\!] \Longrightarrow length\ (takeWhile\ P\ xs) < length\ xs$
  $\langle proof \rangle$

lemma *nth-length-takeWhile-less*:
  $[\![\ sorted\ xs;\ \ distinct\ xs;\ \ (\exists\ \ a \in set\ xs.\ a \geq n)\ ]\!] \Longrightarrow xs\ !\ length\ (takeWhile\ (\lambda x.$
$x < n)\ xs) \geq n$
  $\langle proof \rangle$

2.5.5 Last and But Last

lemma *length-gt-zero-butlast-concat*:
  assumes $length\ ys > 0$
  shows $butlast\ (xs \bullet ys) = xs \bullet (butlast\ ys)$
  $\langle proof \rangle$

lemma *length-eq-zero-butlast-concat*:
  assumes $length\ ys = 0$
  shows $butlast\ (xs \bullet ys) = butlast\ xs$
  $\langle proof \rangle$

lemma *butlast-single-element*:
  shows $butlast\ [e] = []$
  $\langle proof \rangle$

lemma *last-single-element*:
  shows $last\ [e] = e$
  ⟨*proof*⟩

lemma *length-zero-last-concat*:
  assumes $length\ t = 0$
  shows $last\ (s \bullet t) = last\ s$
  ⟨*proof*⟩

lemma *length-gt-zero-last-concat*:
  assumes $length\ t > 0$
  shows $last\ (s \bullet t) = last\ t$
  ⟨*proof*⟩

2.5.6 Prefixes and Strict Prefixes

lemma *prefix-length-eq*:
  $[\![\ length\ xs = length\ ys;\ \ prefix\ xs\ ys\ ]\!] \Longrightarrow xs = ys$
  ⟨*proof*⟩

lemma *prefix-Cons-elim* [*elim*]:
  assumes $prefix\ (x\ \#\ xs)\ ys$
  obtains ys' **where** $ys = x\ \#\ ys'\ prefix\ xs\ ys'$
  ⟨*proof*⟩

lemma *prefix-map-inj*:
  $[\![\ inj\text{-}on\ f\ (set\ xs \cup set\ ys);\ \ prefix\ (map\ f\ xs)\ (map\ f\ ys)\ ]\!] \Longrightarrow$
   $prefix\ xs\ ys$
⟨*proof*⟩

lemma *prefix-map-inj-eq* [*simp*]:
  $inj\text{-}on\ f\ (set\ xs \cup set\ ys) \Longrightarrow$
   $prefix\ (map\ f\ xs)\ (map\ f\ ys) \longleftrightarrow prefix\ xs\ ys$
  ⟨*proof*⟩

lemma *strict-prefix-Cons-elim* [*elim*]:
  assumes $strict\text{-}prefix\ (x\ \#\ xs)\ ys$
  obtains ys' **where** $ys = x\ \#\ ys'\ strict\text{-}prefix\ xs\ ys'$
  ⟨*proof*⟩

lemma *strict-prefix-map-inj*:
  $[\![\ inj\text{-}on\ f\ (set\ xs \cup set\ ys);\ \ strict\text{-}prefix\ (map\ f\ xs)\ (map\ f\ ys)\ ]\!] \Longrightarrow$
   $strict\text{-}prefix\ xs\ ys$
  ⟨*proof*⟩

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

$\langle proof \rangle$

lemma *prefix-drop*:
$[\![$ *drop* (*length xs*) *ys* = *zs*; *prefix xs ys* $]\!]$
$\Longrightarrow$ *ys* = *xs* $\bullet$ *zs*
$\langle proof \rangle$

lemma *list-append-prefixD* [*dest*]: $x \bullet y \leq z \Longrightarrow x \leq z$
$\langle proof \rangle$

lemma *prefix-not-empty*:
assumes *strict-prefix xs ys* **and** $xs \neq []$
shows $ys \neq []$
$\langle proof \rangle$

lemma *prefix-not-empty-length-gt-zero*:
assumes *strict-prefix xs ys* **and** $xs \neq []$
shows *length ys* > *0*
$\langle proof \rangle$

lemma *butlast-prefix-suffix-not-empty*:
assumes *strict-prefix* (*butlast xs*) *ys*
shows $ys \neq []$
$\langle proof \rangle$

lemma *prefix-and-concat-prefix-is-concat-prefix*:
assumes *prefix s t prefix* ($e \bullet t$) *u*
shows *prefix* ($e \bullet s$) *u*
$\langle proof \rangle$

lemma *prefix-eq-exists*:
prefix s t $\longleftrightarrow$ ($\exists$ *xs* . *s* $\bullet$ *xs* = *t*)
$\langle proof \rangle$

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

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*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma *prefix-imp-length-lteq*:
  assumes *prefix s t*
  shows *length s* $\leq$ *length t*
  ⟨*proof*⟩

lemma *prefix-imp-length-not-gt*:
  assumes *prefix s t*
  shows ¬ *length t* < *length s*
  ⟨*proof*⟩

lemma *prefix-and-eq-length-imp-eq-list*:
  assumes *prefix s t* **and** *length t* = *length s*
  shows *s*=*t*
  ⟨*proof*⟩

lemma *butlast-prefix-imp-length-not-gt*:
  assumes *length s* > *0 strict-prefix* (*butlast s*) *t*
  shows ¬ (*length t* < *length s*)
  ⟨*proof*⟩

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*⟩

lemma *list-prefix-iff*:
  (*prefix xs ys* $\longleftrightarrow$ (*length xs* $\leq$ *length ys* $\wedge$ ($\forall$ *i*<*length xs*. *xs*!*i* = *ys*!*i*)))
  ⟨*proof*⟩

lemma *list-le-prefix-iff*:
  (*xs* $\leq$ *ys* $\longleftrightarrow$ (*length xs* $\leq$ *length ys* $\wedge$ ($\forall$ *i*<*length xs*. *xs*!*i* = *ys*!*i*)))
  ⟨*proof*⟩

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* [] = []
  ⟨*proof*⟩

lemma *gcp-append* [*simp*]: *gcp* (*xs* • *ys*) (*xs* • *zs*) = *xs* • *gcp ys zs*
  ⟨*proof*⟩

lemma *gcp-lb1*: *prefix* (*gcp xs ys*) *xs*
⟨*proof*⟩

lemma *gcp-lb2*: *prefix* (*gcp xs ys*) *ys*

⟨*proof*⟩

interpretation *prefix-semilattice*: *semilattice-inf gcp prefix strict-prefix*
⟨*proof*⟩

2.5.7 Lexicographic Order

lemma *lexord-append*:
assumes $(xs_1 \bullet ys_1,\ xs_2 \bullet ys_2) \in lexord\ R\ length(xs_1) = length(xs_2)$
shows $(xs_1,\ xs_2) \in lexord\ R \lor (xs_1 = xs_2 \land (ys_1,\ ys_2) \in lexord\ R)$
⟨*proof*⟩

lemma *strict-prefix-lexord-rel*:
$strict\text{-}prefix\ xs\ ys \Longrightarrow (xs,\ ys) \in lexord\ R$
⟨*proof*⟩

lemma *strict-prefix-lexord-left*:
assumes $trans\ R\ (xs,\ ys) \in lexord\ R\ strict\text{-}prefix\ xs'\ xs$
shows $(xs',\ ys) \in lexord\ R$
⟨*proof*⟩

lemma *prefix-lexord-right*:
assumes $trans\ R\ (xs,\ ys) \in lexord\ R\ strict\text{-}prefix\ ys\ ys'$
shows $(xs,\ ys') \in lexord\ R$
⟨*proof*⟩

lemma *lexord-eq-length*:
assumes $(xs,\ ys) \in lexord\ R\ length\ xs = length\ ys$
shows $\exists\ i.\ (xs!i,\ ys!i) \in R \land i < length\ xs \land (\forall\ j{<}i.\ xs!j = ys!j)$
⟨*proof*⟩

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 $(xs,\ ys) \in lexord\ R$
⟨*proof*⟩

2.6 Distributed Concatenation

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

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

lemma *dist-concat-left-empty* [*simp*]:
$\{\} \frown ys = \{\}$
⟨*proof*⟩

lemma *dist-concat-right-empty* [*simp*]:

$xs \frown \{\} = \{\}$
⟨*proof*⟩

lemma *dist-concat-insert* [*simp*]:
$insert\ l\ ls1 \frown ls2 = ((\bullet)\ l\ `\ (\ ls2)) \cup (ls1 \frown ls2)$
⟨*proof*⟩

2.7 List Domain and Range

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

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

2.8 Extracting List Elements

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

lemma *seq-extract-Nil* [*simp*]: $A \upharpoonleft_l [] = []$
⟨*proof*⟩

lemma *seq-extract-Cons*:
$A \upharpoonleft_l (x \# xs) = (if\ 0 \in A\ then\ [x]\ else\ []) \bullet \{j.\ Suc\ j \in A\} \upharpoonleft_l xs$
⟨*proof*⟩

lemma *seq-extract-empty* [*simp*]: $\{\} \upharpoonleft_l xs = []$
⟨*proof*⟩

lemma *seq-extract-ident* [*simp*]: $\{0..<length\ xs\} \upharpoonleft_l xs = xs$
⟨*proof*⟩

lemma *seq-extract-split*:
assumes $i \leq length\ xs$
shows $\{0..<i\} \upharpoonleft_l xs \bullet \{i..<length\ xs\} \upharpoonleft_l xs = xs$
⟨*proof*⟩

lemma *seq-extract-append*:
$A \upharpoonleft_l (xs \bullet ys) = (A \upharpoonleft_l xs) \bullet (\{j.\ j + length\ xs \in A\} \upharpoonleft_l ys)$
⟨*proof*⟩

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

lemma *seq-extract-out-of-range*:
$A \cap dom_l(xs) = \{\} \Longrightarrow A \upharpoonleft_l xs = []$
⟨*proof*⟩

lemma *seq-extract-length* [*simp*]:

$length\ (A \upharpoonleft_l xs) = card\ (A \cap dom_l(xs))$
⟨proof⟩

lemma *seq-extract-Cons-atLeastLessThan*:
assumes $m < n$
shows $\{m..<n\} \upharpoonleft_l (x \# xs) = (if\ (m = 0)\ then\ x \# (\{0..<n-1\} \upharpoonleft_l xs)\ else\ \{m-1..<n-1\} \upharpoonleft_l xs)$
⟨proof⟩

lemma *seq-extract-singleton*:
assumes $i < length\ xs$
shows $\{i\} \upharpoonleft_l xs = [xs\ !\ i]$
⟨proof⟩

lemma *seq-extract-as-map*:
assumes $m < n$ $n \le length\ xs$
shows $\{m..<n\} \upharpoonleft_l xs = map\ (nth\ xs)\ [m..<n]$
⟨proof⟩

lemma *seq-append-as-extract*:
$xs = ys \bullet zs \longleftrightarrow (\exists\ i \le length(xs).\ ys = \{0..<i\} \upharpoonleft_l xs \wedge zs = \{i..<length(xs)\} \upharpoonleft_l xs)$
⟨proof⟩

2.9 Filtering a list according to a set

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

lemma *seq-filter-Cons-in* [*simp*]:
$x \in cs \Longrightarrow (x \# xs) \restriction_l cs = x \# (xs \restriction_l cs)$
⟨proof⟩

lemma *seq-filter-Cons-out* [*simp*]:
$x \notin cs \Longrightarrow (x \# xs) \restriction_l cs = (xs \restriction_l cs)$
⟨proof⟩

lemma *seq-filter-Nil* [*simp*]: $[] \restriction_l A = []$
⟨proof⟩

lemma *seq-filter-empty* [*simp*]: $xs \restriction_l \{\} = []$
⟨proof⟩

lemma *seq-filter-append*: $(xs \bullet ys) \restriction_l A = (xs \restriction_l A) \bullet (ys \restriction_l A)$
⟨proof⟩

lemma *seq-filter-UNIV* [*simp*]: $xs \restriction_l UNIV = xs$
⟨proof⟩

lemma *seq-filter-twice* [*simp*]: $(xs \upharpoonright_l A) \upharpoonright_l B = xs \upharpoonright_l (A \cap B)$
⟨*proof*⟩

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 ⟨*proof*⟩
end

lemma *minus-cancel* [*simp*]: $xs - xs = []$
⟨*proof*⟩

lemma *append-minus* [*simp*]: $(xs \bullet ys) - xs = ys$
⟨*proof*⟩

lemma *minus-right-nil* [*simp*]: $xs - [] = xs$
⟨*proof*⟩

lemma *list-concat-minus-list-concat*: $(s \bullet t) - (s \bullet z) = t - z$
⟨*proof*⟩

lemma *length-minus-list*: $y \le x \Longrightarrow length(x - y) = length(x) - length(y)$
⟨*proof*⟩

lemma *map-list-minus*:
$xs \le ys \Longrightarrow map\ f\ (ys - xs) = map\ f\ ys - map\ f\ xs$
⟨*proof*⟩

lemma *list-minus-first-tl* [*simp*]:
$[x] \le xs \Longrightarrow (xs - [x]) = tl\ xs$
⟨*proof*⟩

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

lemma *prefix-concat-minus*:
assumes *prefix xs ys*
shows $xs \bullet (ys - xs) = ys$
⟨*proof*⟩

lemma *prefix-minus-concat*:
assumes *prefix s t*
shows $(t - s) \bullet z = (t \bullet z) - s$
⟨*proof*⟩

lemma *strict-prefix-minus-not-empty*:
  assumes *strict-prefix xs ys*
  shows $ys - xs \neq []$
  ⟨*proof*⟩

lemma *strict-prefix-diff-minus*:
  assumes *prefix xs ys* **and** $xs \neq ys$
  shows $(ys - xs) \neq []$
  ⟨*proof*⟩

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
  ⟨*proof*⟩

lemma *list-minus-butlast-eq-butlast-list*:
  assumes *length* $t =$ *length* s **and** *strict-prefix* (*butlast* s) t
  shows $t -$ (*butlast* s) $=$ [*last* t]
  ⟨*proof*⟩

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)
  ⟨*proof*⟩

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)) $\neq []$
  ⟨*proof*⟩

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)) $= []$
  ⟨*proof*⟩

lemma *concat-minus-list-concat-butlast-eq-list-minus-butlast*:
  assumes *prefix* (*butlast* u) s
  shows $(t \bullet s) - (t \bullet$ (*butlast* u)) $= s -$ (*butlast* u)
  ⟨*proof*⟩

lemma *tl-list-minus-butlast-eq-empty*:
  assumes *strict-prefix* (*butlast* s) t **and** *length* $s =$ *length* t
  shows *tl* ($t -$ (*butlast* s)) $= []$
  ⟨*proof*⟩

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

shows $length\ (tl\ (t-s)) = (length\ (t-s)) - 1$
⟨proof⟩

lemma *length-list-minus*:
assumes *strict-prefix s t*
shows $length(t - s) = length(t) - length(s)$
⟨proof⟩

lemma *length-minus-le*: $length\ (ys - xs) \le length\ ys$
⟨proof⟩

lemma *length-minus-less*: $\llbracket\ xs \le ys;\ \ xs \ne []\ \rrbracket \Longrightarrow length\ (ys - xs) < length\ ys$
⟨proof⟩

lemma *filter-minus* [*simp*]: $ys \le xs \Longrightarrow filter\ P\ (xs - ys) = filter\ P\ xs - filter\ P\ ys$
⟨proof⟩

2.11 Laws on *list-update*

lemma *list-update-0*: $length(xs) > 0 \Longrightarrow xs[0 := x] = x\ \#\ tl\ xs$
⟨proof⟩

lemma *tl-list-update*: $\llbracket\ length\ xs > 0;\ \ k > 0\ \rrbracket \Longrightarrow tl(xs[k := v]) = (tl\ xs)[k-1 := v]$
⟨proof⟩

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

lemma *take-prefix*: $m \le n \Longrightarrow take\ m\ xs \le take\ n\ xs$
⟨proof⟩

lemma *nths-atLeastAtMost-0-take*: $nths\ xs\ \{0..m\} = take\ (Suc\ m)\ xs$
⟨proof⟩

lemma *nths-atLeastLessThan-0-take*: $nths\ xs\ \{0..<m\} = take\ m\ xs$
⟨proof⟩

lemma *nths-atLeastAtMost-prefix*: $m \le n \Longrightarrow nths\ xs\ \{0..m\} \le nths\ xs\ \{0..n\}$
⟨proof⟩

lemma *sorted-nths-atLeastAtMost-0*: $\llbracket\ m \le n;\ \ sorted\ (nths\ xs\ \{0..n\})\ \rrbracket \Longrightarrow sorted\ (nths\ xs\ \{0..m\})$
⟨proof⟩

lemma *sorted-nths-atLeastLessThan-0*: $\llbracket\ m \le n;\ \ sorted\ (nths\ xs\ \{0..<n\})\ \rrbracket \Longrightarrow sorted\ (nths\ xs\ \{0..<m\})$
⟨proof⟩

lemma *list-augment-as-update*:

$k < \mathit{length}\ xs \Longrightarrow \mathit{list\text{-}augment}\ xs\ k\ x = \mathit{list\text{-}update}\ xs\ k\ x$
⟨proof⟩

lemma *nths-list-update-out*: $k \notin A \Longrightarrow \mathit{nths}\ (\mathit{list\text{-}update}\ xs\ k\ x)\ A = \mathit{nths}\ xs\ A$
⟨proof⟩

lemma *nths-list-augment-out*: $[\![\ k < \mathit{length}\ xs;\ \ k \notin A\ ]\!] \Longrightarrow \mathit{nths}\ (\mathit{list\text{-}augment}\ xs\ k\ x)\ A = \mathit{nths}\ xs\ A$
⟨proof⟩

lemma *nths-none*:
assumes $\forall\, i \in I.\ i \geq \mathit{length}\ xs$
shows $\mathit{nths}\ xs\ I = []$
⟨proof⟩

lemma *nths-uptoLessThan*:
$[\![\ m \leq n;\ \ n < \mathit{length}\ xs\ ]\!] \Longrightarrow \mathit{nths}\ xs\ \{m..n\} = xs\ !\ m\ \#\ \mathit{nths}\ xs\ \{\mathit{Suc}\ m..n\}$
⟨proof⟩

lemma *nths-upt-nth*: $[\![\ j < i;\ \ i < \mathit{length}\ xs\ ]\!] \Longrightarrow (\mathit{nths}\ xs\ \{0..{<}i\})\ !\ j = xs\ !\ j$
⟨proof⟩

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

lemma *nths-upt-le-length*:
$[\![\ m \leq n;\ \ \mathit{Suc}\ n \leq \mathit{length}\ xs\ ]\!] \Longrightarrow \mathit{length}\ (\mathit{nths}\ xs\ \{m..n\}) = \mathit{Suc}\ n - m$
⟨proof⟩

lemma *sl1*: $n > 0 \Longrightarrow \{j.\ \mathit{Suc}\ j \leq n\} = \{0..n{-}1\}$
⟨proof⟩

lemma *sl2*: $[\![\ 0 < m;\ \ m \leq n\ ]\!] \Longrightarrow \{j.\ m \leq \mathit{Suc}\ j \wedge \mathit{Suc}\ j \leq n\} = \{m{-}1..n{-}1\}$
⟨proof⟩

lemma *nths-upt-le-nth*: $[\![\ m \leq n;\ \ \mathit{Suc}\ n \leq \mathit{length}\ xs;\ \ i < \mathit{Suc}\ n - m\ ]\!]$
$\Longrightarrow (\mathit{nths}\ xs\ \{m..n\})\ !\ i = xs\ !\ (i + m)$
⟨proof⟩

lemma *nths-split-union*:
assumes $\bigwedge x\ y.\ x \in A \Longrightarrow y \in B \Longrightarrow x < y$
shows $\mathit{nths}\ l\ A \bullet \mathit{nths}\ l\ B = \mathit{nths}\ l\ (A \cup B)$
⟨proof⟩

corollary *nths-upt-le-append-split*:
$j \leq i \Longrightarrow \mathit{nths}\ xs\ \{0..{<}j\} \bullet \mathit{nths}\ xs\ \{j..i\} = \mathit{nths}\ xs\ \{0..i\}$
⟨proof⟩

lemma *nths-Cons-atLeastAtMost*: $n > m \Longrightarrow nths\ (x\ \#\ xs)\ \{m..n\} = (\mathit{if}\ m = 0\ \mathit{then}\ x\ \#\ nths\ xs\ \{0..n-1\}\ \mathit{else}\ nths\ xs\ \{m-1..n-1\})$
⟨*proof*⟩

lemma *nths-atLeastAtMost-eq-drop-take*: $nths\ xs\ \{m..n\} = drop\ m\ (take\ (n+1)\ xs)$
⟨*proof*⟩

lemma *drop-as-map*: $drop\ m\ xs = map\ (nth\ xs)\ [m..<length\ xs]$
⟨*proof*⟩

lemma *take-as-map*: $take\ m\ xs = map\ (nth\ xs)\ [0..<min\ m\ (length\ xs)]$
⟨*proof*⟩

lemma *nths-atLeastAtMost-as-map*: $nths\ xs\ \{m..n\} = map\ (\lambda\ i.\ xs\ !\ i)\ [m..<min\ (n+1)\ (length\ xs)]$
⟨*proof*⟩

lemma *nths-single*: $nths\ xs\ \{k\} = (\mathit{if}\ k < length\ xs\ \mathit{then}\ [xs\ !\ k]\ \mathit{else}\ [])$
⟨*proof*⟩

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*⟩

lemma *length-nths-atLeastAtMost* [*simp*]: $length\ (nths\ xs\ \{m..n\}) = min\ (Suc\ n)\ (length\ xs) - m$
⟨*proof*⟩

lemma *hd-nths-atLeastAtMost*: $[\![\ m < length\ xs;\ \ m \le n\ ]\!] \Longrightarrow hd\ (nths\ xs\ \{m..n\}) = xs\ !\ m$
⟨*proof*⟩

lemma *tl-nths-atLeastAtMost*: $tl\ (nths\ xs\ \{m..n\}) = nths\ xs\ \{Suc\ m..n\}$
⟨*proof*⟩

lemma *set-nths-atLeastAtMost*: $set\ (nths\ xs\ \{m..n\}) = \{xs!i\ |\ i.\ m \le i \wedge i \le n \wedge i < length\ xs\}$
⟨*proof*⟩

lemma *nths-atLeastAtMost-neq-Nil* [*simp*]: $[\![\ m \le n;\ \ length\ xs > m\ ]\!] \Longrightarrow nths\ xs\ \{m..n\} \neq []$
⟨*proof*⟩

lemma *nths-atLeastAtMost-head*: $[\![\ m \le n;\ \ m < length\ xs\ ]\!] \Longrightarrow nths\ xs\ \{m..n\} = xs\ !\ m\ \#\ (nths\ xs\ \{Suc\ m..n\})$
⟨*proof*⟩

lemma *sorted-hd-le-all*: $[\![xs \neq [];\ sorted\ xs;\ x \in set\ xs]\!] \Longrightarrow hd\ xs \leq x$
⟨*proof*⟩

2.13 List power

overloading
listpow ≡ *compow* :: $nat \Rightarrow 'a\ list \Rightarrow 'a\ list$
begin

fun *listpow* :: $nat \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$listpow\ 0\ xs = []$
$|\ listpow\ (Suc\ n)\ xs = xs \bullet listpow\ n\ xs$

end

lemma *listpow-Nil* [*simp*]: $[]\ \widehat{}\widehat{}\ n = []$
⟨*proof*⟩

lemma *listpow-Suc-right*: $xs\ \widehat{}\widehat{}\ Suc\ n = xs\ \widehat{}\widehat{}\ n \bullet xs$
⟨*proof*⟩

lemma *listpow-add*: $xs\ \widehat{}\widehat{}\ (m + n) = xs\ \widehat{}\widehat{}\ m \bullet xs\ \widehat{}\widehat{}\ n$
⟨*proof*⟩

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

lemma *list-lex-less-neq* [*simp*]: $x <_l y \Longrightarrow x \neq y$
⟨*proof*⟩

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

lemma *Nil-less-Cons* [*simp*]: $[] <_l a \# x$
⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: $a \# x <_l b \# y \longleftrightarrow a < b \lor a = b \land x <_l y$
⟨*proof*⟩

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* [] = [[]]
⟨*proof*⟩

lemma *length-b-lists-elem*: *ys* ∈ *set* (*b-lists n xs*) ⟹ *length ys* ≤ *n*
⟨*proof*⟩

lemma *b-lists-in-lists*: *ys* ∈ *set* (*b-lists n xs*) ⟹ *ys* ∈ *lists* (*set xs*)
⟨*proof*⟩

lemma *in-blistsI*:
assumes *length xs* ≤ *n xs* ∈ *lists* (*set A*)
shows *xs* ∈ *set* (*b-lists n A*)
⟨*proof*⟩

lemma *ex-list-nonempty-carrier*:
assumes *A* ≠ {}
obtains *xs* **where** *length xs* = *n set xs* ⊆ *A*
⟨*proof*⟩

lemma *n-lists-inj*:
assumes *xs* ≠ [] *List.n-lists m xs* = *List.n-lists n xs*
shows *m* = *n*
⟨*proof*⟩

lemma *distinct-b-lists*: *distinct xs* ⟹ *distinct* (*b-lists n xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* []
⟨*proof*⟩

lemma *list-disjoint-Cons* [*simp*]: *list-disjoint* (*A* # *Bs*) = ((∀ *B* ∈ *set Bs*. *A* ∩ *B* = {}) ∧ *list-disjoint Bs*)

$\langle proof \rangle$

2.17 Code Generation

lemma *set-singleton-iff*: $set\ xs = \{x\} \longleftrightarrow remdups\ xs = [x]$
$\langle proof \rangle$

lemma *list-singleton-iff*: $(\exists\ x.\ xs = [x]) \longleftrightarrow (length\ xs = 1)$
$\langle proof \rangle$

end

3 Infinite Sequences

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

typedef $'a\ infseq = UNIV :: (nat \Rightarrow 'a)\ set$
$\langle proof \rangle$

setup-lifting *type-definition-infseq*

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

lift-definition $nth\text{-}infseq :: 'a\ infseq \Rightarrow nat \Rightarrow 'a$ (**infixl** $!_s$ *100*)
is $\lambda\ f\ i.\ f\ i\ \langle proof \rangle$

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

lemma *sinit-len* [*simp*]:
$length\ (sinit\ i\ xs) = i$
$\langle proof \rangle$

lemma *sinit-0* [*simp*]: $sinit\ 0\ xs = []$
$\langle proof \rangle$

lemma *prefix-upt-0* [*intro*]:
$i \le j \Longrightarrow prefix\ [0..<i]\ [0..<j]$
$\langle proof \rangle$

lemma *sinit-prefix*:
$i \le j \Longrightarrow prefix\ (sinit\ i\ xs)\ (sinit\ j\ xs)$

$\langle proof \rangle$

lemma *sinit-strict-prefix*:
$i < j \Longrightarrow$ *strict-prefix* (*sinit* i *xs*) (*sinit* j *xs*)
$\langle proof \rangle$

lemma *nth-sinit*:
$i < n \Longrightarrow$ *sinit* n *xs* ! i = *xs* $!_s$ i
$\langle proof \rangle$

lemma *sinit-append-split*:
assumes $i < j$
shows *sinit* j *xs* = *sinit* i *xs* ● *ssubstr* i j *xs*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma *range-ext*:
assumes $\forall\, i$ :: *nat*. $\forall\, x \in \{0..<i\}$. $f(x) = g(x)$
shows $f = g$
$\langle proof \rangle$

lemma *sinit-ext*:
($\forall$ i. *sinit* i *xs* = *sinit* i *ys*) $\Longrightarrow$ *xs* = *ys*
$\langle proof \rangle$

definition *infseq-lexord* :: *'a rel* ⇒ (*'a infseq*) *rel* **where**
infseq-lexord R = {(*xs*, *ys*). ($\exists$ i. (*sinit* i *xs*, *sinit* i *ys*) ∈ *lexord* R)}

lemma *infseq-lexord-irreflexive*:
$\forall\, x.\ (x, x) \notin R \Longrightarrow$ (*xs*, *xs*) ∉ *infseq-lexord* R
$\langle proof \rangle$

lemma *infseq-lexord-irrefl*:
irrefl $R \Longrightarrow$ *irrefl* (*infseq-lexord* R)
$\langle proof \rangle$

lemma *infseq-lexord-transitive*:
assumes *trans* R

shows *trans* (*infseq-lexord R*)
⟨*proof*⟩

lemma *infseq-lexord-trans*:
$\llbracket$ (*xs*, *ys*) ∈ *infseq-lexord R*; (*ys*, *zs*) ∈ *infseq-lexord R*; *trans R* $\rrbracket \Longrightarrow$ (*xs*, *zs*) ∈ *infseq-lexord R*
⟨*proof*⟩

lemma *infseq-lexord-antisym*:
$\llbracket$ *asym R*; (*a*, *b*) ∈ *infseq-lexord R* $\rrbracket \Longrightarrow$ (*b*, *a*) ∉ *infseq-lexord R*
⟨*proof*⟩

lemma *infseq-lexord-asym*:
assumes *asym R*
shows *asym* (*infseq-lexord R*)
⟨*proof*⟩

lemma *infseq-lexord-total*:
assumes *total R*
shows *total* (*infseq-lexord R*)
⟨*proof*⟩

lemma *infseq-lexord-strict-linear-order*:
assumes *strict-linear-order R*
shows *strict-linear-order* (*infseq-lexord R*)
⟨*proof*⟩

lemma *infseq-lexord-linear*:
assumes (∀ *a b*. (*a*,*b*)∈ *R* ∨ *a* = *b* ∨ (*b*,*a*) ∈ *R*)
shows (*x*,*y*) ∈ *infseq-lexord R* ∨ *x* = *y* ∨ (*y*,*x*) ∈ *infseq-lexord R*
⟨*proof*⟩

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

definition *less-infseq* :: *'a infseq* ⇒ *'a infseq* ⇒ *bool* **where**
less-infseq xs ys ⟷ (*xs*, *ys*) ∈ *infseq-lexord* {(*xs*, *ys*). *xs* < *ys*}

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

instance ⟨*proof*⟩

end

instance *infseq* :: (*order*) *order*
⟨*proof*⟩

instance *infseq* :: (*linorder*) *linorder*

$\langle proof \rangle$

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*}
$\langle proof \rangle$

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

inductive *sorted-rel* :: *'a rel* $\Rightarrow$ *'a list* $\Rightarrow$ *bool* **where**
Nil-rel [*iff*]: *sorted-rel R* [] |
Cons-rel: $\forall$ *y* $\in$ *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* $\Rightarrow$ *'a set* $\Rightarrow$ *'a list* **where**
list-of-set R = *folding-on.F* (*insort-rel R*) []

lift-definition *infseq-inj* :: *'a infseq infseq* $\Rightarrow$ *'a infseq* **is**
λ *f i*. *f* (*fst* (*prod-decode i*)) (*snd* (*prod-decode i*)) $\langle proof \rangle$

lift-definition *infseq-proj* :: *'a infseq* $\Rightarrow$ *'a infseq infseq* **is**
λ *f i j*. *f* (*prod-encode* (*i*, *j*)) $\langle proof \rangle$

lemma *infseq-inj-inverse*: *infseq-proj* (*infseq-inj x*) = *x*
$\langle proof \rangle$

lemma *infseq-proj-inverse*: *infseq-inj* (*infseq-proj x*) = *x*
$\langle proof \rangle$

lemma *infseq-inj*: *inj infseq-inj*
$\langle proof \rangle$

lemma *infseq-inj-surj*: *bij infseq-inj*
$\langle proof \rangle$

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** $\in_c$ *50*)
notation *cUn* (**infixl** $\cup_c$ *65*)
notation *cInt* (**infixl** $\cap_c$ *70*)
notation *cDiff* (**infixl** $-_c$ *65*)
notation *cUnion* ($\bigcup_c$- [*900*] *900*)
notation *cimage* (**infixr** ‘$_c$ *90*)

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

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

4.2 Countable set functions

setup-lifting *type-definition-cset*

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

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

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

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

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

translations
  $\forall$ $x \in_c A$. *P* == *CONST cBall A* (%*x*. *P*)
  $\exists$ $x \in_c A$. *P* == *CONST cBex A* (%*x*. *P*)

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

lift-definition *cset-Coll* :: $'a$ *cset* $\Rightarrow$ ($'a \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *cset* **is** λ *A P*. $\{x \in A.\ P\ x\}$
  $\langle proof \rangle$

lemma *cset-Coll-equiv*: *cset-Coll A P* = *cset-Collect* (λ *x*. $x \in_c A \wedge P\ x$)
  $\langle proof \rangle$

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*{- $\in_c$/ -./ -}$_c$))
translations
  $\{x \in_c A.\ P\}_c$ => *CONST cset-Coll A* (λ *x*. *P*)

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

lemma *cset-CollI*: $[\![\ a \in_c A;\ \ P\ a\ ]\!] \Longrightarrow a \in_c \{x \in_c A.\ P\ x\}_c$
  ⟨*proof*⟩

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

lemma *cset-Collect-cong*: $(\bigwedge x.\ P\ x = Q\ x)$ ==> $\{x.\ P\ x\}_c = \{x.\ Q\ x\}_c$
  ⟨*proof*⟩

lift-definition *cset-set* :: $'a$ *list* ⇒ $'a$ *cset* **is** *set*
  ⟨*proof*⟩

lemma *countable-finite-power*:
  $countable(A) \Longrightarrow countable\ \{B.\ B \subseteq A \wedge finite(B)\}$
  ⟨*proof*⟩

lift-definition *cInter* :: $'a$ *cset cset* ⇒ $'a$ *cset* ($\bigcap_c$- [*900*] *900*)
  is $\lambda A.$ *if* $A = \{\}$ *then* $\{\}$ *else* $\bigcap\ A$
  ⟨*proof*⟩

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

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

definition *cset-mapM* :: $'a$ *option cset* ⇒ $'a$ *cset option* **where**

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

lemma *cset-mapM-Some-image* [*simp*]:
$cset\text{-}mapM\ (cimage\ Some\ A) = Some\ A$
$\langle proof \rangle$

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

lemma *the-Some-image* [*simp*]:
$the\ `\ Some\ `\ xs = xs$
$\langle proof \rangle$

lemma *CCollect-ext-Some* [*simp*]:
$CCollect\text{-}ext\ Some\ xs = CCollect\ xs$
$\langle proof \rangle$

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

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

lemma *cset-count-inj-seq*:
$inj\text{-}on\ (cset\text{-}count\ A)\ (rcset\ A)$
$\langle proof \rangle$

lemma *cset-count-infinite-bij*:
assumes $infinite\ (rcset\ A)$
shows $bij\text{-}betw\ (cset\text{-}count\ A)\ (rcset\ A)\ UNIV$
$\langle proof \rangle$

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

lemma *cset-seq-ran*: $ran\ (cset\text{-}seq\ A) = rcset(A)$
$\langle proof \rangle$

lemma *cset-seq-inj*: $inj\ cset\text{-}seq$
$\langle proof \rangle$

lift-definition $cset2infseq :: 'a\ cset \Rightarrow 'a\ infseq$
is $(\lambda\ A\ i.\ if\ (i \in cset\text{-}count\ A\ `\ rcset\ A)\ then\ inv\text{-}into\ (rcset\ A)\ (cset\text{-}count\ A)\ i$

else (*SOME* x. $x \in_c A$)) ⟨*proof*⟩

lemma *range-cset2infseq*:
$A \neq \{\}_c \Longrightarrow$ *range* (*Rep-infseq* (*cset2infseq* A)) = *rcset* A
⟨*proof*⟩

lemma *infinite-cset-count-surj*: *infinite* (*rcset* A) $\Longrightarrow$ *surj* (*cset-count* A)
⟨*proof*⟩

lemma *cset2infseq-inj*:
inj-on cset2infseq $\{A.\ A \neq \{\}_c\}$
⟨*proof*⟩

lift-definition *nat-infseq2set* :: *nat infseq* $\Rightarrow$ *nat set* **is**
λ f. *prod-encode* ‘ $\{(x, f\ x) \mid x.\ True\}$ ⟨*proof*⟩

lemma *inj-nat-infseq2set*: *inj nat-infseq2set*
⟨*proof*⟩

lift-definition *bit-infseq-of-nat-set* :: *nat set* $\Rightarrow$ *bool infseq*
is λ A i. $i \in A$ ⟨*proof*⟩

lemma *bit-infseq-of-nat-set-inj*: *inj bit-infseq-of-nat-set*
⟨*proof*⟩

lemma *bit-infseq-of-nat-cset-bij*: *bij bit-infseq-of-nat-set*
⟨*proof*⟩

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

definition *nat-set-cset-collapse* :: *nat set cset* $\Rightarrow$ *nat set* **where**
nat-set-cset-collapse = *inv bit-infseq-of-nat-set* $\circ$ *infseq-inj* $\circ$ *cset2infseq* $\circ$ (λ A. (*bit-infseq-of-nat-set* ‘$_c$ A))

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

lemma *inj-csingle*:
inj csingle
⟨*proof*⟩

lemma *range-csingle*:
range csingle $\subseteq \{A.\ A \neq \{\}_c\}$
⟨*proof*⟩

lift-definition *csets* :: $'a$ *set* $\Rightarrow$ $'a$ *cset set* **is**
λ A. $\{B.\ B \subseteq A \wedge$ *countable* $B\}$ ⟨*proof*⟩

lemma *csets-finite*: $\mathit{finite}\ A \Longrightarrow \mathit{finite}\ (\mathit{csets}\ A)$
$\langle \mathit{proof} \rangle$

lemma *csets-infinite*: $\mathit{infinite}\ A \Longrightarrow \mathit{infinite}\ (\mathit{csets}\ A)$
$\langle \mathit{proof} \rangle$

lemma *csets-UNIV*:
$\mathit{csets}\ (\mathit{UNIV} :: {}'a\ \mathit{set}) = (\mathit{UNIV} :: {}'a\ \mathit{cset}\ \mathit{set})$
$\langle \mathit{proof} \rangle$

lemma *infinite-nempty-cset*:
assumes $\mathit{infinite}\ (\mathit{UNIV} :: {}'a\ \mathit{set})$
shows $\mathit{infinite}\ (\{A.\ A \neq \{\}_c\} :: {}'a\ \mathit{cset}\ \mathit{set})$
$\langle \mathit{proof} \rangle$

lemma *nat-set-cset-partial-bij*:
obtains $f :: \mathit{nat}\ \mathit{set}\ \mathit{cset} \Rightarrow \mathit{nat}\ \mathit{set}$ **where** $\mathit{bij\text{-}betw}\ f\ \{A.\ A \neq \{\}_c\}\ \mathit{UNIV}$
$\langle \mathit{proof} \rangle$

lemma *nat-set-cset-bij*:
obtains $f :: \mathit{nat}\ \mathit{set}\ \mathit{cset} \Rightarrow \mathit{nat}\ \mathit{set}$ **where** $\mathit{bij}\ f$
$\langle \mathit{proof} \rangle$

definition $\mathit{nat\text{-}set\text{-}cset\text{-}bij} = (\mathit{SOME}\ f :: \mathit{nat}\ \mathit{set}\ \mathit{cset} \Rightarrow \mathit{nat}\ \mathit{set}.\ \mathit{bij}\ f)$

lemma *bij-nat-set-cset-bij*:
$\mathit{bij}\ \mathit{nat\text{-}set\text{-}cset\text{-}bij}$
$\langle \mathit{proof} \rangle$

lemma *inj-on-image-csets*:
$\mathit{inj\text{-}on}\ f\ A \Longrightarrow \mathit{inj\text{-}on}\ ((`_c)\ f)\ (\mathit{csets}\ A)$
$\langle \mathit{proof} \rangle$

lemma *image-csets-surj*:
$[\![\ \mathit{inj\text{-}on}\ f\ A;\ \ f\ `\ A = B\ ]\!] \Longrightarrow (`_c)\ f\ `\ \mathit{csets}\ A = \mathit{csets}\ B$
$\langle \mathit{proof} \rangle$

lemma *bij-betw-image-csets*:
$\mathit{bij\text{-}betw}\ f\ A\ B \Longrightarrow \mathit{bij\text{-}betw}\ ((`_c)\ f)\ (\mathit{csets}\ A)\ (\mathit{csets}\ B)$
$\langle \mathit{proof} \rangle$
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*)
$\langle proof \rangle$

theorem *infinite-UNIV-fun1* [*simp*]:
infinite (*UNIV* :: *'a set*) $\Longrightarrow$
card (*UNIV* :: *'b set*) $\neq$ *Suc 0* $\Longrightarrow$
infinite (*UNIV* :: (*'a* $\Rightarrow$ *'b*) *set*)
$\langle proof \rangle$

theorem *infinite-UNIV-fun2* [*simp*]:
infinite (*UNIV* :: *'b set*) $\Longrightarrow$
infinite (*UNIV* :: (*'a* $\Rightarrow$ *'b*) *set*)
$\langle proof \rangle$

theorem *infinite-UNIV-set* [*simp*]:
infinite (*UNIV* :: *'a set*) $\Longrightarrow$
infinite (*UNIV* :: *'a set set*)
$\langle proof \rangle$

theorem *infinite-UNIV-prod1* [*simp*]:
infinite (*UNIV* :: *'a set*) $\Longrightarrow$
infinite (*UNIV* :: (*'a* $\times$ *'b*) *set*)
$\langle proof \rangle$

theorem *infinite-UNIV-prod2* [*simp*]:
infinite (*UNIV* :: *'b set*) $\Longrightarrow$
infinite (*UNIV* :: (*'a* $\times$ *'b*) *set*)
$\langle proof \rangle$

theorem *infinite-UNIV-sum1* [*simp*]:

infinite (*UNIV* :: *'a set*) $\Longrightarrow$
infinite (*UNIV* :: (*'a* + *'b*) *set*)
⟨*proof*⟩

theorem *infinite-UNIV-sum2* [*simp*]:
infinite (*UNIV* :: *'b set*) $\Longrightarrow$
infinite (*UNIV* :: (*'a* + *'b*) *set*)
⟨*proof*⟩

theorem *infinite-UNIV-list* [*simp*]:
infinite (*UNIV* :: *'a list set*)
⟨*proof*⟩

theorem *infinite-UNIV-option* [*simp*]:
infinite (*UNIV* :: *'a set*) $\Longrightarrow$
infinite (*UNIV* :: *'a option set*)
⟨*proof*⟩

theorem *infinite-image* [*intro*]:
infinite A $\Longrightarrow$ *inj-on f A* $\Longrightarrow$ *infinite* (*f* ‘ *A*)
⟨*proof*⟩

theorem *infinite-transfer* :
infinite B $\Longrightarrow$ $B \subseteq f$ ‘ A $\Longrightarrow$ *infinite A*
⟨*proof*⟩

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* ⟨*proof*⟩
instance *int* :: *infinite* ⟨*proof*⟩
instance *real* :: *infinite* ⟨*proof*⟩
instance *fun* :: (*type*, *infinite*) *infinite* ⟨*proof*⟩
instance *set* :: (*infinite*) *infinite* ⟨*proof*⟩
instance *prod* :: (*infinite*, *infinite*) *infinite* ⟨*proof*⟩
instance *sum* :: (*infinite*, *infinite*) *infinite* ⟨*proof*⟩
instance *list* :: (*type*) *infinite* ⟨*proof*⟩
instance *option* :: (*infinite*) *infinite* ⟨*proof*⟩

subclass (**in** *infinite*) *two* ⟨*proof*⟩

end

6 Positive Subtypes

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

6.1 Type Definition

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

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* ($n \geq 0$) *then n else 0* ⟨*proof*⟩

lift-definition *real-of-pos* :: *real pos* ⇒ *real* **is** *id* ⟨*proof*⟩

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

6.3 Instantiations

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

instantiation *pos* :: ({*zero*, *linorder*}) *linorder*
begin
  lift-definition *less-eq-pos* :: $'a$ *pos* ⇒ $'a$ *pos* ⇒ *bool*
    is (≤) :: $'a$ ⇒ $'a$ ⇒ *bool* ⟨*proof*⟩
  lift-definition *less-pos* :: $'a$ *pos* ⇒ $'a$ *pos* ⇒ *bool*
    is (<) :: $'a$ ⇒ $'a$ ⇒ *bool* ⟨*proof*⟩
  instance
    ⟨*proof*⟩
end

instance *pos* :: ({*zero*, *linorder*, *no-top*}) *no-top*
⟨*proof*⟩

instance *pos* :: ({*zero*, *linorder*, *no-top*}) *infinite*
⟨*proof*⟩

instantiation *pos* :: (*linordered-semidom*) *linordered-semidom*
begin
lift-definition *one-pos* :: *'a pos*
is *1* :: *'a* ⟨*proof*⟩
lift-definition *plus-pos* :: $'a\ pos \Rightarrow 'a\ pos \Rightarrow 'a\ pos$
is (+) ⟨*proof*⟩
lift-definition *minus-pos* :: $'a\ pos \Rightarrow 'a\ pos \Rightarrow 'a\ pos$
is $\lambda x\ y.\ if\ y \leq x\ then\ x - y\ else\ 0$
⟨*proof*⟩
lift-definition *times-pos* :: $'a\ pos \Rightarrow 'a\ pos \Rightarrow 'a\ pos$
is *times* ⟨*proof*⟩
instance
⟨*proof*⟩
end

instantiation *pos* :: (*linordered-field*) *semidom-divide*
begin
lift-definition *divide-pos* :: $'a\ pos \Rightarrow 'a\ pos \Rightarrow 'a\ pos$
is *divide* ⟨*proof*⟩
instance
⟨*proof*⟩
end

instantiation *pos* :: (*linordered-field*) *inverse*
begin
lift-definition *inverse-pos* :: $'a\ pos \Rightarrow 'a\ pos$
is *inverse* ⟨*proof*⟩
instance ⟨*proof*⟩
end

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

6.4 Theorems

lemma *mk-pos-zero* [*simp*]: *mk-pos 0 = 0*
⟨*proof*⟩

lemma *mk-pos-one* [*simp*]: *mk-pos 1 = 1*
⟨*proof*⟩

lemma *mk-pos-leq*:

$[\![\ 0 \leq x;\ \ x \leq y \ ]\!] \Longrightarrow$ *mk-pos* $x \leq$ *mk-pos* y
⟨*proof*⟩

lemma *mk-pos-less*:
$[\![\ 0 \leq x;\ \ x < y \ ]\!] \Longrightarrow$ *mk-pos* $x <$ *mk-pos* y
⟨*proof*⟩

lemma *real-of-pos* [*simp*]: $x \geq 0 \Longrightarrow$ *real-of-pos* (*mk-pos* x) $= x$
⟨*proof*⟩

lemma *mk-pos-real-of-pos* [*simp*]: *mk-pos* (*real-of-pos* x) $= x$
⟨*proof*⟩

6.5 Transfer to Reals

named-theorems *pos-transfer*

lemma *real-of-pos-0* [*pos-transfer*]:
real-of-pos $0 = 0$
⟨*proof*⟩

lemma *real-of-pos-1* [*pos-transfer*]:
real-of-pos $1 = 1$
⟨*proof*⟩

lemma *real-op-pos-plus* [*pos-transfer*]:
real-of-pos $(x + y) =$ *real-of-pos* $x +$ *real-of-pos* y
⟨*proof*⟩

lemma *real-op-pos-minus* [*pos-transfer*]:
$x \geq y \Longrightarrow$ *real-of-pos* $(x - y) =$ *real-of-pos* $x -$ *real-of-pos* y
⟨*proof*⟩

lemma *real-op-pos-mult* [*pos-transfer*]:
real-of-pos $(x * y) =$ *real-of-pos* $x *$ *real-of-pos* y
⟨*proof*⟩

lemma *real-op-pos-div* [*pos-transfer*]:
real-of-pos $(x \ / \ y) =$ *real-of-pos* $x \ / $ *real-of-pos* y
⟨*proof*⟩

lemma *real-of-pos-numeral* [*pos-transfer*]:
real-of-pos (*numeral* n) $=$ *numeral* n
⟨*proof*⟩

lemma *real-of-pos-eq-transfer* [*pos-transfer*]:
$x = y \longleftrightarrow$ *real-of-pos* $x =$ *real-of-pos* y
⟨*proof*⟩

lemma *real-of-pos-less-eq-transfer* [*pos-transfer*]:
  $x \le y \longleftrightarrow$ *real-of-pos* $x \le$ *real-of-pos* y
  ⟨*proof*⟩

lemma *real-of-pos-less-transfer* [*pos-transfer*]:
  $x < y \longleftrightarrow$ *real-of-pos* $x <$ *real-of-pos* y
  ⟨*proof*⟩

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* :: $'a \Rightarrow$ *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 ⟨*proof*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

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 ⟨*proof*⟩

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*
⟨*proof*⟩

instantiation *nat* :: *show*
begin

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

instance ⟨*proof*⟩

end

lemma *show-Nat*: *show* (*Nat x*) = *show* (*max 0 x*)
⟨*proof*⟩

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

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

instance ⟨*proof*⟩

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 ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

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*⟩

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*⟩

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.

$\langle ML \rangle$

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*: $x \;\#\#\; y \Longrightarrow y \;\#\#\; x$

unbundle *lattice-syntax*

class *override* = *oplus* + *bot* + *compatible* +
  assumes *compatible-zero* [*simp*]: $x \;\#\#\; \bot$
  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*: $P \;\#\#\; Q \Longrightarrow P \oplus Q = Q \oplus P$
  and *override-compat*: $\llbracket\, P \;\#\#\; Q;\; (P \oplus Q) \;\#\#\; R \,\rrbracket \Longrightarrow P \;\#\#\; R$
  and *override-compatI*: $\llbracket\, P \;\#\#\; Q;\; P \;\#\#\; R;\; Q \;\#\#\; R \,\rrbracket \Longrightarrow (P \oplus Q) \;\#\#\; R$
begin

lemma *override-rzero* [*simp*]: $P \oplus \bot = P$
  $\langle proof \rangle$

lemma *override-compat'*: $\llbracket\, P \;\#\#\; Q;\; (P \oplus Q) \;\#\#\; R \,\rrbracket \Longrightarrow Q \;\#\#\; R$
  $\langle proof \rangle$

lemma *override-compat-iff*: $P \;\#\#\; Q \Longrightarrow (P \oplus Q) \;\#\#\; R \longleftrightarrow (P \;\#\#\; R) \wedge (Q \;\#\#\; R)$
  $\langle proof \rangle$

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.

⟨*ML*⟩

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*]: $x \in [{'a}]_T$
  ⟨*proof*⟩

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** $\lhd_r$ *85*) **where**
rel-domres A R = {*p* ∈ *R*. *fst p* ∈ *A*}

lemma *rel-domres-math-def*: $A \lhd_r R = \{(k, v) \in R.\ k \in A\}$
$\langle proof \rangle$

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**
$rel\text{-}ranres\ R\ A = \{p \in R.\ snd\ p \in A\}$

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* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$
and *res* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ set$

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*]: $cmpt\text{-}prod\ R\ S = ((Domain\ R) \lhd_r S = (Domain\ S) \lhd_r R)$
definition *res-prod* :: $('a \leftrightarrow 'b) \Rightarrow ('a \leftrightarrow 'b) \Rightarrow 'a \leftrightarrow 'b$
where [*simp*]: $res\text{-}prod\ R\ S = ((-\ Domain\ S) \lhd_r R) \cup S$
instance $\langle proof \rangle$
end

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

definition *oplus-set* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
$oplus\text{-}set = res$

instance $\langle proof \rangle$

end

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

Relational update adds a new pair to a relation.

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

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

12.4 Domain laws

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

lemma *Domain-Preimage*: $Domain\ P = P^{-1}$ `` $UNIV$
⟨*proof*⟩

lemma *Domain-relcomp* [*simp*]: $Domain\ (P\ O\ Q) = Domain\ (P \rhd_r Domain\ Q)$
⟨*proof*⟩

lemma *Domain-relcomp-conv*: $Domain\ (P\ O\ Q) = (P^{-1}$ `` $Domain(Q))$
⟨*proof*⟩

lemma *Domain-set*: $Domain\ (set\ xs) = set\ (map\ fst\ xs)$
⟨*proof*⟩

12.5 Range laws

lemma *Range-Image*: $Range\ P = P$ `` $UNIV$
⟨*proof*⟩

lemma *Range-relcomp*: $Range\ (P\ O\ Q) = (Q$ `` $Range(P))$
⟨*proof*⟩

12.6 Domain Restriction

lemma *Domain-rel-domres* [*simp*]: $Domain\ (A \lhd_r R) = A \cap Domain(R)$
⟨*proof*⟩

lemma *rel-domres-empty* [*simp*]: $\{\} \lhd_r R = \{\}$
⟨*proof*⟩

lemma *rel-domres-UNIV* [*simp*]: $UNIV \lhd_r R = R$
⟨*proof*⟩

lemma *rel-domres-nil* [*simp*]: $A \lhd_r \{\} = \{\}$
⟨*proof*⟩

lemma *rel-domres-inter* [*simp*]: $A \lhd_r B \lhd_r R = (A \cap B) \lhd_r R$
⟨*proof*⟩

lemma *rel-domres-compl-disj*: $A \cap Domain\ P = \{\} \Longrightarrow (-\ A) \lhd_r P = P$
⟨*proof*⟩

lemma *rel-domres-notin-Dom*: $k \notin Domain(f) \Longrightarrow (-\ \{k\}) \lhd_r f = f$
⟨*proof*⟩

lemma *rel-domres-Id-on*: $A \lhd_r R = Id\text{-}on\ A\ O\ R$
⟨*proof*⟩

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* $A \lhd_r R)$
$\langle proof \rangle$

lemma *rel-domres-insert-set* [*simp*]: $x \notin$ *Domain* $P \Longrightarrow$ (*insert* x A) $\lhd_r P = A \lhd_r P$
$\langle proof \rangle$

lemma *Image-as-rel-domres*: R `` A = *Range* $(A \lhd_r R)$
$\langle proof \rangle$

lemma *rel-domres-Un* [*simp*]: $A \lhd_r (S \cup R) = (A \lhd_r S) \cup (A \lhd_r R)$
$\langle proof \rangle$

12.7 Range Restriction

lemma *rel-ranres-UNIV* [*simp*]: $P \rhd_r$ *UNIV* $= P$
$\langle proof \rangle$

lemma *rel-ranres-Un* [*simp*]: $(P \cup Q) \rhd_r A = (P \rhd_r A \cup Q \rhd_r A)$
$\langle proof \rangle$

lemma *rel-ranres-relcomp* [*simp*]: $(P\ O\ Q) \rhd_r A = P\ O\ (Q \rhd_r A)$
$\langle proof \rangle$

lemma *conv-rel-domres* [*simp*]: $(P \rhd_r A)^{-1} = A \lhd_r P^{-1}$
$\langle proof \rangle$

lemma *rel-ranres-le*: $A \subseteq B \Longrightarrow f \rhd_r A \leq f \rhd_r B$
$\langle proof \rangle$

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
$\langle proof \rangle$

instance *prod* :: (*type*, *type*) *restrict*
$\langle proof \rangle$

instantiation *set* :: (*restrict*) *override*

begin
definition *compatible-set* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ **where**
compatible-set = *cmpt*

instance
⟨*proof*⟩
end

lemma *override-eq*: $R \oplus S = ((- \ Domain\ S) \lhd_r R) \cup S$
⟨*proof*⟩

lemma *Domain-rel-override* [*simp*]: $Domain\ (R \oplus S) = Domain(R) \cup Domain(S)$
⟨*proof*⟩

lemma *Range-rel-override*: $Range(R \oplus S) \subseteq Range(R) \cup Range(S)$
⟨*proof*⟩

lemma *compatible-rel*: $R\ \#\#\ S = (Domain\ R \lhd_r S = Domain\ S \lhd_r R)$
⟨*proof*⟩

lemma *compatible-relI*: $Domain\ R \lhd_r S = Domain\ S \lhd_r R \Longrightarrow R\ \#\#\ S$
⟨*proof*⟩

12.9 Functional Relations

abbreviation *functional* :: $('a \leftrightarrow 'b) \Rightarrow bool$ **where**
functional $R \equiv$ *single-valued* R

lemma *functional-def*: *functional* $R \longleftrightarrow$ *inj-on fst* R
⟨*proof*⟩

lemma *functional-algebraic*: *functional* $R \longleftrightarrow R^{-1}\ O\ R \subseteq Id$
⟨*proof*⟩

lemma *functional-apply*:
assumes *functional* $R\ (x, y) \in R$
shows $R(x)_r = y$
⟨*proof*⟩

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

lemma *functional-elem*:
assumes *functional* $R\ x \in Domain(R)$
shows $(x, R(x)_r) \in R$
⟨*proof*⟩

lemma *functional-override* [*intro*!]: ⟦ *functional* R; *functional* S ⟧ $\Longrightarrow$ *functional*

$(R \oplus S)$
⟨*proof*⟩

lemma *functional-union* [*intro*!]: $[\![$ *functional* R; *functional* S; R ## S $]\!] \Longrightarrow$ *functional* $(R \cup S)$
⟨*proof*⟩

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

lemma *functional-insert* [*simp*]: *functional* (*insert* (x,y) g) $\longleftrightarrow$ (g‘‘$\{x\} \subseteq \{y\} \wedge$ *functional* g)
⟨*proof*⟩

lemma *functional-list-nil*[*simp*]: *functional-list* []
⟨*proof*⟩

lemma *functional-list*: *functional-list* xs $\longleftrightarrow$ *functional* (*set* xs)
⟨*proof*⟩

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

lemma *functional-fun-rel*: *functional* (*fun-rel* f)
⟨*proof*⟩

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

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

definition *mk-functional* :: $('a \leftrightarrow 'b) \Rightarrow ('a \leftrightarrow 'b)$ **where**
mk-functional $R = \{(x,\ y) \in R.\ \forall\ y'.\ (x,\ y') \in R \longrightarrow y = y'\}$

lemma *mk-functional* [*simp*]: *functional* (*mk-functional* R)
⟨*proof*⟩

lemma *mk-functional-fp*: *functional* $R \Longrightarrow$ *mk-functional* $R = R$
⟨*proof*⟩

lemma *mk-functional-idem*: *mk-functional* (*mk-functional* R) = *mk-functional* R
⟨*proof*⟩

lemma *mk-functional-subset* [*simp*]: *mk-functional* $R \subseteq R$
⟨*proof*⟩

lemma *Domain-mk-functional*: *Domain* (*mk-functional* R) $\subseteq$ *Domain* R
⟨*proof*⟩

definition *single-valued-dom* :: $('a \times 'b)$ *set* $\Rightarrow$ *'a set* **where**
single-valued-dom $R = \{x \in \mathit{Domain}(R).\ \exists\ y.\ R\ ``\ \{x\} = \{y\}\}$

lemma *mk-functional-single-valued-dom*: *mk-functional* R = *single-valued-dom* R $\lhd_r$ R
⟨*proof*⟩

12.10 Left-Total Relations

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

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

lemma *left-totalr-algebraic*: *left-totalr* $R \longleftrightarrow \mathit{Id} \subseteq R\ O\ R^{-1}$
⟨*proof*⟩

lemma *left-totalr-fun-rel*: *left-totalr* (*fun-rel* f)
⟨*proof*⟩

12.11 Injective Relations

definition *injective* :: $('a \leftrightarrow 'b) \Rightarrow$ *bool* **where**
injective R = (*functional* $R \wedge R\ O\ R^{-1} \subseteq \mathit{Id}$)

lemma *injectiveI*:
assumes *functional* R $\bigwedge x\ y.\ [\![\ x \in \mathit{Domain}\ R;\ \ y \in \mathit{Domain}\ R;\ \ R(x)_r = R(y)_r\ ]\!] \Longrightarrow x = y$
shows *injective* R
⟨*proof*⟩

12.12 Relation Sets

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

lemma *rel-typed-intro*: $[\![\ \mathit{Domain}(R) \subseteq A;\ \ \mathit{Range}(R) \subseteq B\ ]\!] \Longrightarrow R \in A \leftrightarrow B$
⟨*proof*⟩

definition *rel-pfun* :: *'a set* $\Rightarrow$ *'b set* $\Rightarrow$ $('a \leftrightarrow 'b)$ *set* (**infixr** $\rightarrow_p$ *55*) **where**
rel-pfun A $B = \{R.\ R \in A \leftrightarrow B \wedge \mathit{functional}\ R\}$ — Partial Functions

lemma *rel-pfun-intro*: $[\![\ R \in A \leftrightarrow B;\ \ \mathit{functional}\ R\ ]\!] \Longrightarrow R \in A \rightarrow_p B$
⟨*proof*⟩

definition *rel-tfun* :: *'a set* $\Rightarrow$ *'b set* $\Rightarrow$ $('a \leftrightarrow 'b)$ *set* (**infixr** $\rightarrow_t$ *55*) **where**
rel-tfun A $B = \{R.\ R \in A \rightarrow_p B \wedge \mathit{left\text{-}totalr}\ R\}$ — Total Functions

definition *rel-ffun* :: *'a set* $\Rightarrow$ *'b set* $\Rightarrow$ $('a \leftrightarrow 'b)$ *set* (**infixr** $\rightarrow_f$ *55*) **where**
rel-ffun A $B = \{R.\ R \in A \rightarrow_p B \wedge \mathit{finite}(\mathit{Domain}\ R)\}$ — 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*]: $R \in \textit{rel-ffun}\ A\ B \Longrightarrow R \in A \rightarrow_p B$
⟨*proof*⟩

lemma *rel-tfun-is-pfun* [*rclos*]: $R \in A \rightarrow_t B \Longrightarrow R \in A \rightarrow_p B$
⟨*proof*⟩

lemma *rel-pfun-is-typed* [*rclos*]: $R \in A \rightarrow_p B \Longrightarrow R \in A \leftrightarrow B$
⟨*proof*⟩

lemma *rel-ffun-empty* [*rclos*]: $\{\} \in \textit{rel-ffun}\ A\ B$
⟨*proof*⟩

lemma *rel-pfun-apply* [*rclos*]: $[\![\ x \in \textit{Domain}(R);\ \ R \in A \rightarrow_p B\ ]\!] \Longrightarrow R(x)_r \in B$
⟨*proof*⟩

lemma *rel-tfun-apply* [*rclos*]: $[\![\ x \in A;\ \ R \in A \rightarrow_t B\ ]\!] \Longrightarrow R(x)_r \in B$
⟨*proof*⟩

lemma *rel-typed-insert* [*rclos*]: $[\![\ R \in A \leftrightarrow B;\ \ x \in A;\ \ y \in B\ ]\!] \Longrightarrow \textit{insert}\ (x,\ y)\ R \in A \leftrightarrow B$
⟨*proof*⟩

lemma *rel-pfun-insert* [*rclos*]: $[\![\ R \in A \rightarrow_p B;\ \ x \in A;\ \ y \in B;\ \ x \notin \textit{Domain}(R)\ ]\!] \Longrightarrow \textit{insert}\ (x,\ y)\ R \in A \rightarrow_p B$
⟨*proof*⟩

lemma *rel-pfun-override* [*rclos*]: $[\![\ R \in A \rightarrow_p B;\ \ S \in A \rightarrow_p B\ ]\!] \Longrightarrow (R \oplus S) \in A \rightarrow_p B$
⟨*proof*⟩

12.14 Code Generation

lemma *rel-conv-alist* [*code*]: $(\textit{set}\ xs)^{-1} = \textit{set}\ (\textit{map}\ (\lambda(x,\ y).\ (y,\ x))\ xs)$
⟨*proof*⟩

lemma *rel-domres-alist* [*code*]: $A \lhd_r \textit{set}\ xs = \textit{set}\ (\textit{AList.restrict}\ A\ xs)$
⟨*proof*⟩

lemma *Image-alist* [*code*]: $\textit{set}\ xs\ ``\ A = \textit{set}\ (\textit{map}\ \textit{snd}\ (\textit{AList.restrict}\ A\ xs))$
⟨*proof*⟩

lemma *Collect-set*: $\{x \in \textit{set}\ xs.\ P\ x\} = \textit{set}\ (\textit{filter}\ P\ xs)$
⟨*proof*⟩

lemma *single-valued-dom-alist* [*code*]:
single-valued-dom (*set xs*) = *set* (*filter* (λx. *length* (*remdups* (*map snd* (*AList.restrict* $\{x\}$ *xs*))) = *1*) (*map fst xs*))
⟨*proof*⟩

lemma *AList-restrict-in-dom*: *AList.restrict* (*set* (*filter P* (*map fst xs*))) *xs* = *filter* (λ (x, y). P x) *xs*
⟨*proof*⟩

lemma *mk-functional-alist* [*code*]:
mk-functional (*set xs*) = *set* (*filter* (λ (x,y). *length* (*remdups* (*map snd* (*AList.restrict* $\{x\}$ *xs*))) = *1*) *xs*)
⟨*proof*⟩

lemma *rel-apply-set* [*code*]:
rel-apply (*set xs*) k =
(*let ys* = *filter* (λ (k', v). $k = k'$) *xs in*
if (*length ys* > *0* $\wedge$ *ys* = *replicate* (*length ys*) (*hd ys*)) *then snd* (*hd ys*) *else undefined*)
⟨*proof*⟩

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*: $f = g \longleftrightarrow (\forall\ x \in dom(f) \cup dom(g).\ f\ x = g\ x)$
⟨*proof*⟩

13.2 Graphing Maps

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

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

definition *graph-map'* :: ($'a \leftrightarrow 'b$) $\rightharpoonup$ ($'a \rightharpoonup 'b$) **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*
  $\langle proof \rangle$

lemma *map-graph-functional*[*simp*]: *functional* (*map-graph f*)
  $\langle proof \rangle$

lemma *map-graph-countable* [*simp*]: *countable* (*dom f*) $\Longrightarrow$ *countable* (*map-graph f*)
  $\langle proof \rangle$

lemma *map-graph-inv* [*simp*]: *graph-map* (*map-graph f*) = *f*
  $\langle proof \rangle$

lemma *graph-map-empty*[*simp*]: *graph-map* $\{\}$ = *Map.empty*
  $\langle proof \rangle$

lemma *graph-map-insert* [*simp*]: $[\![$*functional g*; $g``\{x\} \subseteq \{y\}]\!] \Longrightarrow$ *graph-map* (*insert* (x,y) *g*) = (*graph-map g*)$(x \mapsto y)$
  $\langle proof \rangle$

lemma *dom-map-graph*: *dom f* = *Domain*(*map-graph f*)
  $\langle proof \rangle$

lemma *ran-map-graph*: *ran f* = *Range*(*map-graph f*)
  $\langle proof \rangle$

lemma *graph-map-set*: *functional* (*set xs*) $\Longrightarrow$ *graph-map* (*set xs*) = *map-of xs*
  $\langle proof \rangle$

lemma *rel-apply-map-graph*:
  $x \in dom(f) \Longrightarrow (map\text{-}graph\ f)(x)_r = the\ (f\ x)$
  $\langle proof \rangle$

lemma *ran-map-add-subset*:
  ran $(x\ {+}{+}\ y) \subseteq$ (*ran x*) $\cup$ (*ran y*)
  $\langle proof \rangle$

lemma *finite-dom-graph*: *finite* (*dom f*) $\Longrightarrow$ *finite* (*map-graph f*)
  $\langle proof \rangle$

lemma *finite-dom-ran* [*simp*]: *finite* (*dom f*) $\Longrightarrow$ *finite* (*ran f*)
  $\langle proof \rangle$

lemma *functional-insert*: *functional* (*insert a R*) $\Longrightarrow$ *functional R*
  $\langle proof \rangle$

lemma *Domain-insert*: *Domain* (*insert a R*) = *insert* (*fst a*) (*Domain R*)
  $\langle proof \rangle$

lemma *card-map-graph*: $[\![$ *finite R*; *functional R* $]\!] \Longrightarrow$ *card R* $=$ *card* (*Domain R*)
⟨*proof*⟩

lemma *graph-map-inv* [*simp*]: *functional g* $\Longrightarrow$ *map-graph* (*graph-map g*) $=$ *g*
⟨*proof*⟩

lemma *graph-map-dom*: *dom* (*graph-map R*) $=$ *fst* ‘ *R*
⟨*proof*⟩

lemma *graph-map-countable-dom*: *countable R* $\Longrightarrow$ *countable* (*dom* (*graph-map R*))
⟨*proof*⟩

lemma *countable-ran*:
assumes *countable* (*dom f*)
shows *countable* (*ran f*)
⟨*proof*⟩

lemma *map-graph-inv'* [*simp*]:
graph-map' (*map-graph f*) $=$ *Some f*
⟨*proof*⟩

lemma *map-graph-inj*:
inj map-graph
⟨*proof*⟩

lemma *map-eq-graph*: *f* $=$ *g* $\longleftrightarrow$ *map-graph f* $=$ *map-graph g*
⟨*proof*⟩

lemma *map-le-graph*: *f* $\subseteq_m$ *g* $\longleftrightarrow$ *map-graph f* $\subseteq$ *map-graph g*
⟨*proof*⟩

lemma *map-graph-comp*: *map-graph* (*g* $\circ_m$ *f*) $=$ (*map-graph f*) *O* (*map-graph g*)
⟨*proof*⟩

lemma *rel-comp-map*: *R O map-graph f* $=$ (λ *p*. (*fst p*, *the* (*f* (*snd p*)))) ‘ (*R* $\rhd_r$ *dom*(*f*))
⟨*proof*⟩

lemma *map-graph-update*: *map-graph* (*f*(*k* $\mapsto$ *v*)) $=$ *insert* (*k*, *v*) (($-$ {*k*}) $\lhd_r$ *map-graph f*)
⟨*proof*⟩

13.3 Map Application

definition *map-apply* :: (*'a* $\rightharpoonup$ *'b*) $\Rightarrow$ *'a* $\Rightarrow$ *'b* (-'(-')$_m$ [*999*,*0*] *999*) **where**
map-apply $=$ (λ *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*:
$[\![\bigwedge x\, y.\ (x, y) \in_m A \longleftrightarrow (x, y) \in_m B]\!] \Longrightarrow A = B$
$\langle proof \rangle$

lemma *map-member-alt-def*:
$(x, y) \in_m A \longleftrightarrow (x \in dom\ A \wedge A(x)_m = y)$
$\langle proof \rangle$

lemma *map-le-member*:
$f \subseteq_m g \longleftrightarrow (\forall\ x\ y.\ (x,y) \in_m f \longrightarrow (x,y) \in_m g)$
$\langle proof \rangle$

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*: $preimage\ f\ (ran\ f) = dom\ f$
$\langle proof \rangle$

lemma *dom-preimage*: $dom\ (m \circ_m f) = preimage\ f\ (dom\ m)$
$\langle proof \rangle$

lemma *countable-preimage*:
assumes *countable A inj-on f (preimage f A)*
shows *countable (preimage f A)*
$\langle proof \rangle$

13.6 Minus operation for maps

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

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

lemma *map-member-plus*:
$(x, y) \in_m f ++ g \longleftrightarrow ((x \notin dom(g) \wedge (x, y) \in_m f) \vee (x, y) \in_m g)$
$\langle proof \rangle$

lemma *map-member-minus*:
$(x, y) \in_m f\ --\ g \longleftrightarrow (x, y) \in_m f \wedge (\neg\ (x, y) \in_m g)$
$\langle proof \rangle$

lemma *map-minus-plus-commute*:
$dom(g) \cap dom(h) = \{\} \Longrightarrow (f -- g) ++ h = (f ++ h) -- g$
$\langle proof \rangle$

lemma *map-graph-minus*: $map\text{-}graph\ (f -- g) = map\text{-}graph\ f - map\text{-}graph\ g$
$\langle proof \rangle$

lemma *map-minus-common-subset*:
$[\![\ h \subseteq_m f;\ \ h \subseteq_m g\ ]\!] \Longrightarrow (f -- h = g -- h) = (f = g)$
$\langle proof \rangle$

13.7 Map Bind

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

lemma *option-bindSomeE* [*elim!*]:
$[\![\ X >>= F = Some(v);\ \ \bigwedge x.\ [\![\ X = Some(x);\ \ F(x) = Some(v)\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
$\langle proof \rangle$

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

lemma *ifSomeE* [*elim*]: $[\![\ (if\ c\ then\ Some(x)\ else\ None) = Some(y);\ \ [\![\ c;\ \ x = y\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
$\langle proof \rangle$

13.8 Range Restriction

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

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

lemma *ran-restrict-alt-def*: $f{\restriction}_B = (\lambda\ x.\ if\ x \in dom(f) \wedge the(f(x)) \in B\ then\ f\ x\ else\ None)$
$\langle proof \rangle$

lemma *ran-restrict-empty* [*simp*]: $f{\restriction}_{\{\}} = Map.empty$
$\langle proof \rangle$

lemma *ran-restrict-ran* [*simp*]: $f{\restriction}_{ran(f)} = f$
$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *dom-ran-restrict*: $dom(f{\upharpoonright}_B) \subseteq dom(f)$
$\langle proof \rangle$

lemma *ran-restrict-finite-dom* [*intro*]:
$finite(dom(f)) \Longrightarrow finite(dom(f{\upharpoonright}_B))$
$\langle proof \rangle$

lemma *dom-Some* [*simp*]: $dom\ (Some \circ f) = UNIV$
$\langle proof \rangle$

lemma *map-dres-rres-commute*: $f{\upharpoonright}_B\ |`\ A = (f\ |`\ A){\upharpoonright}_B$
$\langle proof \rangle$

lemma *ran-restrict-map-twice* [*simp*]: $(f{\upharpoonright}_A){\upharpoonright}_B = f{\upharpoonright}_{(A\ \cap\ B)}$
$\langle proof \rangle$

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

lemma *dom-right-map-add* [*simp*]: $[\![\ x \notin dom\ g;\ \ x \in dom\ f\ ]\!] \Longrightarrow (f\ {+}{+}\ g)\ x = f\ x$
$\langle proof \rangle$

lemma *map-add-restrict*:
$f\ {+}{+}\ g = (f\ |`\ (-\ dom\ g))\ {+}{+}\ g$
$\langle proof \rangle$

13.9 Map Inverse and Identity

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

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

lemma *map-id-on-in* [*simp*]:
$x \in xs \Longrightarrow map\text{-}id\text{-}on\ xs\ x = Some\ x$
$\langle proof \rangle$

lemma *map-id-on-out* [*simp*]:
$x \notin xs \Longrightarrow map\text{-}id\text{-}on\ xs\ x = None$
$\langle proof \rangle$

lemma *map-id-dom* [*simp*]: $dom\ (map\text{-}id\text{-}on\ xs) = xs$
$\langle proof \rangle$

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

⟨proof⟩

lemma *map-id-on-UNIV*[*simp*]: *map-id-on UNIV* = *Some*
⟨proof⟩

lemma *map-id-on-inj* [*simp*]:
inj-on (*map-id-on xs*) *xs*
⟨proof⟩

lemma *restrict-map-inj-on*:
inj-on f (*dom f*) $\Longrightarrow$ *inj-on* (*f* |‘ *A*) (*dom f* ∩ *A*)
⟨proof⟩

lemma *map-inv-empty* [*simp*]: *map-inv Map.empty* = *Map.empty*
⟨proof⟩

lemma *map-inv-id* [*simp*]:
map-inv (*map-id-on xs*) = *map-id-on xs*
⟨proof⟩

lemma *map-inv-Some* [*simp*]: *map-inv Some* = *Some*
⟨proof⟩

lemma *map-inv-f-f* [*simp*]:
⟦ *inj-on f* (*dom f*); *f x* = *Some y* ⟧ $\Longrightarrow$ *map-inv f y* = *Some x*
⟨proof⟩

lemma *dom-map-inv* [*simp*]:
dom (*map-inv f*) = *ran f*
⟨proof⟩

lemma *ran-map-inv* [*simp*]:
inj-on f (*dom f*) $\Longrightarrow$ *ran* (*map-inv f*) = *dom f*
⟨proof⟩

lemma *dom-image-ran*: *f* ‘ *dom f* = *Some* ‘ *ran f*
⟨proof⟩

lemma *inj-map-inv* [*intro*]:
inj-on f (*dom f*) $\Longrightarrow$ *inj-on* (*map-inv f*) (*ran f*)
⟨proof⟩

lemma *inj-map-bij*: *inj-on f* (*dom f*) $\Longrightarrow$ *bij-betw f* (*dom f*) (*Some* ‘ *ran f*)
⟨proof⟩

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

lemma *map-self-adjoin-complete* [*intro*]:
  assumes $dom\ f \cap ran\ f = \{\}$ $inj\text{-}on\ f\ (dom\ f)$
  shows $inj\text{-}on\ (map\text{-}inv\ f\ {+}{+}\ f)\ (dom\ f \cup ran\ f)$
⟨*proof*⟩

lemma *inj-completed-map* [*intro*]:
  assumes $dom\ f = ran\ f$ $inj\text{-}on\ f\ (dom\ f)$
  shows $inj\ (Some\ {+}{+}\ f)$
⟨*proof*⟩

lemma *bij-completed-map* [*intro*]:
  fixes $f :: 'a \rightharpoonup 'a$
  assumes $dom\ f = ran\ f$ $inj\text{-}on\ f\ (dom\ f)$
  shows $bij\text{-}betw\ (Some\ {+}{+}\ f)\ UNIV\ (range\ Some)$
⟨*proof*⟩

lemma *bij-map-Some*:
  $bij\text{-}betw\ f\ a\ (Some\ `\ b) \Longrightarrow bij\text{-}betw\ (the \circ f)\ a\ b$
  ⟨*proof*⟩

lemma *ran-map-add* [*simp*]:
  $m`(dom\ m \cap dom\ n) = n`(dom\ m \cap dom\ n) \Longrightarrow$
   $ran(m{+}{+}n) = ran\ n \cup ran\ m$
  ⟨*proof*⟩

lemma *ran-maplets* [*simp*]:
  $[\![\ length\ xs = length\ ys;\ \ distinct\ xs\ ]\!] \Longrightarrow ran\ [xs\ [\mapsto]\ ys] = set\ ys$
    ⟨*proof*⟩

lemma *inj-map-add*:
  $[\![\ inj\text{-}on\ f\ (dom\ f);\ \ inj\text{-}on\ g\ (dom\ g);\ \ ran\ f \cap ran\ g = \{\}\ ]\!] \Longrightarrow$
   $inj\text{-}on\ (f\ {+}{+}\ g)\ (dom\ f \cup dom\ g)$
  ⟨*proof*⟩

lemma *map-inv-add'*:
  assumes $inj\text{-}on\ f\ (dom\ f)$ $inj\text{-}on\ g\ (dom\ g)$
          $dom\ f \cap dom\ g = \{\}$ $ran\ f \cap ran\ g = \{\}$
 shows $map\text{-}inv\ (f\ {+}{+}\ g) = map\text{-}inv\ f\ {+}{+}\ map\text{-}inv\ g$
⟨*proof*⟩

lemma *map-inv-dom-res*:
  assumes $inj\text{-}on\ f\ (dom\ f)$
  shows $map\text{-}inv\ (f\ |`\ A) = (map\text{-}inv\ f) \upharpoonright_A$
  ⟨*proof*⟩

lemma *map-inv-ran-res*:
  assumes $inj\text{-}on\ f\ (dom\ f)$

shows $map\text{-}inv\ (f\ \restriction_A) = (map\text{-}inv\ f)\ |`\ A$
$\langle proof \rangle$

lemma *map-update-as-add*: $f(x \mapsto y) = f\ ++\ [x \mapsto y]$
$\langle proof \rangle$

lemma *map-add-lookup* [*simp*]:
$x \notin dom\ f \Longrightarrow ([x \mapsto y]\ ++\ f)\ x = Some\ y$
$\langle proof \rangle$

lemma *map-add-Some*: $Some\ ++\ f = map\text{-}id\text{-}on\ (-\ dom\ f)\ ++\ f$
$\langle proof \rangle$

lemma *distinct-map-dom*:
$x \notin set\ xs \Longrightarrow x \notin dom\ [xs\ [\mapsto]\ ys]$
$\langle proof \rangle$

lemma *distinct-map-ran*:
$[\![\ distinct\ xs;\ \ y \notin set\ ys;\ \ length\ xs = length\ ys\ ]\!] \Longrightarrow$
$y \notin ran\ ([xs\ [\mapsto]\ ys])$
$\langle proof \rangle$

lemma *maplets-lookup* [*dest*]:
$[\![\ length\ xs = length\ ys;\ \ distinct\ xs;\ \ \forall\ \ y.\ [xs\ [\mapsto]\ ys]\ x = Some\ y\ ]\!] \Longrightarrow y \in set\ ys$
$\langle proof \rangle$

lemma *maplets-distinct-inj* [*intro*]:
$[\![\ length\ xs = length\ ys;\ \ distinct\ xs;\ \ distinct\ ys;\ \ set\ xs \cap set\ ys = \{\}\ ]\!] \Longrightarrow$
$inj\text{-}on\ [xs\ [\mapsto]\ ys]\ (set\ xs)$
$\langle proof \rangle$

lemma *map-inv-maplet*[*simp*]: $map\text{-}inv\ [x \mapsto y] = [y \mapsto x]$
$\langle proof \rangle$

lemma *map-inv-add*:
assumes $inj\text{-}on\ f\ (dom\ f)\ inj\text{-}on\ g\ (dom\ g)$
$ran\ f \cap ran\ g = \{\}$
shows $map\text{-}inv\ (f\ ++\ g) = map\text{-}inv\ f\restriction_{(-\ dom\ g)}\ ++\ map\text{-}inv\ g$
$\langle proof \rangle$

lemma *map-inv-upd*:
assumes $inj\text{-}on\ f\ (dom\ f)\ inj\text{-}on\ g\ (dom\ g)\ v \notin ran\ f$
shows $map\text{-}inv\ (f(k \mapsto v)) = (map\text{-}inv\ (f\ |`\ (-\ \{k\})))(v \mapsto k)$
$\langle proof \rangle$

lemma *map-inv-maplets* [*simp*]:
$[\![\ length\ xs = length\ ys;\ \ distinct\ xs;\ \ distinct\ ys;\ \ set\ xs \cap set\ ys = \{\}\ ]\!] \Longrightarrow$
$map\text{-}inv\ [xs\ [\mapsto]\ ys] = [ys\ [\mapsto]\ xs]$

$\langle proof \rangle$

lemma *maplets-lookup-nth* [*simp*]:
$[\![$ *length xs* = *length ys*; *distinct xs*; $i <$ *length ys* $]\!] \Longrightarrow$
[*xs* [$\mapsto$] *ys*] (*xs* ! *i*) = *Some* (*ys* ! *i*)
$\langle proof \rangle$

theorem *inv-map-inv*:
assumes *inj-on f* (*dom f*) *ran f* = *dom f*
shows *inv* (*the* $\circ$ (*Some* ++ *f*)) = *the* $\circ$ *map-inv* (*Some* ++ *f*)
$\langle proof \rangle$

lemma *map-comp-dom*: *dom* ($g \circ_m f$) $\subseteq$ *dom f*
$\langle proof \rangle$

lemma *map-comp-assoc*: $f \circ_m (g \circ_m h) = f \circ_m g \circ_m h$
$\langle proof \rangle$

lemma *map-comp-runit* [*simp*]: $f \circ_m$ *Some* = *f*
$\langle proof \rangle$

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

lemma *map-comp-apply* [*simp*]: $(f \circ_m g)\ x = g(x)$ >>= *f*
$\langle proof \rangle$

lemma *map-graph-map-inv*: *inj-on f* (*dom f*) $\Longrightarrow$ *map-graph* (*map-inv f*) = (*map-graph f*)$^{-1}$
$\langle proof \rangle$

13.10 Merging of compatible maps

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

lemma *comp-map-unit*: *Map.empty* $\|_m f$
$\langle proof \rangle$

lemma *comp-map-refl*: $f \|_m f$
$\langle proof \rangle$

lemma *comp-map-sym*: $f \|_m g \Longrightarrow g \|_m f$
$\langle proof \rangle$

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*: $merge\ \{\} = Map.empty$
⟨*proof*⟩

lemma *merge-singleton*: $merge\ \{f\} = f$
⟨*proof*⟩

13.11 Conversion between lists and maps

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

lemma *map-of-list-nil* [*simp*]: $map\text{-}of\text{-}list\ [] = Map.empty$
⟨*proof*⟩

lemma *dom-map-of-list* [*simp*]: $dom\ (map\text{-}of\text{-}list\ xs) = \{0..<length\ xs\}$
⟨*proof*⟩

lemma *ran-map-of-list* [*simp*]: $ran\ (map\text{-}of\text{-}list\ xs) = set\ xs$
⟨*proof*⟩

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

lemma *list-of-map-empty* [*simp*]: $list\text{-}of\text{-}map\ Map.empty = []$
⟨*proof*⟩

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

lemma *map-of-list-inv* [*simp*]: $list\text{-}of\text{-}map\ (map\text{-}of\text{-}list\ xs) = xs$
⟨*proof*⟩

13.12 Map Comprehension

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

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

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

lemma *map-compr-eta*:
$[x \mapsto y\ |\ x\ y.\ (x,\ y) \in_m f] = f$
⟨*proof*⟩

lemma *map-compr-simple*:

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

lemma *map-compr-dom-simple* [*simp*]:
$dom\ [x \mapsto f\ x \mid x.\ P\ x] = \{x.\ P\ x\}$
⟨*proof*⟩

lemma *map-compr-ran-simple* [*simp*]:
$ran\ [x \mapsto f\ x \mid x.\ P\ x] = \{f\ x \mid x.\ P\ x\}$
⟨*proof*⟩

lemma *map-compr-eval-simple* [*simp*]:
$[x \mapsto f\ x \mid x.\ P\ x]\ x = (if\ \ (P\ x)\ \ then\ Some\ (f\ x)\ \ else\ None)$
⟨*proof*⟩

13.13 Sorted lists from maps

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

lemma *sorted-list-of-map-empty* [*simp*]:
$sorted\text{-}list\text{-}of\text{-}map\ \ Map.empty = []$
⟨*proof*⟩

lemma *sorted-list-of-map-inv*:
assumes $finite(dom(f))$
shows $map\text{-}of\ \ (sorted\text{-}list\text{-}of\text{-}map\ f) = f$
⟨*proof*⟩

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

13.14 Extra map lemmas

lemma *map-eqI*:
$[\![\ dom\ f\ =\ dom\ g;\ \ \forall\ \ x{\in}dom(f).\ the(f\ x)\ =\ the(g\ x)\ ]\!] \Longrightarrow f\ =\ g$
⟨*proof*⟩

lemma *map-restrict-dom* [*simp*]: $f\ |`\ dom\ f\ =\ f$
⟨*proof*⟩

lemma *map-restrict-dom-compl*: $f\ |`\ (-\ \ dom\ f)\ =\ Map.empty$
⟨*proof*⟩

lemma *restrict-map-neg-disj*:
$dom(f) \cap A = \{\} \Longrightarrow f\ |`\ (-\ A) = f$
⟨*proof*⟩

lemma *map-plus-restrict-dist*: $(f\ ++\ g)\ |`\ A = (f\ |`\ A)\ ++\ (g\ |`\ A)$
⟨*proof*⟩

lemma *map-plus-eq-left*:
  assumes $f ++ h = g ++ h$
  shows $(f \mid` (- \ dom \ h)) = (g \mid` (- \ dom \ h))$
$\langle proof \rangle$

lemma *map-add-split*:
  $dom(f) = A \cup B \Longrightarrow (f \mid` A) ++ (f \mid` B) = f$
  $\langle proof \rangle$

lemma *map-le-via-restrict*:
  $f \subseteq_m g \longleftrightarrow g \mid` dom(f) = f$
  $\langle proof \rangle$

lemma *map-add-cancel*:
  $f \subseteq_m g \Longrightarrow f ++ (g -- f) = g$
  $\langle proof \rangle$

lemma *map-le-iff-add*: $f \subseteq_m g \longleftrightarrow (\exists \ h. \ dom(f) \cap dom(h) = \{\} \wedge f ++ h = g)$
$\langle proof \rangle$

lemma *map-add-comm-weak*: $(\forall \ k \in dom \ m1 \cap dom \ m2. \ m1(k) = m2(k)) \Longrightarrow m1 ++ m2 = m2 ++ m1$
  $\langle proof \rangle$

lemma *map-add-comm-weak'*: $Q \mid` dom \ P = P \mid` dom \ Q \Longrightarrow P ++ Q = Q ++ P$
  $\langle proof \rangle$

lemma *map-compat-add*: $Q \mid` dom \ P = P \mid` dom \ Q \Longrightarrow R \mid` (dom \ Q \cup dom \ P) = (P ++ Q) \mid` dom \ R \Longrightarrow R \mid` dom \ P = P \mid` dom \ R$
  $\langle proof \rangle$

abbreviation *rel-map* $R \equiv$ *rel-fun* $(=)$ (*rel-option* R)

lemma *rel-map-iff*:
  rel-map $R\ f\ g \longleftrightarrow (dom(f) = dom(g) \wedge (\forall \ x{\in}dom(f). \ R \ (the \ (f\ x)) \ (the \ (g\ x))))$
  $\langle proof \rangle$

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* :: $('a \rightharpoonup 'b)$ *set*
morphisms *pfun-lookup pfun-of-map* ⟨*proof*⟩

type-notation *pfun* (**infixr** $\nrightarrow$ *1*)

setup-lifting *type-definition-pfun*

lemma *pfun-lookup-map* [*simp*]: *pfun-lookup* (*pfun-of-map f*) = *f*
⟨*proof*⟩

lift-bnf ('k, *pran*: 'v) *pfun* [*wits*: *Map.empty* :: $'k \Rightarrow 'v$ *option*] **for** *map*: *map-pfun* *rel*: *relt-pfun*
⟨*proof*⟩

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

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

definition *HOL.equal m1 m2* $\longleftrightarrow$ ($\forall$ *k*. *pfun-lookup m1 k* = *pfun-lookup m2 k*)

instance
⟨*proof*⟩

end

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

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

lift-definition *pdom* :: $('a, 'b)$ *pfun* $\Rightarrow 'a$ *set* **is** *dom* ⟨*proof*⟩

lemma *pran-rep-eq* [*transfer-rule*]: *pran f* = *ran* (*pfun-lookup f*)
⟨*proof*⟩

lift-definition *pfun-comp* :: $('b, 'c)$ *pfun* $\Rightarrow ('a, 'b)$ *pfun* $\Rightarrow ('a, 'c)$ *pfun* (**infixl** $\circ_p$ *55*) **is**
λ *f g*. $f \circ_m g$ ⟨*proof*⟩

lift-definition *map-pfun'* :: $('c \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a, 'b)$ *pfun* $\Rightarrow ('c, 'd)$ *pfun*
is λ*f g m*. (*map-option* $g \circ m \circ f$) **parametric** *map-parametric* ⟨*proof*⟩

functor *map-pfun′*
⟨*proof*⟩

lift-definition *pfun-member* :: *′a* × *′b* ⇒ (*′a*, *′b*) *pfun* ⇒ *bool* (**infix** $\in_p$ *50*) **is** $(\in_m)$ ⟨*proof*⟩

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

lift-definition *pfun-inv* :: (*′a*, *′b*) *pfun* ⇒ (*′b*, *′a*) *pfun* **is** *map-inv* ⟨*proof*⟩

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

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

lift-definition *pdom-res* :: *′a set* ⇒ (*′a*, *′b*) *pfun* ⇒ (*′a*, *′b*) *pfun* (**infixr** $\lhd_p$ *85*) **is** λ *A f*. *restrict-map f A* ⟨*proof*⟩

abbreviation *pdom-nres* (**infixr** $-\lhd_p$ *85*) **where** *pdom-nres A P* ≡ (− *A*) $\lhd_p$ *P*

lift-definition *pran-res* :: (*′a*, *′b*) *pfun* ⇒ *′b set* ⇒ (*′a*, *′b*) *pfun* (**infixl** $\rhd_p$ *86*) **is** *ran-restrict-map* ⟨*proof*⟩

abbreviation *pran-nres* (**infixr** $\rhd_p-$ *66*) **where** *pran-nres P A* ≡ *P* $\rhd_p$ (− *A*)

definition *pfun-image* :: *′a* ⇸ *′b* ⇒ *′a set* ⇒ *′b set* **where**
[*simp*]: *pfun-image f A* = *pran* (*A* $\lhd_p$ *f*)

lift-definition *pfun-graph* :: (*′a*, *′b*) *pfun* ⇒ (*′a* × *′b*) *set* **is** *map-graph* ⟨*proof*⟩

lift-definition *graph-pfun* :: (*′a* × *′b*) *set* ⇒ (*′a*, *′b*) *pfun* **is** *graph-map* ∘ *mk-functional* ⟨*proof*⟩

definition *pfun-pfun* :: *′a set* ⇒ *′b set* ⇒ (*′a* ⇸ *′b*) *set* **where**
pfun-pfun A B = {*f* :: *′a* ⇸ *′b*. *pdom*(*f*) ⊆ *A* ∧ *pran*(*f*) ⊆ *B*}

definition *pfun-tfun* :: *′a set* ⇒ *′b set* ⇒ (*′a* ⇸ *′b*) *set* **where**
pfun-tfun A B = {*f* ∈ *pfun-pfun A B*. *pdom*(*f*) = *UNIV*}

definition *pfun-ffun* :: *′a set* ⇒ *′b set* ⇒ (*′a* ⇸ *′b*) *set* **where**
pfun-ffun A B = {*f* ∈ *pfun-pfun A B*. *finite*(*pdom*(*f*))}

definition *pfun-pinj* :: *′a set* ⇒ *′b set* ⇒ (*′a* ⇸ *′b*) *set* **where**
pfun-pinj A B = {*f* ∈ *pfun-pfun A B*. *pfun-inj f*}

definition *pfun-psurj* :: *′a set* ⇒ *′b set* ⇒ (*′a* ⇸ *′b*) *set* **where**
pfun-psurj A B = {*f* ∈ *pfun-pfun A B*. *pran*(*f*) = *UNIV*}

definition *pfun-finj A B* = *pfun-ffun A B* ∩ *pfun-pinj A B*
definition *pfun-tinj A B* = *pfun-tfun A B* ∩ *pfun-pinj A B*
definition *pfun-tsurj A B* = *pfun-tfun A B* ∩ *pfun-psurj A B*
definition *pfun-bij A B* = *pfun-tfun A B* ∩ *pfun-pinj A B* ∩ *pfun-psurj A B*

lift-definition *pfun-entries* :: *'k set* ⇒ (*'k* ⇒ *'v*) ⇒ (*'k*, *'v*) *pfun* **is**
λ *d f x*. *if x* ∈ *d then Some* (*f x*) *else None* ⟨*proof*⟩

definition *pfuse* :: (*'a* ⇸ *'b*) ⇒ (*'a* ⇸ *'c*) ⇒ (*'a* ⇸ *'b* × *'c*)
where *pfuse f g* = *pfun-entries* (*pdom*(*f*) ∩ *pdom*(*g*)) (λ *x*. (*pfun-app f x*, *pfun-app g x*))

lift-definition *ptabulate* :: *'a list* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a*, *'b*) *pfun*
is λ*ks f*. (*map-of* (*List.map* (λ*k*. (*k*, *f k*)) *ks*)) ⟨*proof*⟩

lift-definition *pcombine* ::
(*'b* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *pfun* ⇒ (*'a*, *'b*) *pfun* ⇒ (*'a*, *'b*) *pfun*
is λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*) ⟨*proof*⟩

abbreviation *fun-pfun* ≡ *pfun-entries UNIV*

definition *pfun-disjoint* :: *'a* ⇸ *'b set* ⇒ *bool* **where**
pfun-disjoint S = (∀ *i* ∈ *pdom S*. ∀ *j* ∈ *pdom S*. *i* ≠ *j* ⟶ *pfun-app S i* ∩ *pfun-app S j* = {})

definition *pfun-partitions* :: *'a* ⇸ *'b set* ⇒ *'b set* ⇒ *bool* **where**
pfun-partitions S T = (*pfun-disjoint S* ∧ ⋃ (*pran S*) = *T*)

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

definition *pabs* :: *'a set* ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ *'a* ⇸ *'b* **where**
pabs A P f = (*A* ∩ *Collect P*) $\lhd_p$ *fun-pfun f*

definition *pcard* :: (*'a*, *'b*) *pfun* ⇒ *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* ⟨*proof*⟩
instance ⟨*proof*⟩
end

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

instantiation *pfun* :: (*type*, *type*) *oplus*

begin
lift-definition *oplus-pfun* :: $('a, 'b)$ *pfun* $\Rightarrow$ $('a, 'b)$ *pfun* $\Rightarrow$ $('a, 'b)$ *pfun* **is** $(++)$
$\langle proof \rangle$
instance $\langle proof \rangle$
end

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

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

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

instantiation *pfun* :: (*type*, *type*) *order*
begin
lift-definition *less-eq-pfun* :: $('a, 'b)$ *pfun* $\Rightarrow$ $('a, 'b)$ *pfun* $\Rightarrow$ *bool* **is**
$\lambda\, f\, g.\ f \subseteq_m g$ $\langle proof \rangle$
lift-definition *less-pfun* :: $('a, 'b)$ *pfun* $\Rightarrow$ $('a, 'b)$ *pfun* $\Rightarrow$ *bool* **is**
$\lambda\, f\, g.\ f \subseteq_m g \wedge f \neq g$ $\langle proof \rangle$
instance
$\langle proof \rangle$
end

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

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

instance *pfun* :: (*type*, *type*) *semilattice-inf*
$\langle proof \rangle$

lemma *pfun-subset-eq-least* [*simp*]:
$\{\}_p \subseteq_p f$
$\langle proof \rangle$

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*: $f \circ_p (g \circ_p h) = (f \circ_p g) \circ_p h$
⟨*proof*⟩

lemma *pfun-comp-left-id* [*simp*]: $pId \circ_p f = f$
⟨*proof*⟩

lemma *pfun-comp-right-id* [*simp*]: $f \circ_p pId = f$
⟨*proof*⟩

lemma *pfun-comp-left-zero* [*simp*]: $\{\}_p \circ_p f = \{\}_p$
⟨*proof*⟩

lemma *pfun-comp-right-zero* [*simp*]: $f \circ_p \{\}_p = \{\}_p$
⟨*proof*⟩

lemma *pfun-override-dist-comp*:
$(f \oplus g) \circ_p h = (f \circ_p h) \oplus (g \circ_p h)$
⟨*proof*⟩

lemma *pfun-minus-unit* [*simp*]:
fixes *f* :: ('*a*, '*b*) *pfun*
shows $f - \bot = f$
⟨*proof*⟩

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

fixes f :: $('a,\ 'b)$ *pfun*
shows $\bot - f = \bot$
⟨*proof*⟩

lemma *pfun-minus-self* [*simp*]:
fixes f :: $('a,\ 'b)$ *pfun*
shows $f - f = \bot$
⟨*proof*⟩

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

lemma *pfun-compat-add*: $(P :: 'a \nrightarrow 'b)\ \#\#\ Q \Longrightarrow P \oplus Q\ \#\#\ R \Longrightarrow P\ \#\#\ R$
⟨*proof*⟩

lemma *pfun-compat-addI*: $[\![\ (P :: 'a \nrightarrow 'b)\ \#\#\ Q;\ \ P\ \#\#\ R;\ \ Q\ \#\#\ R\ ]\!] \Longrightarrow P \oplus Q\ \#\#\ R$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *pfun-indep-compat*: $pdom(f) \cap pdom(g) = \{\} \Longrightarrow f\ \#\#\ g$
⟨*proof*⟩

lemma *pfun-override-commute*:
$pdom(f) \cap pdom(g) = \{\} \Longrightarrow f \oplus g = g \oplus f$
⟨*proof*⟩

lemma *pfun-override-commute-weak*:
$(\forall\ k \in pdom(f) \cap pdom(g).\ f(k)_p = g(k)_p) \Longrightarrow f \oplus g = g \oplus f$
⟨*proof*⟩

lemma *pfun-override-fully*: $pdom\ f \subseteq pdom\ g \Longrightarrow f \oplus g = g$
⟨*proof*⟩

lemma *pfun-override-res*: $pdom\ g - \lhd_p f \oplus g = f \oplus g$
⟨*proof*⟩

lemma *pfun-minus-override-commute*:
$pdom(g) \cap pdom(h) = \{\} \Longrightarrow (f - g) \oplus h = (f \oplus h) - g$
⟨*proof*⟩

lemma *pfun-override-minus*:
$f \subseteq_p g \Longrightarrow (g - f) \oplus f = g$
⟨*proof*⟩

lemma *pfun-minus-common-subset*:
$\llbracket h \subseteq_p f;\ h \subseteq_p g \rrbracket \Longrightarrow (f - h = g - h) = (f = g)$
$\langle proof \rangle$

lemma *pfun-minus-override*:
$pdom(f) \cap pdom(g) = \{\} \Longrightarrow (f \oplus g) - g = f$
$\langle proof \rangle$

lemma *pfun-override-pos*: $x \oplus y = \{\}_p \Longrightarrow x = \{\}_p$
$\langle proof \rangle$

lemma *pfun-le-override*: $pdom\ x \cap pdom\ y = \{\} \Longrightarrow x \leq x \oplus y$
$\langle proof \rangle$

14.3 Membership, application, and update

lemma *pfun-ext*: $\llbracket \bigwedge x\ y.\ (x,\ y) \in_p f \longleftrightarrow (x,\ y) \in_p g \rrbracket \Longrightarrow f = g$
$\langle proof \rangle$

lemma *pfun-member-alt-def*:
$(x,\ y) \in_p f \longleftrightarrow (x \in pdom\ f \wedge f(x)_p = y)$
$\langle proof \rangle$

lemma *pfun-member-override*:
$(x,\ y) \in_p f \oplus g \longleftrightarrow ((x \notin pdom(g) \wedge (x,\ y) \in_p f) \vee (x,\ y) \in_p g)$
$\langle proof \rangle$

lemma *pfun-member-minus*:
$(x,\ y) \in_p f - g \longleftrightarrow (x,\ y) \in_p f \wedge (\neg\ (x,\ y) \in_p g)$
$\langle proof \rangle$

lemma *pfun-app-in-ran* [*simp*]: $x \in pdom\ f \Longrightarrow f(x)_p \in pran\ f$
$\langle proof \rangle$

lemma *pfun-app-map* [*simp*]: $(pfun\text{-}of\text{-}map\ f)(x)_p = (if\ (x \in dom(f))\ then\ the\ (f\ x)\ else\ undefined)$
$\langle proof \rangle$

lemma *pfun-app-upd-1*: $x = y \Longrightarrow (f(x \mapsto v)_p)(y)_p = v$
$\langle proof \rangle$

lemma *pfun-app-upd-2*: $x \neq y \Longrightarrow (f(x \mapsto v)_p)(y)_p = f(y)_p$
$\langle proof \rangle$

lemma *pfun-app-upd* [*simp*]: $(f(x \mapsto e)_p)(y)_p = (if\ (x = y)\ then\ e\ else\ f(y)_p)$
$\langle proof \rangle$

lemma *pfun-graph-apply* [*simp*]: $rel\text{-}apply\ (pfun\text{-}graph\ f)\ x = f(x)_p$

$\langle proof \rangle$

lemma *pfun-upd-ext* [*simp*]: $x \in pdom(f) \Longrightarrow f(x \mapsto f(x)_p)_p = f$
$\langle proof \rangle$

lemma *pfun-app-add* [*simp*]: $x \in pdom(g) \Longrightarrow (f \oplus g)(x)_p = g(x)_p$
$\langle proof \rangle$

lemma *pfun-upd-add* [*simp*]: $f \oplus g(x \mapsto v)_p = (f \oplus g)(x \mapsto v)_p$
$\langle proof \rangle$

lemma *pfun-upd-add-left* [*simp*]: $x \notin pdom(g) \Longrightarrow f(x \mapsto v)_p \oplus g = (f \oplus g)(x \mapsto v)_p$
$\langle proof \rangle$

lemma *pfun-app-add'* [*simp*]: $e \notin pdom\ g \Longrightarrow (f \oplus g)(e)_p = f(e)_p$
$\langle proof \rangle$

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

lemma *pfun-upd-comm*:
assumes $x \neq y$
shows $f(y \mapsto u, x \mapsto v)_p = f(x \mapsto v, y \mapsto u)_p$
$\langle proof \rangle$

lemma *pfun-upd-comm-linorder* [*simp*]:
fixes $x\ y :: 'a :: linorder$
assumes $x < y$
shows $f(y \mapsto u, x \mapsto v)_p = f(x \mapsto v, y \mapsto u)_p$
$\langle proof \rangle$

lemma *pfun-upd-as-ovrd*: $f(k \mapsto v)_p = f \oplus \{k \mapsto v\}_p$
$\langle proof \rangle$

lemma *pfun-ovrd-single-upd*: $x \in pdom(g) \Longrightarrow f \oplus (\{x\} \lhd_p g) = f(x \mapsto g(x)_p)_p$
$\langle proof \rangle$

lemma *pfun-app-minus* [*simp*]: $x \notin pdom\ g \Longrightarrow (f - g)(x)_p = f(x)_p$
$\langle proof \rangle$

lemma *pfun-app-empty* [*simp*]: $\{\}_p(x)_p = undefined$
$\langle proof \rangle$

lemma *pfun-app-not-in-dom*:
$x \notin pdom(f) \Longrightarrow f(x)_p = undefined$
$\langle proof \rangle$

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

$x \notin pdom\ g \Longrightarrow (f - g)(x \mapsto v)_p = (f(x \mapsto v)_p - g)$
$\langle proof \rangle$

lemma *pdom-member-minus-iff* [*simp*]:
$x \notin pdom\ g \Longrightarrow x \in pdom(f - g) \longleftrightarrow x \in pdom(f)$
$\langle proof \rangle$

lemma *psubseteq-pfun-upd1* [*intro*]:
$[\![f \subseteq_p g;\ x \notin pdom(g)]\!] \Longrightarrow f \subseteq_p g(x \mapsto v)_p$
$\langle proof \rangle$

lemma *psubseteq-pfun-upd2* [*intro*]:
$[\![f \subseteq_p g;\ x \notin pdom(f)]\!] \Longrightarrow f \subseteq_p g(x \mapsto v)_p$
$\langle proof \rangle$

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

lemma *psubseteq-dom-subset*:
$f \subseteq_p g \Longrightarrow pdom(f) \subseteq pdom(g)$
$\langle proof \rangle$

lemma *psubseteq-ran-subset*:
$f \subseteq_p g \Longrightarrow pran(f) \subseteq pran(g)$
$\langle proof \rangle$

lemma *pfun-eq-iff*: $f = g \longleftrightarrow (pdom(f) = pdom(g) \wedge (\forall\ x \in pdom(f).\ f(x)_p = g(x)_p))$
$\langle proof \rangle$

lemma *pfun-leI*: $[\![pdom\ f \subseteq pdom\ g;\ \forall x{\in}pdom\ f.\ f(x)_p = g(x)_p]\!] \Longrightarrow f \subseteq_p g$
$\langle proof \rangle$

lemma *pfun-le-iff*: $(f \subseteq_p g) = (pdom\ f \subseteq pdom\ g \wedge (\forall x{\in}pdom\ f.\ f(x)_p = g(x)_p))$
$\langle proof \rangle$

14.4 Map laws

lemma *map-pfun-empty* [*simp*]: $map\text{-}pfun\ f\ \{\}_p = \{\}_p$
$\langle proof \rangle$

lemma *map-pfun'-empty* [*simp*]: $map\text{-}pfun'\ f\ g\ \{\}_p = \{\}_p$
$\langle proof \rangle$

lemma *map-pfun-upd* [*simp*]: $map\text{-}pfun\ f\ (g(x \mapsto v)_p) = (map\text{-}pfun\ f\ g)(x \mapsto f\ v)_p$
$\langle proof \rangle$

lemma *map-pfun-apply* [*simp*]: $x \in pdom\ G \Longrightarrow (map\text{-}pfun\ F\ G)(x)_p = F(G(x)_p)$

$\langle proof \rangle$

lemma *map-pfun-as-pabs*: *map-pfun f g* $= (\lambda\ x \in pdom(g) \cdot f(g(x)_p))$
$\langle proof \rangle$

lemma *map-pfun-ovrd* [*simp*]: *map-pfun f* $(g \oplus h) =$ (*map-pfun f g*) $\oplus$ (*map-pfun f h*)
$\langle proof \rangle$

lemma *map-pfun-dres* [*simp*]: *map-pfun f* $(A \lhd_p g) = A \lhd_p$ *map-pfun f g*
$\langle proof \rangle$

14.5 Domain laws

lemma *pdom-zero* [*simp*]: *pdom* $\bot = \{\}$
$\langle proof \rangle$

lemma *pdom-pId-on* [*simp*]: *pdom* (*pId-on A*) $= A$
$\langle proof \rangle$

lemma *pdom-plus* [*simp*]: *pdom* $(f \oplus g) =$ *pdom f* $\cup$ *pdom g*
$\langle proof \rangle$

lemma *pdom-minus* [*simp*]: $g \leq f \Longrightarrow$ *pdom* $(f - g) =$ *pdom f* $-$ *pdom g*
$\langle proof \rangle$

lemma *pdom-inter*: *pdom* $(f \cap_p g) \subseteq$ *pdom f* $\cap$ *pdom g*
$\langle proof \rangle$

lemma *pdom-comp* [*simp*]: *pdom* $(g \circ_p f) =$ *pdom* $(f \rhd_p$ *pdom g*$)$
$\langle proof \rangle$

lemma *pdom-upd* [*simp*]: *pdom* $(f(k \mapsto v)_p) =$ *insert k* (*pdom f*)
$\langle proof \rangle$

lemma *pdom-pdom-res* [*simp*]: *pdom* $(A \lhd_p f) = A \cap pdom(f)$
$\langle proof \rangle$

lemma *pdom-graph-pfun*: *pdom* (*graph-pfun R*) $\subseteq$ *Domain R*
$\langle proof \rangle$

lemma *pdom-functional-graph-pfun* [*simp*]:
functional R $\Longrightarrow$ *pdom* (*graph-pfun R*) $=$ *Domain R*
$\langle proof \rangle$

lemma *pdom-pran-res-finite* [*simp*]:
finite (*pdom f*) $\Longrightarrow$ *finite* (*pdom* $(f \rhd_p A)$)
$\langle proof \rangle$

lemma *pdom-pfun-graph-finite* [*simp*]:
$finite\ (pdom\ f) \Longrightarrow finite\ (pfun\text{-}graph\ f)$
⟨*proof*⟩

lemma *pdom-map-pfun* [*simp*]: $pdom\ (map\text{-}pfun\ F\ G) = pdom\ G$
⟨*proof*⟩

lemma *rel-comp-pfun*: $R\ O\ pfun\text{-}graph\ f = (\lambda\ p.\ (fst\ p,\ pfun\text{-}app\ f\ (snd\ p)))\ `\ (R \rhd_r pdom(f))$
⟨*proof*⟩

lemma *pdom-empty-iff-dom-empty*: $f = \{\}_p \longleftrightarrow pdom\ f = \{\}$
⟨*proof*⟩

lemma *empty-map-pfunD* [*dest!*]: $\{\}_p = map\text{-}pfun\ f\ F \Longrightarrow F = \{\}_p$
⟨*proof*⟩

14.6 Range laws

lemma *pran-zero* [*simp*]: $pran\ \bot = \{\}$
⟨*proof*⟩

lemma *pran-pId-on* [*simp*]: $pran\ (pId\text{-}on\ A) = A$
⟨*proof*⟩

lemma *pran-upd* [*simp*]: $pran\ (f(k \mapsto v)_p) = insert\ v\ (pran\ ((-\ \{k\}) \lhd_p f))$
⟨*proof*⟩

lemma *pran-pran-res* [*simp*]: $pran\ (f \rhd_p A) = pran(f) \cap A$
⟨*proof*⟩

lemma *pran-comp* [*simp*]: $pran\ (g \circ_p f) = pran\ (pran\ f \lhd_p g)$
⟨*proof*⟩

lemma *pran-finite* [*simp*]: $finite\ (pdom\ f) \Longrightarrow finite\ (pran\ f)$
⟨*proof*⟩

lemma *pran-pdom*: $pran\ F = pfun\text{-}app\ F\ `\ pdom\ F$
⟨*proof*⟩

lemma *pran-override* [*simp*]: $pran\ (f \oplus g) = pran(g) \cup pran(pdom(g) -\!\lhd_p f)$
⟨*proof*⟩

14.7 Graph laws

lemma *pfun-graph-inv* [*code-unfold*]: $graph\text{-}pfun\ (pfun\text{-}graph\ f) = f$
⟨*proof*⟩

lemma *pfun-eq-graph*: $f = g \longleftrightarrow pfun\text{-}graph\ f = pfun\text{-}graph\ g$
⟨*proof*⟩

lemma *Dom-pfun-graph*: *Domain* (*pfun-graph f*) = *pdom f*
⟨*proof*⟩

lemma *Range-pfun-graph*: *Range* (*pfun-graph f*) = *pran f*
⟨*proof*⟩

lemma *card-pfun-graph*: *finite* (*pdom f*) $\Longrightarrow$ *card* (*pfun-graph f*) = *card* (*pdom f*)
⟨*proof*⟩

lemma *functional-pfun-graph* [*simp*]: *functional* (*pfun-graph f*)
⟨*proof*⟩

lemma *pfun-graph-zero*: *pfun-graph* $\perp = \{\}$
⟨*proof*⟩

lemma *pfun-graph-pId-on*: *pfun-graph* (*pId-on A*) = *Id-on A*
⟨*proof*⟩

lemma *pfun-graph-minus*: *pfun-graph* $(f - g)$ = *pfun-graph f* $-$ *pfun-graph g*
⟨*proof*⟩

lemma *pfun-graph-inter*: *pfun-graph* $(f \cap_p g)$ = *pfun-graph f* $\cap$ *pfun-graph g*
⟨*proof*⟩

lemma *pfun-graph-domres*: *pfun-graph* $(A \lhd_p f)$ = $(A \lhd_r$ *pfun-graph f*$)$
⟨*proof*⟩

lemma *pfun-graph-override*: *pfun-graph* $(f \oplus g)$ = *pfun-graph f* $\oplus$ *pfun-graph g*
⟨*proof*⟩

lemma *pfun-graph-update*: *pfun-graph* $(f(k \mapsto v)_p)$ = *insert* (k, v) $((- \{k\}) \lhd_r$ *pfun-graph f*$)$
⟨*proof*⟩

lemma *pfun-graph-comp*: *pfun-graph* $(f \circ_p g)$ = *pfun-graph g O pfun-graph f*
⟨*proof*⟩

lemma *comp-pfun-graph*: *pfun-graph f O pfun-graph g* = *pfun-graph* $(g \circ_p f)$
⟨*proof*⟩

lemma *pfun-graph-pfun-inv*: *pfun-inj f* $\Longrightarrow$ *pfun-graph* (*pfun-inv f*) = (*pfun-graph* $f)^{-1}$
⟨*proof*⟩

lemma *pfun-graph-pabs*: *pfun-graph* $(\lambda\ x \in A \mid P\ x \cdot f\ x) = \{(k, v).\ k \in A \wedge P\ k \wedge v = f\ k\}$
⟨*proof*⟩

lemma *pfun-graph-le-iff*:
pfun-graph f $\subseteq$ *pfun-graph g* $\longleftrightarrow$ *f* $\subseteq_p$ *g*
⟨*proof*⟩

lemma *pfun-member-iff* [*simp*]: $(k, v) \in$ *pfun-graph f* $\longleftrightarrow$ ($k \in$ *pdom*(*f*) $\wedge$ *pfun-app f k* $= v$)
⟨*proof*⟩

lemma *pfun-graph-rres*: *pfun-graph* ($f \rhd_p A$) = *pfun-graph f* $\rhd_r A$
⟨*proof*⟩

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*
⟨*proof*⟩

bundle *pfun-graph-lifting*
begin

unbundle *lifting-syntax*

lemma *bi-unique-cr-pfung* [*transfer-rule*]: *bi-unique cr-pfung*
⟨*proof*⟩

lemma *right-total-cr-pfung* [*transfer-rule*]: *right-total cr-pfung*
⟨*proof*⟩

lemma *cr-pfung-empty* [*transfer-rule*]: *cr-pfung* {} $\{\}_p$
⟨*proof*⟩

lemma *cr-pfung-dom* [*transfer-rule*]: (*cr-pfung* ===> (=)) *Domain pdom*
⟨*proof*⟩

lemma *cr-pfung-ran* [*transfer-rule*]: (*cr-pfung* ===> (=)) *Range pran*
⟨*proof*⟩

lemma *cr-pfung-id* [*transfer-rule*]: ((=) ===> *cr-pfung*) *Id-on pId-on*
⟨*proof*⟩

lemma *cr-pfung-ovrd* [*transfer-rule*]: (*cr-pfung* ===> *cr-pfung* ===> *cr-pfung*) ($\oplus$) ($\oplus$)
⟨*proof*⟩

lemma *cr-pfung-ovrd* [*transfer-rule*]: (*cr-pfung* ===> *cr-pfung* ===> *cr-pfung*) (*O*) (λ *x y*. $y \circ_p x$)
⟨*proof*⟩

lemma *cr-pfung-dres* [*transfer-rule*]: $((=)$ ===> *cr-pfung* ===> *cr-pfung*$)$ $(\lhd_r)$ $(\lhd_p)$
  $\langle proof \rangle$

lemma *cr-pfung-rres* [*transfer-rule*]: $($*cr-pfung* ===> $(=)$ ===> *cr-pfung*$)$ $(\rhd_r)$ $(\rhd_p)$
  $\langle proof \rangle$

lemma *cr-pfung-le* [*transfer-rule*]: $($*cr-pfung* ===> *cr-pfung* ===> $(=))$ $(\leq)$ $(\leq)$
  $\langle proof \rangle$

lemma *cr-pfung-update* [*transfer-rule*]: $($*cr-pfung* ===> $(=)$ ===> $(=)$ ===> *cr-pfung*$)$ $(\lambda\ f\ k\ v.\ insert\ (k,\ v)\ ((-\ \{k\})\ \lhd_r\ f))$ *pfun-upd*
  $\langle proof \rangle$

end

14.9 Partial Injections

lemma *pfun-inj-empty* [*simp*]: *pfun-inj* $\{\}_p$
  $\langle proof \rangle$

lemma *pinj-pId-on* [*simp*]: *pfun-inj* $($*pId-on* $A)$
  $\langle proof \rangle$

lemma *pfun-inj-inv-inv*: *pfun-inj* $f \Longrightarrow$ *pfun-inv* $($*pfun-inv* $f) = f$
  $\langle proof \rangle$

lemma *pfun-inj-inv*: *pfun-inj* $f \Longrightarrow$ *pfun-inj* $($*pfun-inv* $f)$
  $\langle proof \rangle$

lemma *f-pfun-inv-f-apply*: $[\![$ *pfun-inj* f; $x \in$ *pran* f $]\!] \Longrightarrow f($*pfun-inv* $f(x)_p)_p = x$
  $\langle proof \rangle$

lemma *pfun-inv-f-f-apply*: $[\![$ *pfun-inj* f; $x \in$ *pdom* f $]\!] \Longrightarrow$ *pfun-inv* $f(f(x)_p)_p = x$
  $\langle proof \rangle$

lemma *pfun-inj-upd*: $[\![$ *pfun-inj* f; $v \notin$ *pran* f $]\!] \Longrightarrow$ *pfun-inj* $(f(k \mapsto v)_p)$
  $\langle proof \rangle$

lemma *pfun-inj-dres*: *pfun-inj* $f \Longrightarrow$ *pfun-inj* $(A \lhd_p f)$
  $\langle proof \rangle$

lemma *pfun-inj-rres*: *pfun-inj* $f \Longrightarrow$ *pfun-inj* $(f \rhd_p A)$
  $\langle proof \rangle$

lemma *pfun-inj-comp*: $[\![$ *pfun-inj* f; *pfun-inj* g $]\!] \Longrightarrow$ *pfun-inj* $(f \circ_p g)$
  $\langle proof \rangle$

lemma *pfun-inj-ovrd*: $\llbracket$ *pfun-inj f*; *pfun-inj g*; *pran f* $\cap$ *pran g* $= \{\}$ $\rrbracket \Longrightarrow$ *pfun-inj* $(f \oplus g)$
⟨*proof*⟩

lemma *pfun-inv-dres*: *pfun-inj f* $\Longrightarrow$ *pfun-inv* $(A \lhd_p f) =$ (*pfun-inv f*) $\rhd_p A$
⟨*proof*⟩

lemma *pfun-inv-rres*: *pfun-inj f* $\Longrightarrow$ *pfun-inv* $(f \rhd_p A) = A \lhd_p$ (*pfun-inv f*)
⟨*proof*⟩

lemma *pfun-inv-empty* [*simp*]: *pfun-inv* $\{\}_p = \{\}_p$
⟨*proof*⟩

lemma *pdom-pfun-inv* [*simp*]: *pdom* (*pfun-inv f*) = *pran f*
⟨*proof*⟩

lemma *pfun-inv-add*:
assumes *pfun-inj f pfun-inj g pran f* $\cap$ *pran g* $= \{\}$
shows *pfun-inv* $(f \oplus g) =$ (*pfun-inv f* $\rhd_p$ $(-$ *pdom g*$)) \oplus$ *pfun-inv g*
⟨*proof*⟩

lemma *pfun-inv-upd*:
assumes *pfun-inj f* $v \notin$ *pran f*
shows *pfun-inv* $(f(k \mapsto v)_p) =$ (*pfun-inv* $((- \{k\}) \lhd_p f))(v \mapsto k)_p$
⟨*proof*⟩

14.10 Domain restriction laws

lemma *pdom-res-zero* [*simp*]: $A \lhd_p \{\}_p = \{\}_p$
⟨*proof*⟩

lemma *pdom-res-empty* [*simp*]:
$(\{\} \lhd_p f) = \{\}_p$
⟨*proof*⟩

lemma *pdom-res-pdom* [*simp*]:
pdom$(f) \lhd_p f = f$
⟨*proof*⟩

lemma *pdom-res-UNIV* [*simp*]: *UNIV* $\lhd_p f = f$
⟨*proof*⟩

lemma *pdom-res-alt-def*: $A \lhd_p f = f \circ_p$ *pId-on A*
⟨*proof*⟩

lemma *pdom-res-upd-in* [*simp*]:
$k \in A \Longrightarrow A \lhd_p f(k \mapsto v)_p = (A \lhd_p f)(k \mapsto v)_p$
⟨*proof*⟩

lemma *pdom-res-upd-out* [*simp*]:
$k \notin A \Longrightarrow A \lhd_p f(k \mapsto v)_p = A \lhd_p f$
$\langle proof \rangle$

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

lemma *pdom-antires-insert-notin* [*simp*]:
$k \notin pdom(f) \Longrightarrow (- \ insert\ k\ A) \lhd_p f = (- A) \lhd_p f$
$\langle proof \rangle$

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

lemma *pdom-res-minus* [*simp*]: $A \lhd_p (f - g) = (A \lhd_p f) - g$
$\langle proof \rangle$

lemma *pdom-res-swap*: $(A \lhd_p f) \rhd_p B = A \lhd_p (f \rhd_p B)$
$\langle proof \rangle$

lemma *pdom-res-twice* [*simp*]: $A \lhd_p (B \lhd_p f) = (A \cap B) \lhd_p f$
$\langle proof \rangle$

lemma *pdom-res-comp* [*simp*]: $A \lhd_p (g \circ_p f) = g \circ_p (A \lhd_p f)$
$\langle proof \rangle$

lemma *pdom-res-apply* [*simp*]:
$x \in A \Longrightarrow (A \lhd_p f)(x)_p = f(x)_p$
$\langle proof \rangle$

lemma *pdom-res-frame-update* [*simp*]:
$[\![x \in pdom(f); (-\{x\}) \lhd_p f = (-\{x\}) \lhd_p g]\!] \Longrightarrow g(x \mapsto f(x)_p)_p = f$
$\langle proof \rangle$

lemma *pdres-rres-commute*: $A \lhd_p (P \rhd_p B) = (A \lhd_p P) \rhd_p B$
$\langle proof \rangle$

lemma *pdom-nres-disjoint*: $pdom(f) \cap A = \{\} \Longrightarrow (- A) \lhd_p f = f$
$\langle proof \rangle$

lemma *pranres-pdom* [*simp*]: $pdom\ (f \rhd_p A) \lhd_p f = f \rhd_p A$
$\langle proof \rangle$

lemma *pdom-pranres* [*simp*]: $pdom\ (f \rhd_p A) \subseteq pdom\ f$
$\langle proof \rangle$

lemma *pfun-split-domain*: $A \lhd_p f \oplus (- A) \lhd_p f = f$

$\langle proof \rangle$

14.11 Range restriction laws

lemma *pran-res-UNIV* [*simp*]: $f \rhd_p UNIV = f$
$\langle proof \rangle$

lemma *pran-res-empty* [*simp*]: $f \rhd_p \{\} = \{\}_p$
$\langle proof \rangle$

lemma *pran-res-zero* [*simp*]: $\{\}_p \rhd_p A = \{\}_p$
$\langle proof \rangle$

lemma *pran-res-upd-1* [*simp*]: $v \in A \Longrightarrow f(x \mapsto v)_p \rhd_p A = (f \rhd_p A)(x \mapsto v)_p$
$\langle proof \rangle$

lemma *pran-res-upd-2* [*simp*]: $v \notin A \Longrightarrow f(x \mapsto v)_p \rhd_p A = ((- \{x\}) \lhd_p f) \rhd_p A$
$\langle proof \rangle$

lemma *pran-res-twice* [*simp*]: $f \rhd_p A \rhd_p B = f \rhd_p (A \cap B)$
$\langle proof \rangle$

lemma *pran-res-alt-def*: $f \rhd_p A = pId\text{-}on\ A \circ_p f$
$\langle proof \rangle$

lemma *pran-res-override*: $(f \oplus g) \rhd_p A \subseteq_p (f \rhd_p A) \oplus (g \rhd_p A)$
$\langle proof \rangle$

lemma *pcomp-ranres* [*simp*]: $(f \circ_p g) \rhd_p A = (f \rhd_p A) \circ_p g$
$\langle proof \rangle$

lemma *pranres-le*: $A \subseteq B \Longrightarrow f \rhd_p A \leq f \rhd_p B$
$\langle proof \rangle$

lemma *pranres-neg-ran* [*simp*]: $P \rhd_p - pran\ P = \{\}_p$
$\langle proof \rangle$

14.12 Preimage Laws

lemma *ppreimageI* [*intro!*]: $[\![x \in pdom(f);\ f(x)_p \in A]\!] \Longrightarrow x \in pdom\ (f \rhd_p A)$
$\langle proof \rangle$

lemma *ppreimageD*: $x \in pdom\ (f \rhd_p A) \Longrightarrow \exists\ y \in A.\ f(x)_p = y$
$\langle proof \rangle$

lemma *ppreimageE* [*elim!*]: $[\![x \in pdom\ (f \rhd_p A);\ \bigwedge y.\ [\![x \in pdom(f);\ y \in A;\ f(x)_p = y]\!] \Longrightarrow P]\!] \Longrightarrow P$
$\langle proof \rangle$

lemma *mem-pimage-iff*: $x \in pran\ (A \lhd_p f) \longleftrightarrow (\exists\ y \in A \cap pdom(f).\ f(y)_p = x)$
$\langle proof \rangle$

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

14.13 Composition

lemma *pcomp-apply* [*simp*]: $[\![\ x \in pdom(g)\ ]\!] \Longrightarrow (f \circ_p\ g)(x)_p = f(g(x)_p)_p$
$\langle proof \rangle$

lemma *pcomp-mono*: $[\![\ f \leq f';\ \ g \leq g'\ ]\!] \Longrightarrow f \circ_p\ g \leq f' \circ_p\ g'$
$\langle proof \rangle$

lemma *pdom-UNIV-comp*: $pdom\ f = UNIV \Longrightarrow pdom\ (f \circ_p\ g) = pdom\ g$
$\langle proof \rangle$

14.14 Entries

lemma *pfun-entries-empty* [*simp*]: $pfun\text{-}entries\ \{\}\ f = \{\}_p$
$\langle proof \rangle$

lemma *pdom-pfun-entries* [*simp*]: $pdom\ (pfun\text{-}entries\ A\ f) = A$
$\langle proof \rangle$

lemma *pran-pfun-entries* [*simp*]: $pran\ (pfun\text{-}entries\ A\ f) = f\ `\ A$
$\langle proof \rangle$

lemma *pfun-entries-apply-1* [*simp*]:
$x \in d \Longrightarrow (pfun\text{-}entries\ d\ f)(x)_p = f\ x$
$\langle proof \rangle$

lemma *pfun-entries-apply-2* [*simp*]:
$x \notin d \Longrightarrow (pfun\text{-}entries\ d\ f)(x)_p = undefined$
$\langle proof \rangle$

lemma *pdom-res-entries*: $A \lhd_p\ pfun\text{-}entries\ B\ f = pfun\text{-}entries\ (A \cap B)\ f$
$\langle proof \rangle$

lemma *pfuse-empty* [*simp*]: $pfuse\ \{\}_p\ g = \{\}_p$
$\langle proof \rangle$

lemma *pfuse-app* [*simp*]:
$[\![\ e \in pdom\ F;\ \ e \in pdom\ G\ ]\!] \Longrightarrow (pfuse\ F\ G)(e)_p = (F(e)_p,\ G(e)_p)$
$\langle proof \rangle$

lemma *pdom-pfuse* [*simp*]: $pdom\ (pfuse\ f\ g) = pdom(f) \cap pdom(g)$
$\langle proof \rangle$

lemma *pfuse-upd*:
$pfuse\ (f(k \mapsto v)_p)\ g =$
$(if\ k \in pdom\ g\ then\ (pfuse\ ((-\{k\}) \lhd_p f)\ g)(k \mapsto (v,\ pfun\text{-}app\ g\ k))_p\ else\ pfuse\ f\ g)$
$\langle proof \rangle$

14.15 Lambda abstraction

lemma *pabs-cong*:
assumes $A = B \bigwedge x.\ x \in A \Longrightarrow P(x) = Q(x) \bigwedge x.\ [\![x \in A;\ \ P\ x]\!] \Longrightarrow F(x) = G(x)$
shows $(\lambda\ x \in A \mid P\ x \cdot F(x)) = (\lambda\ x \in B \mid Q\ x \cdot G(x))$
$\langle proof \rangle$

lemma *pabs-apply* [*simp*]: $[\![y \in A;\ \ P\ y]\!] \Longrightarrow (\lambda\ x \in A \mid P\ x \cdot f\ x)\ (y)_p = f\ y$
$\langle proof \rangle$

lemma *pdom-pabs* [*simp*]: $pdom\ (\lambda\ x \in A \mid P\ x \cdot f\ x) = A \cap Collect\ P$
$\langle proof \rangle$

lemma *pran-pabs* [*simp*]: $pran\ (\lambda\ x \in A \mid P\ x \cdot f\ x) = \{f\ x \mid x.\ x \in A \wedge P\ x\}$
$\langle proof \rangle$

lemma *pabs-eta* [*simp*]: $(\lambda\ x \in pdom(f) \cdot f(x)_p) = f$
$\langle proof \rangle$

lemma *pabs-id* [*simp*]: $(\lambda\ x \in A \mid P\ x \cdot x) = pId\text{-}on\ \{x{\in}A.\ P\ x\}$
$\langle proof \rangle$

lemma *pfun-entries-pabs*: $pfun\text{-}entries\ A\ f = (\lambda\ x \in A \cdot f\ x)$
$\langle proof \rangle$

lemma *pabs-empty* [*simp*]: $(\lambda\ x{\in}\{\} \cdot f(x)) = \{\}_p$
$\langle proof \rangle$

lemma *pabs-insert-maplet*: $(\lambda\ x{\in}insert\ y\ A \cdot f(x)) = (\lambda\ x{\in}A \cdot f(x)) \oplus \{y \mapsto f(y)\}_p$
$\langle proof \rangle$

This rule can perhaps be simplified

lemma *pcomp-pabs*:
$(\lambda\ x \in A \mid P\ x \cdot f\ x) \circ_p (\lambda\ x \in B \mid Q\ x \cdot g\ x)$
$= (\lambda\ x \in pdom\ (pabs\ B\ Q\ g \rhd_p (A \cap Collect\ P)) \cdot (f\ (g\ x)))$
$\langle proof \rangle$

lemma *pabs-rres* [*simp*]: $pabs\ A\ P\ f \rhd_p B = pabs\ A\ (\lambda\ x.\ P\ x \wedge f\ x \in B)\ f$
$\langle proof \rangle$

lemma *pabs-simple-comp* [*simp*]: $(\lambda\ x \cdot f\ x) \circ_p g(k \mapsto v)_p = ((\lambda\ x \cdot f\ x) \circ_p g)(k \mapsto f\ v)_p$
  $\langle proof \rangle$

lemma *pabs-comp*: $(\lambda\ x \in A \cdot f\ x) \circ_p g = (\lambda\ x \in pdom\ (g \rhd_p A) \cdot f\ (pfun\text{-}app\ g\ x))$
  $\langle proof \rangle$

lemma *map-pfun-pabs* [*simp*]: $map\text{-}pfun\ f\ (\lambda\ x{\in}A \mid B(x) \cdot g(x)) = (\lambda\ x{\in}A \mid B(x) \cdot f(g(x)))$
  $\langle proof \rangle$

14.16 Singleton Partial Functions

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

lemma *pfun-singleton-dom*: $pfun\text{-}singleton\ f \longleftrightarrow (\exists\ k.\ pdom(f) = \{k\})$
  $\langle proof \rangle$

lemma *pfun-singleton-maplet* [*simp*]:
  $pfun\text{-}singleton\ \{k \mapsto v\}_p$
  $\langle proof \rangle$

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

lemma *dest-pfsingle-maplet* [*simp*]: $dest\text{-}pfsingle\ \{k \mapsto v\}_p = (k, v)$
  $\langle proof \rangle$

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*]: $pfun\text{-}sum\ \{\}_p = 0$
  $\langle proof \rangle$

lemma *pfun-sum-upd-1*:
  assumes $finite(pdom(m))\ k \notin pdom(m)$
  shows $pfun\text{-}sum\ (m(k \mapsto v)_p) = pfun\text{-}sum\ m + v$
$\langle proof \rangle$

lemma *pfun-sums-upd-2*:
  assumes $finite(pdom(m))$
  shows $pfun\text{-}sum\ (m(k \mapsto v)_p) = pfun\text{-}sum\ ((-\ \{k\}) \lhd_p m) + v$
$\langle proof \rangle$

lemma *pfun-sum-dom-res-insert* [*simp*]:

assumes $x \in pdom\ f\ x \notin A\ finite\ A$
shows $pfun\text{-}sum\ ((insert\ x\ A) \lhd_p f) = f(x)_p + pfun\text{-}sum\ (A \lhd_p f)$
$\langle proof \rangle$

lemma *pfun-sum-pdom-res*:
fixes $f :: ('a,'b::ab\text{-}group\text{-}add)\ pfun$
assumes $finite(pdom\ f)$
shows $pfun\text{-}sum\ (A \lhd_p f) = pfun\text{-}sum\ f - (pfun\text{-}sum\ ((-\ A) \lhd_p f))$
$\langle proof \rangle$

lemma *pfun-sum-pdom-antires* [*simp*]:
fixes $f :: ('a,'b::ab\text{-}group\text{-}add)\ pfun$
assumes $finite(pdom\ f)$
shows $pfun\text{-}sum\ ((-\ A) \lhd_p f) = pfun\text{-}sum\ f - pfun\text{-}sum\ (A \lhd_p f)$
$\langle proof \rangle$

14.18 Conversions

definition $list\text{-}pfun :: 'a\ list \Rightarrow nat \nrightarrow 'a$ **where**
$list\text{-}pfun\ xs = (\lambda\ i \mid 0 < i \wedge i \leq length\ xs \cdot xs\ !\ (i-1))$

lemma *pdom-list-pfun* [*simp*]: $pdom\ (list\text{-}pfun\ xs) = \{1..length\ xs\}$
$\langle proof \rangle$

lemma *pran-list-pfun* [*simp*]: $pran\ (list\text{-}pfun\ xs) = set\ xs$
$\langle proof \rangle$

lemma *pfun-app-list-pfun*: $[\![\ 0 < i;\ \ i \leq length\ xs\ ]\!] \Longrightarrow (list\text{-}pfun\ xs)(i)_p = xs\ !\ (i - 1)$
$\langle proof \rangle$

lemma *pfun-graph-list-pfun*: $pfun\text{-}graph\ (list\text{-}pfun\ xs) = (\lambda\ i.\ (i,\ xs\ !\ (i - 1)))\ `\ \{1..length\ xs\}$
$\langle proof \rangle$

lemma *range-list-pfun*:
$range\ list\text{-}pfun = \{f :: nat \nrightarrow 'a.\ \exists\ i.\ pdom(f) = \{1..i\}\}$
$\langle proof \rangle$

lemma *list-pfun-le-iff-prefix* [*simp*]: $list\text{-}pfun\ xs \leq list\text{-}pfun\ ys \longleftrightarrow xs \leq ys$
$\langle proof \rangle$

lemma *pfun-upd-le-iff*: $(f(k \mapsto v)_p \subseteq_p g) = (k \in pdom\ g \wedge g(k)_p = v \wedge (-\ \{k\}) \lhd_p f \subseteq_p g)$
$\langle proof \rangle$

lemma *pfun-upd-le-pfun-upd*: $(f(k \mapsto v)_p \subseteq_p g(k \mapsto v)_p) = ((-\ \{k\}) \lhd_p f \subseteq_p (-\ \{k\}) \lhd_p g)$
$\langle proof \rangle$

14.19 Partial Function Lens

definition *pfun-lens* :: $'a \Rightarrow ('b \Longrightarrow ('a, 'b)\ pfun)$ **where**
[*lens-defs*]: $pfun\text{-}lens\ i = (\!|\ lens\text{-}get = \lambda\ s.\ s(i)_p,\ lens\text{-}put = \lambda\ s\ v.\ s(i \mapsto v)_p\ |\!)$

lemma *pfun-lens-mwb* [*simp*]: *mwb-lens* (*pfun-lens i*)
⟨*proof*⟩

lemma *pfun-lens-src*: $\mathcal{S}_{pfun\text{-}lens\ i} = \{f.\ i \in pdom(f)\}$
⟨*proof*⟩

lemma *lens-override-pfun-lens*:
$x \in pdom(g) \Longrightarrow f \oplus_L g\ on\ pfun\text{-}lens\ x = f \oplus (\{x\} \lhd_p g)$
⟨*proof*⟩

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*]: $prism\text{-}fun\ c\ A\ PB = (\lambda\ x{\in}build_c\ `\ A\ |\ fst\ (PB\ (the\ (match_c\ x)))\ \cdot\ snd\ (PB\ (the\ (match_c\ x))))$

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*]: $prism\text{-}fun\text{-}upd\ F\ c\ A\ PB = F \oplus prism\text{-}fun\ c\ A\ PB$

nonterminal *prism-maplet* **and** *prism-maplets*

syntax
-prism-maplet :: $id \Rightarrow pttrn \Rightarrow logic \Rightarrow logic \Rightarrow logic \Rightarrow prism\text{-}maplet$ (-{- ∈ -./ -} ⇒ -)
-prism-maplet-mem :: $id \Rightarrow pttrn \Rightarrow logic \Rightarrow logic \Rightarrow prism\text{-}maplet$ (-{- ∈ -} ⇒ -)
-prism-maplet-simple :: $id \Rightarrow pttrn \Rightarrow logic \Rightarrow prism\text{-}maplet$ (- - ⇒ -)
:: $prism\text{-}maplet \Rightarrow prism\text{-}maplets$ (-)
-prism-Maplets :: $[prism\text{-}maplet, prism\text{-}maplets] \Rightarrow prism\text{-}maplets$ (- |/ -)
-prism-fun-upd :: $logic \Rightarrow prism\text{-}maplets \Rightarrow logic$ (-'(-') [900, 0] 900)
-prism-fun :: $prism\text{-}maplets \Rightarrow logic$ ({-}$_p$)

translations
$f(c\{v \in A.\ P\} \Rightarrow B)$ == *CONST prism-fun-upd f c A* $(\lambda\ v.\ (P, B))$
$f(c\{v \in A\} \Rightarrow B) == f(c\{v \in A.\ CONST\ True\} \Rightarrow B)$
$f(c\ v \Rightarrow B) == f(c\{v \in CONST\ UNIV\} \Rightarrow B)$
-prism-fun-upd m (*-prism-Maplets xy ms*) ⇌ *-prism-fun-upd* (*-prism-fun-upd m xy*) *ms*
-prism-fun ms ⇌ *-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*: $wb\text{-}prism\ c \Longrightarrow pdom(prism\text{-}fun\ c\ A\ PB) = \{build_c\ v \mid v.\ v \in A \wedge fst\ (PB\ v)\}$
⟨*proof*⟩

lemma *prism-fun-compat*: $c \nabla d \Longrightarrow prism\text{-}fun\ c\ A\ PB\ \#\#\ prism\text{-}fun\ d\ B\ QB$
⟨*proof*⟩

lemma *prism-fun-commute*: $c \nabla d \Longrightarrow prism\text{-}fun\ c\ A\ PB \oplus prism\text{-}fun\ d\ B\ QB = prism\text{-}fun\ d\ B\ QB \oplus prism\text{-}fun\ c\ A\ PB$
⟨*proof*⟩

lemma *prism-fun-apply*: $[\![\ wb\text{-}prism\ c;\ \ v \in A;\ \ fst\ (PB\ v)\ ]\!] \Longrightarrow (prism\text{-}fun\ c\ A\ PB)(build_c\ v)_p = snd\ (PB\ v)$
⟨*proof*⟩

lemma *prism-fun-update-app-1* [*simp*]: $[\![\ wb\text{-}prism\ c;\ \ v \in A;\ \ P\ v\ ]\!] \Longrightarrow (f(c\{x \in A.\ P(x)\} \Rightarrow B(x)))(build_c\ v)_p = B\ v$
⟨*proof*⟩

lemma *prism-fun-update-app-2* [*simp*]: $[\![\ wb\text{-}prism\ c;\ \ wb\text{-}prism\ d;\ \ d \nabla c\ ]\!] \Longrightarrow (f(c\{x \in A.\ P(x)\} \Rightarrow B(x)))(build_d\ v)_p = f(build_d\ v)_p$
⟨*proof*⟩

lemma *prism-fun-update-cancel* [*simp*]: $f(c\{x \in A.\ P(x)\} \Rightarrow g(x) \mid c\{x \in A.\ P(x)\} \Rightarrow h(x)) = f(c\{x \in A.\ P(x)\} \Rightarrow h(x))$
⟨*proof*⟩

lemma *prism-fun-update-commute*:
$c \nabla d \Longrightarrow f(c\{x \in A.\ P(x)\} \Rightarrow g(x) \mid d\{y \in B.\ Q(y)\} \Rightarrow h(y))$
$= f(d\{y \in B.\ Q(y)\} \Rightarrow h(y) \mid c\{x \in A.\ P(x)\} \Rightarrow g(x))$
⟨*proof*⟩

lemma *case-sum-Plus*: *case-sum f g* ‘ $(A <\!+\!> B) = (f`A) \cup (g`B)$
⟨*proof*⟩

lemma *build-in-dom-prism-fun*: $[\![\ wb\text{-}prism\ c;\ \ x \in A;\ \ fst\ (PB\ x)\ ]\!] \Longrightarrow build_c\ x \in pdom\ (prism\text{-}fun\ c\ A\ PB)$
⟨*proof*⟩

lemma *prism-fun-combine*:
assumes $wb\text{-}prism\ c\ wb\text{-}prism\ d\ c \nabla d$
shows $prism\text{-}fun\ c\ A\ PB \oplus prism\text{-}fun\ d\ B\ QB = prism\text{-}fun\ (c +_{\triangle} d)\ (A <\!+\!> B)\ (case\text{-}sum\ PB\ QB)$
⟨*proof*⟩

lemma *prism-diff-implies-indep-funs*:
$[\![\ wb\text{-}prism\ c;\ \ wb\text{-}prism\ d;\ \ c \nabla d\ ]\!] \Longrightarrow pdom(prism\text{-}fun\ c\ A\ P\sigma) \cap pdom(prism\text{-}fun$

$d\ B\ Q\varrho) = \{\}$
⟨proof⟩

lemma *prism-fun-cong*: $[\![\ c = d;\ \ A = B;\ \ PB = QB\ ]\!] \Longrightarrow$ *prism-fun* $c\ A\ PB =$ *prism-fun* $d\ B\ QB$
⟨proof⟩

lemma *prism-fun-cong2*:
assumes
wb-prism c_1 *wb-prism* c_2
$c_1 = c_2\ A_1 = A_2$
$\bigwedge i.\ i \in A_1 \Longrightarrow P_1\ i \longleftrightarrow P_2\ i$
$\bigwedge i.\ [\![\ i \in A_1;\ \ P_1\ i\ ]\!] \Longrightarrow B_1\ i = B_2\ i$
shows *prism-fun* $c_1\ A_1\ (\lambda\ x.\ (P_1\ x,\ B_1\ x)) =$ *prism-fun* $c_2\ A_2\ (\lambda\ y.\ (P_2\ y,\ B_2\ y))$
⟨proof⟩

lemma *map-pfun-prism-fun* [*simp*]: *map-pfun* f (*prism-fun* $a\ A\ (\lambda\ x.\ (B\ x,\ C\ x))$) = *prism-fun* $a\ A\ (\lambda\ x.\ (B\ x,\ f\ (C\ x)))$
⟨proof⟩

lemma *prism-fun-as-map*:
wb-prism $b \Longrightarrow$
prism-fun $b\ A\ PB =$ *pfun-of-map* (λ x. *case* $match_b$ x *of None* $\Rightarrow$ *None* | *Some* $x \Rightarrow$ *if* $x \in A \wedge$ *fst* $(PB\ x)$ *then Some* (*snd* $(PB\ x)$) *else None*)
⟨proof⟩

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). $R\ (f(x)_p)\ (g(x)_p)$))
⟨proof⟩

lift-definition *pfun-of-alist* :: $('a \times 'b)$ *list* $\Rightarrow 'a \nrightarrow 'b$ **is** *map-of* ⟨proof⟩

lemma *pfun-of-alist-clearjunk*: *pfun-of-alist xs* = *pfun-of-alist* (*AList.clearjunk xs*)
⟨proof⟩

lemma *pfun-of-alist-Nil* [*simp*]: *pfun-of-alist* [] = $\{\}_p$
⟨proof⟩

lemma *pfun-of-alist-Cons* [*simp*]: *pfun-of-alist* (p # *ps*) = *pfun-of-alist ps*(*fst* $p \mapsto$ *snd* $p)_p$
⟨proof⟩

lemma *dom-pfun-alist* [*simp*, *code*]: *pdom* (*pfun-of-alist xs*) = *set* (*map fst xs*)
⟨proof⟩

lemma *ran-pfun-alist* [*simp*, *code*]: *pran* (*pfun-of-alist xs*) = *set* (*remdups* (*map snd* (*AList.clearjunk xs*)))
⟨*proof*⟩

lemma *map-graph-map-of*: *map-graph* (*map-of xs*) = *set* (*AList.clearjunk xs*)
⟨*proof*⟩

lemma *pfun-graph-alist* [*code*]: *pfun-graph* (*pfun-of-alist xs*) = *set* (*AList.clearjunk xs*)
⟨*proof*⟩

lemma *empty-pfun-alist* [*code*]: $\{\}_p$ = *pfun-of-alist* []
⟨*proof*⟩

lemma *update-pfun-alist* [*code*]: *pfun-upd* (*pfun-of-alist xs*) *k v* = *pfun-of-alist* (*AList.update k v xs*)
⟨*proof*⟩

lemma *apply-pfun-alist* [*code*]:
pfun-app (*pfun-of-alist xs*) *k* = (*if k* ∈ *set* (*map fst xs*) *then the* (*map-of xs k*) *else undefined*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *map-pfun-alist* [*code*]:
map-pfun f (*pfun-of-alist m*) = *pfun-of-alist* (*map* (λ (*k*, *v*). (*k*, *f v*)) *m*)
⟨*proof*⟩

lemma *map-pfun-of-map* [*code*]: *map-pfun f* (*pfun-of-map g*) = *pfun-of-map* (λ *x*. *map-option f* (*g x*))
⟨*proof*⟩

lemma *pdom-res-alist* [*code*]:
$A \lhd_p$ (*pfun-of-alist m*) = *pfun-of-alist* (*AList.restrict A m*)
⟨*proof*⟩

lemma *pran-res-alist-distinct*:
distinct (*map fst xs*) ⟹ *pfun-of-alist xs* $\rhd_p$ *A* = *pfun-of-alist* (*filter* (λ(*k*, *v*). *v* ∈ *A*) *xs*)
⟨*proof*⟩

lemma *pran-res-alist* [*code*]: *pfun-of-alist xs* $\rhd_p$ *A* = *pfun-of-alist* (*filter* (λ(*k*, *v*). *v* ∈ *A*) (*AList.clearjunk xs*))
⟨*proof*⟩

lemma *pdom-res-set-map* [*code*]:
set xs $\lhd_p$ (*pfun-of-map m*) = *pfun-of-alist* (*map* (λ *x*. (*x*, *the* (*m x*))) (*filter* (λ *x*. *m x* ≠ *None*) *xs*))
⟨*proof*⟩

lemma *plus-pfun-alist* [*code*]: *pfun-of-alist f* ⊕ *pfun-of-alist g* = *pfun-of-alist* (*g* • *f*)
⟨*proof*⟩

lemma *pfun-entries-alist* [*code*]: *pfun-entries* (*set ks*) *f* = *pfun-of-alist* (*map* (λ *k*. (*k*, *f k*)) *ks*)
⟨*proof*⟩

lemma *pdom-res-entries-alist* [*code*]:
A $\lhd_p$ *pfun-entries* (*set bs*) *f* =
pfun-of-alist (*map* (λ *k*. (*k*, *f k*)) (*filter* (λ*x*. *x* ∈ *A*) *bs*))
⟨*proof*⟩

lemma *pfun-alist-oplus-map* [*code*]:
pfun-of-alist xs ⊕ *pfun-of-map f* = *pfun-of-map* (λ *k*. *case f k of None* ⇒ *map-of xs k* | *Some v* ⇒ *Some v*)
⟨*proof*⟩

lemma *pfun-map-oplus-alist* [*code*]:
pfun-of-map f ⊕ *pfun-of-alist xs* = *pfun-of-map* (λ *k*. *if k* ∈ *set* (*map fst xs*) *then map-of xs k else f k*)
⟨*proof*⟩

lemma *pfun-singleton-alist* [*code*]: *pfun-singleton* (*pfun-of-alist* [(*k*, *v*)]) = *True*
⟨*proof*⟩

lemma *dest-pfsingle-alist* [*code*]: *dest-pfsingle* (*pfun-of-alist* [(*k*, *v*)]) = (*k*, *v*)
⟨*proof*⟩

Adapted from Mapping theory

lemma *ptabulate-alist* [*code*]: *ptabulate ks f* = *pfun-of-alist* (*map* (λ*k*. (*k*, *f k*)) *ks*)
⟨*proof*⟩

lemma *pcombine-alist* [*code*]:
pcombine f (*pfun-of-alist xs*) (*pfun-of-alist ys*) =
ptabulate (*remdups* (*map fst xs* • *map fst ys*))
(λ*x*. *the* (*combine-options f* (*map-of xs x*) (*map-of ys x*)))
⟨*proof*⟩

lemma *pfun-comp-alist* [*code*]: *pfun-of-alist ys* $\circ_p$ *pfun-of-alist xs* = *pfun-of-alist* (*AList.compose xs ys*)
⟨*proof*⟩

lemma *equal-pfun* [*code*]:
  HOL.equal (*pfun-of-alist xs*) (*pfun-of-alist ys*) $\longleftrightarrow$
    (*let ks* = *map fst xs*; *ls* = *map fst ys*
     in ($\forall$ *l*$\in$*set ls*. *l* $\in$ *set ks*) $\wedge$ ($\forall$ *k*$\in$*set ks*. *k* $\in$ *set ls* $\wedge$ *map-of xs k* = *map-of ys k*))
  ⟨*proof*⟩

lemma *set-inter-Collect*: *set xs* $\cap$ *Collect P* = *set* (*filter P xs*)
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

lemma *pabs-coset* [*code*]:
  pabs (*List.coset A*) *P f* = *pfun-of-map* (λ *x*. *if x* $\in$ *List.coset A* $\wedge$ *P x then Some* (*f x*) *else None*)
  ⟨*proof*⟩

lemma *pfun-app-of-map* [*code*]: *pfun-app* (*pfun-of-map f*) *x* = *the* (*f x*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *pabs-basic-pfun-entries* [*code-unfold*]: (λ *x* $\cdot$ *f x*) = *pfun-entries* (*List.coset* []) *f*
  ⟨*proof*⟩

declare *pdom-pfun-entries* [*code*]

lemma *pfun-app-entries* [*code*]: *pfun-app* (*pfun-entries A f*) *x* = (*if x* $\in$ *A then f x else undefined*)
  ⟨*proof*⟩

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* $\nrightarrow$ '*b*) $\Rightarrow$ ('*k* $\times$ ('*a* $\times$ '*b*)) *list* **where**
pfuse-alist [] *f* = [] |
pfuse-alist ((*k*, *v*) # *ps*) *f* =

$(if\ k \in pdom\ f\ then\ (k,\ (v,\ pfun\text{-}app\ f\ k))\ \#\ pfuse\text{-}alist\ ps\ f\ else\ pfuse\text{-}alist\ ps\ f)$

lemma *pfuse-pfun-of-alist-aux*:
pfuse (*pfun-of-alist xs*) *g* = *pfun-of-alist* (*pfuse-alist xs g*)
⟨*proof*⟩

lemma *pfuse-pfun-of-alist* [*code*]:
pfuse (*pfun-of-alist xs*) *g* = *pfun-of-alist* (*pfuse-alist* (*AList.clearjunk xs*) *g*)
⟨*proof*⟩

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*
$\langle proof \rangle$

lemma *pinj-eq-pfun*: $f = g \longleftrightarrow$ *pfun-of-pinj* $f =$ *pfun-of-pinj* g
$\langle proof \rangle$

lemma *pfun-inj-pinj* [*simp*]: *pfun-inj* (*pfun-of-pinj* f)
$\langle proof \rangle$

type-notation *pinj* (**infixr** $\rightarrowtail\!\!\!\!\!\!\leftrightarrow$ *1*)

setup-lifting *type-definition-pinj*

lift-definition *pinv* :: $'a \rightarrowtail\!\!\!\!\!\!\leftrightarrow 'b \Rightarrow 'b \rightarrowtail\!\!\!\!\!\!\leftrightarrow 'a$ **is** *pfun-inv*
$\langle proof \rangle$

unbundle *lattice-syntax*

instantiation *pinj* :: (*type*, *type*) *bot*
begin
lift-definition *bot-pinj* :: $('a, 'b)$ *pinj* **is** $\bot$
$\langle proof \rangle$
instance $\langle proof \rangle$
end

abbreviation *pinj-empty* :: $('a, 'b)$ *pinj* $(\{\}_\varrho)$ **where** $\{\}_\varrho \equiv \bot$

lift-definition *pinj-app* :: $('a, 'b)$ *pinj* $\Rightarrow 'a \Rightarrow 'b$ (-'(-')$_\varrho$ [*999*,*0*] *999*)
is *pfun-app* $\langle proof \rangle$

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 $\lambda\, f\, k\, v.$ *pfun-upd* $(f \rhd_p (- \{v\}))\ k\ v$
$\langle proof \rangle$

lift-definition *pidom* :: $'a \rightarrowtail\!\!\!\!\!\!\leftrightarrow 'b \Rightarrow 'a$ *set* **is** *pdom* $\langle proof \rangle$

lift-definition *piran* :: $'a \rightarrowtail\!\!\!\!\!\!\leftrightarrow 'b \Rightarrow 'b$ *set* **is** *pran* $\langle proof \rangle$

lift-definition *pinj-dres* :: $'a$ *set* $\Rightarrow ('a, 'b)$ *pinj* $\Rightarrow ('a, 'b)$ *pinj* (**infixr** $\lhd_\varrho$ *85*) **is**
pdom-res
$\langle proof \rangle$

lift-definition *pinj-rres* :: $('a, 'b)$ *pinj* $\Rightarrow 'b$ *set* $\Rightarrow ('a, 'b)$ *pinj* (**infixl** $\rhd_\varrho$ *86*) **is**
pran-res

$\langle proof \rangle$

lift-definition *pinj-comp* :: $'b \rightarrowtail\!\!\!\!\rightarrow 'c \Rightarrow 'a \rightarrowtail\!\!\!\!\rightarrow 'b \Rightarrow 'a \rightarrowtail\!\!\!\!\rightarrow 'c$ (**infixl** $\circ_\varrho$ *55*) **is** $(\circ_p)$
$\langle proof \rangle$

syntax
-PinjUpd :: [(*'a*, *'b*) *pinj*, *maplets*] => (*'a*, *'b*) *pinj* (-'(-')$_\varrho$ [*900*,*0*]*900*)
-Pinj :: *maplets* => (*'a*, *'b*) *pinj* ((*1*{-}$_\varrho$))

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*]: $(f(k \mapsto v)_\varrho)(x)_\varrho = (if\ (k = x)\ then\ v\ else\ (f \rhd_\varrho (-\{v\}))\ (x)_\varrho)$
$\langle proof \rangle$

lemma *pinj-eq-iff*: $f = g \longleftrightarrow (pidom(f) = pidom(g) \wedge (\forall\ x{\in}pidom(f).\ f(x)_\varrho = g(x)_\varrho))$
$\langle proof \rangle$

lemma *pinv-pempty* [*simp*]: $pinv\ \{\}_\varrho = \{\}_\varrho$
$\langle proof \rangle$

lemma *pinv-pinj-upd* [*simp*]: $pinv\ (f(x \mapsto y)_\varrho) = (pinv\ ((-\{x\}) \lhd_\varrho f))(y \mapsto x)_\varrho$
$\langle proof \rangle$

lemma *pinv-pinv*: $pinv\ (pinv\ f) = f$
$\langle proof \rangle$

lemma *pinv-pcomp*: $pinv\ (f \circ_\varrho g) = pinv\ g \circ_\varrho pinv\ f$
$\langle proof \rangle$

lemmas *pidom-empty* [*simp*] = *pdom-zero*[*Transfer.transferred*]
lemma *piran-zero* [*simp*]: $piran\ \{\}_\varrho = \{\}$ $\langle proof \rangle$

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*: $A \lhd_\varrho f(k \mapsto v)_\varrho = (if\ k \in A\ then\ (A \lhd_\varrho f)(k \mapsto v)_\varrho\ else\ A \lhd_\varrho (f \rhd_\varrho (-\ \{v\})))$
$\langle proof \rangle$

lemma *piran-res-upd*: $f(x \mapsto v)_\varrho \rhd_\varrho A = (\mathit{if}\ v \in A\ \mathit{then}\ (f \rhd_\varrho A)(x \mapsto v)_\varrho\ \mathit{else}\ ((-\ \{x\}) \lhd_\varrho f) \rhd_\varrho A)$
$\langle proof \rangle$

lemma *pinj-upd-with-dres-rres*: $((-\{x\}) \lhd_\varrho f \rhd_\varrho (-\{y\}))(x \mapsto y)_\varrho = f(x \mapsto y)_\varrho$
$\langle proof \rangle$

lemma *pidres-twice*: $A \lhd_\varrho B \lhd_\varrho f = (A \cap B) \lhd_\varrho f$
$\langle proof \rangle$

lemma *pidres-commute*: $A \lhd_\varrho B \lhd_\varrho f = B \lhd_\varrho A \lhd_\varrho f$
$\langle proof \rangle$

lemma *pidres-rres-commute*: $A \lhd_\varrho (P \rhd_\varrho B) = (A \lhd_\varrho P) \rhd_\varrho B$
$\langle proof \rangle$

lemma *pirres-twice*: $f \rhd_\varrho A \rhd_\varrho B = f \rhd_\varrho (A \cap B)$
$\langle proof \rangle$

lemma *pirres-commute*: $f \rhd_\varrho A \rhd_\varrho B = f \rhd_\varrho B \rhd_\varrho A$
$\langle proof \rangle$

lemma *pidom-upd*: $\mathit{pidom}\ (f(k \mapsto v)_\varrho) = \mathit{insert}\ k\ (\mathit{pidom}\ (f \rhd_\varrho (-\ \{v\})))$
$\langle proof \rangle$

lemma *f-pinv-f-apply*: $x \in \mathit{pran}\ (\mathit{pfun\text{-}of\text{-}pinj}\ f) \Longrightarrow (\mathit{pfun\text{-}of\text{-}pinj}\ f)(\mathit{pfun\text{-}of\text{-}pinj}\ (\mathit{pinv}\ f)\ (x)_p)_p = x$
$\langle proof \rangle$

fun *pinj-of-alist* :: $('a \times 'b)\ \mathit{list} \Rightarrow 'a \rightarrowtail\!\!\!\!\rightarrow 'b$ **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*]: $\{\}_\varrho = \mathit{pinj\text{-}of\text{-}alist}\ []$
$\langle proof \rangle$

lemma *pinj-upd-alist* [*code*]: $(\mathit{pinj\text{-}of\text{-}alist}\ xs)(k \mapsto v)_\varrho = \mathit{pinj\text{-}of\text{-}alist}\ ((k, v)\ \#\ xs)$
$\langle proof \rangle$

context begin

Injective associative lists

definition *ialist* :: $('a \times 'b)\ \mathit{list} \Rightarrow \mathit{bool}$ **where**
$\mathit{ialist}\ xs = (\mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ xs) \wedge \mathit{distinct}\ (\mathit{map}\ \mathit{snd}\ xs))$

Remove pairs where either the key or value appeared in a previous pair

qualified fun *clearjunk* :: $('a \times 'b)\ \mathit{list} \Rightarrow ('a \times 'b)\ \mathit{list}$ **where**

$clearjunk\ [] = []\ |$
$clearjunk\ (p\#ps) = p\ \#\ filter\ (\lambda\ (k',\ v').\ k' \neq fst\ p \wedge v' \neq snd\ p)\ (clearjunk\ ps)$

lemma *ialist-clearjunk*: $ialist\ (clearjunk\ xs)$
  $\langle proof \rangle$

lemma *ialist-clearjunk-fp*: $ialist\ xs \Longrightarrow clearjunk\ xs = xs$
  $\langle proof \rangle$

lemma *clearjunk-idem* [*simp*]: $clearjunk\ (clearjunk\ xs) = clearjunk\ xs$
  $\langle proof \rangle$

lemma *pinj-of-alist-ndres*: $k \notin fst\ `\ set\ xs \Longrightarrow (-\{k\}) \lhd_\varrho (pinj\text{-}of\text{-}alist\ xs) = pinj\text{-}of\text{-}alist\ xs$
  $\langle proof \rangle$

lemma *pinj-of-alist-nrres*: $v \notin snd\ `\ set\ xs \Longrightarrow (pinj\text{-}of\text{-}alist\ xs) \rhd_\varrho (-\ \{v\}) = pinj\text{-}of\text{-}alist\ xs$
  $\langle proof \rangle$

lemma *pidom-ialist*: $ialist\ xs \Longrightarrow pidom\ (pinj\text{-}of\text{-}alist\ xs) = set\ (map\ fst\ xs)$
  $\langle proof \rangle$

lemma *pinj-of-alist-filter-as-dres-rres*:
  $ialist\ xs \Longrightarrow pinj\text{-}of\text{-}alist\ (filter\ (\lambda(k',\ v').\ k' \neq fst\ p \wedge v' \neq snd\ p)\ xs) = (-\{fst\ p\}) \lhd_\varrho pinj\text{-}of\text{-}alist\ xs \rhd_\varrho (-\{snd\ p\})$
  $\langle proof \rangle$

lemma *pinj-of-alist-clearjunk*: $pinj\text{-}of\text{-}alist\ (clearjunk\ xs) = pinj\text{-}of\text{-}alist\ xs$
  $\langle proof \rangle$

lemma *pinv-pinj-of-ialist*:
  $ialist\ xs \Longrightarrow pinv\ (pinj\text{-}of\text{-}alist\ xs) = pinj\text{-}of\text{-}alist\ (map\ (\lambda\ (x,\ y).\ (y,\ x))\ xs)$
  $\langle proof \rangle$

lemma *pfun-of-ialist*: $ialist\ xs \Longrightarrow pfun\text{-}of\text{-}pinj\ (pinj\text{-}of\text{-}alist\ xs) = pfun\text{-}of\text{-}alist\ xs$
  $\langle proof \rangle$

declare *clearjunk.simps* [*simp del*]

end

lemma *pinv-pinj-of-alist* [*code*]: $pinv\ (pinj\text{-}of\text{-}alist\ xs) = pinj\text{-}of\text{-}alist\ (map\ (\lambda\ (x,\ y).\ (y,\ x))\ (Partial\text{-}Inj.clearjunk\ xs))$
  $\langle proof \rangle$

lemma *pfun-of-pinj-of-alist* [*code*]:

pfun-of-pinj (*pinj-of-alist xs*) = *pfun-of-alist* (*Partial-Inj.clearjunk xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

type-notation *ffun* (**infixr** ⇸⇸ *1*)

setup-lifting *type-definition-ffun*

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

lift-definition *ffun-upd* :: '*a* ⇸⇸ '*b* ⇒ '*a* ⇒ '*b* ⇒ '*a* ⇸⇸ '*b* **is** *pfun-upd* ⟨*proof*⟩

lift-definition *fdom* :: '*a* ⇸⇸ '*b* ⇒ '*a set* **is** *pdom* ⟨*proof*⟩

lift-definition *fran* :: '*a* ⇸⇸ '*b* ⇒ '*b set* **is** *pran* ⟨*proof*⟩

lift-definition *ffun-comp* :: ('*b*, '*c*) *ffun* ⇒ '*a* ⇸⇸ '*b* ⇒ ('*a*, '*c*) *ffun* (**infixl** $\circ_f$ *55*)
is *pfun-comp* ⟨*proof*⟩

lift-definition *ffun-member* :: '*a* × '*b* ⇒ '*a* ⇸⇸ '*b* ⇒ *bool* (**infix** $\in_f$ *50*) **is** ($\in_p$)
⟨*proof*⟩

lift-definition *fdom-res* :: '*a set* ⇒ '*a* ⇸⇸ '*b* ⇒ '*a* ⇸⇸ '*b* (**infixr** $\lhd_f$ *85*)
is *pdom-res* ⟨*proof*⟩

abbreviation *fdom-nres* (**infixr** $-\lhd_f$ *85*) **where** *fdom-nres A P* ≡ (− *A*) $\lhd_f$ *P*

lift-definition *fran-res* :: '*a* ⇸⇸ '*b* ⇒ '*b set* ⇒ '*a* ⇸⇸ '*b* (**infixl** $\rhd_f$ *85*)
is *pran-res* ⟨*proof*⟩

abbreviation *fran-nres* (**infixr** $\rhd_f-$ *66*) **where** *fran-nres P A* ≡ *P* $\rhd_f$ (− *A*)

lift-definition *ffun-graph* :: '*a* ⇸⇸ '*b* ⇒ ('*a* × '*b*) *set* **is** *pfun-graph* ⟨*proof*⟩

lift-definition *graph-ffun* :: $('a \times 'b)$ *set* $\Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b$ **is**
λ *R. if* (*finite* (*Domain R*)) *then graph-pfun R else pempty*
$\langle proof \rangle$

unbundle *lattice-syntax*

lift-definition *ffun-entries* :: $'a$ *set* $\Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b$
is λ *A f. if* (*finite A*) *then pfun-entries A f else* $\perp$ $\langle proof \rangle$

instantiation *ffun* :: (*type*, *type*) *bot*
begin
lift-definition *bot-ffun* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b$ **is** $\perp$ $\langle proof \rangle$
instance $\langle proof \rangle$
end

abbreviation *fempty* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b$ $(\{\}_f)$
where *fempty* $\equiv \perp$

instantiation *ffun* :: (*type*, *type*) *oplus*
begin
lift-definition *oplus-ffun* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b$ **is** $(\oplus)$ $\langle proof \rangle$
instance $\langle proof \rangle$
end

instantiation *ffun* :: (*type*, *type*) *minus*
begin
lift-definition *minus-ffun* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b$ **is** $(-)$
$\langle proof \rangle$
instance $\langle proof \rangle$
end

instantiation *ffun* :: (*type*, *type*) *inf*
begin
lift-definition *inf-ffun* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b$ **is** *inf*
$\langle proof \rangle$
instance $\langle proof \rangle$
end

abbreviation *ffun-inter* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b$ (**infixl** $\cap_f$ *80*)
where *ffun-inter* $\equiv$ *inf*

instantiation *ffun* :: (*type*, *type*) *order*
begin
lift-definition *less-eq-ffun* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow$ *bool* **is**
$\lambda\, f\, g.\ f \subseteq_p g$ $\langle proof \rangle$
lift-definition *less-ffun* :: $'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow 'a \Rightarrow\!\!\!\!\!\!+\; 'b \Rightarrow$ *bool* **is**
$\lambda\, f\, g.\ f < g$ $\langle proof \rangle$
instance

$\langle proof \rangle$
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*
$\langle proof \rangle$

lemma *ffun-subset-eq-least* [*simp*]:
$\{\}_f \subseteq_f f$
$\langle proof \rangle$

instantiation *ffun* :: (*type*, *type*) *size*
begin

definition *size-ffun* :: $'a \nrightarrow 'b \Rightarrow nat$ **where**
[*simp*]: *size-ffun f* = *card* (*fdom f*)

instance $\langle proof \rangle$

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*: $f \circ_f (g \circ_f h) = (f \circ_f g) \circ_f h$
$\langle proof \rangle$

lemma *ffun-override-dist-comp*:
$(f \oplus g) \circ_f h = (f \circ_f h) \oplus (g \circ_f h)$
$\langle proof \rangle$

lemma *ffun-minus-unit* [*simp*]:
fixes $f :: 'a \nrightarrow 'b$
shows $f - \bot = f$

⟨*proof*⟩

lemma *ffun-minus-zero* [*simp*]:
fixes f :: $'a \nrightarrow\!\!\!\!\rightarrow 'b$
shows $\bot - f = \bot$
⟨*proof*⟩

lemma *ffun-minus-self* [*simp*]:
fixes f :: $'a \nrightarrow\!\!\!\!\rightarrow 'b$
shows $f - f = \bot$
⟨*proof*⟩

instantiation *ffun* :: (*type*, *type*) *override*
begin
lift-definition *compatible-ffun* :: $'a \nrightarrow\!\!\!\!\rightarrow 'b \Rightarrow 'a \nrightarrow\!\!\!\!\rightarrow 'b \Rightarrow bool$ **is** *compatible* ⟨*proof*⟩

instance
⟨*proof*⟩
end

lemma *compatible-ffun-alt-def*: $R \ \#\# \ S = ((fdom\ R) \lhd_f S = (fdom\ S) \lhd_f R)$
⟨*proof*⟩

lemma *ffun-indep-compat*: $fdom(f) \cap fdom(g) = \{\} \Longrightarrow f \ \#\# \ g$
⟨*proof*⟩

lemma *ffun-override-commute*:
$fdom(f) \cap fdom(g) = \{\} \Longrightarrow f \oplus g = g \oplus f$
⟨*proof*⟩

lemma *ffun-minus-override-commute*:
$fdom(g) \cap fdom(h) = \{\} \Longrightarrow (f - g) \oplus h = (f \oplus h) - g$
⟨*proof*⟩

lemma *ffun-override-minus*:
$f \subseteq_f g \Longrightarrow (g - f) \oplus f = g$
⟨*proof*⟩

lemma *ffun-minus-common-subset*:
$[\![h \subseteq_f f;\ h \subseteq_f g]\!] \Longrightarrow (f - h = g - h) = (f = g)$
⟨*proof*⟩

lemma *ffun-minus-override*:
$fdom(f) \cap fdom(g) = \{\} \Longrightarrow (f \oplus g) - g = f$
⟨*proof*⟩

lemma *ffun-override-pos*: $x \oplus y = \{\}_f \Longrightarrow x = \{\}_f$
⟨*proof*⟩

lemma *ffun-le-override*: $fdom\ x \cap fdom\ y = \{\} \Longrightarrow x \leq x \oplus y$
  $\langle proof \rangle$

16.3 Membership, application, and update

lemma *ffun-ext*: $\llbracket \bigwedge x\ y.\ (x,\ y) \in_f f \longleftrightarrow (x,\ y) \in_f g \rrbracket \Longrightarrow f = g$
  $\langle proof \rangle$

lemma *ffun-member-alt-def*:
  $(x,\ y) \in_f f \longleftrightarrow (x \in fdom\ f \wedge f(x)_f = y)$
  $\langle proof \rangle$

lemma *ffun-member-override*:
  $(x,\ y) \in_f f \oplus g \longleftrightarrow ((x \notin fdom(g) \wedge (x,\ y) \in_f f) \vee (x,\ y) \in_f g)$
  $\langle proof \rangle$

lemma *ffun-member-minus*:
  $(x,\ y) \in_f f - g \longleftrightarrow (x,\ y) \in_f f \wedge (\neg\ (x,\ y) \in_f g)$
  $\langle proof \rangle$

lemma *ffun-app-upd-1* [*simp*]: $x = y \Longrightarrow (f(x \mapsto v)_f)(y)_f = v$
  $\langle proof \rangle$

lemma *ffun-app-upd-2* [*simp*]: $x \neq y \Longrightarrow (f(x \mapsto v)_f)(y)_f = f(y)_f$
  $\langle proof \rangle$

lemma *ffun-upd-ext* [*simp*]: $x \in fdom(f) \Longrightarrow f(x \mapsto f(x)_f)_f = f$
  $\langle proof \rangle$

lemma *ffun-app-add* [*simp*]: $x \in fdom(g) \Longrightarrow (f \oplus g)(x)_f = g(x)_f$
  $\langle proof \rangle$

lemma *ffun-upd-add* [*simp*]: $f \oplus g(x \mapsto v)_f = (f \oplus g)(x \mapsto v)_f$
  $\langle proof \rangle$

lemma *ffun-upd-twice* [*simp*]: $f(x \mapsto u,\ x \mapsto v)_f = f(x \mapsto v)_f$
  $\langle proof \rangle$

lemma *ffun-upd-comm*:
  assumes $x \neq y$
  shows $f(y \mapsto u,\ x \mapsto v)_f = f(x \mapsto v,\ y \mapsto u)_f$
  $\langle proof \rangle$

lemma *ffun-upd-add-left* [*simp*]: $x \notin fdom(g) \Longrightarrow f(x \mapsto v)_f \oplus g = (f \oplus g)(x \mapsto v)_f$
  $\langle proof \rangle$

lemma *ffun-app-add'* [*simp*]: $\llbracket\ e \in fdom\ f;\ \ e \notin fdom\ g\ \rrbracket \Longrightarrow (f \oplus g)(e)_f = f(e)_f$

$\langle proof \rangle$

lemma *ffun-upd-comm-linorder* [*simp*]:
fixes $x\ y :: 'a :: linorder$
assumes $x < y$
shows $f(y \mapsto u,\ x \mapsto v)_f = f(x \mapsto v,\ y \mapsto u)_f$
$\langle proof \rangle$

lemma *ffun-app-minus* [*simp*]: $x \notin fdom\ g \Longrightarrow (f\ -\ g)(x)_f = f(x)_f$
$\langle proof \rangle$

lemma *ffun-upd-minus* [*simp*]:
$x \notin fdom\ g \Longrightarrow (f\ -\ g)(x \mapsto v)_f = (f(x \mapsto v)_f\ -\ g)$
$\langle proof \rangle$

lemma *fdom-member-minus-iff* [*simp*]:
$x \notin fdom\ g \Longrightarrow x \in fdom(f\ -\ g) \longleftrightarrow x \in fdom(f)$
$\langle proof \rangle$

lemma *fsubseteq-ffun-upd1* [*intro*]:
$\llbracket\ f \subseteq_f\ g;\ \ x \notin fdom(g)\ \rrbracket \Longrightarrow f \subseteq_f\ g(x \mapsto v)_f$
$\langle proof \rangle$

lemma *fsubseteq-ffun-upd2* [*intro*]:
$\llbracket\ f \subseteq_f\ g;\ \ x \notin fdom(f)\ \rrbracket \Longrightarrow f \subseteq_f\ g(x \mapsto v)_f$
$\langle proof \rangle$

lemma *psubseteq-pfun-upd3* [*intro*]:
$\llbracket\ f \subseteq_f\ g;\ \ g(x)_f = v\ \rrbracket \Longrightarrow f \subseteq_f\ g(x \mapsto v)_f$
$\langle proof \rangle$

lemma *fsubseteq-dom-subset*:
$f \subseteq_f\ g \Longrightarrow fdom(f) \subseteq fdom(g)$
$\langle proof \rangle$

lemma *fsubseteq-ran-subset*:
$f \subseteq_f\ g \Longrightarrow fran(f) \subseteq fran(g)$
$\langle proof \rangle$

lemma *fdom-res-apply* [*simp*]:
$x \in A \Longrightarrow (A \lhd_f\ f)(x)_f = f(x)_f$
$\langle proof \rangle$

16.4 Domain laws

lemma *fdom-finite* [*simp*]: $finite(fdom(f))$
$\langle proof \rangle$

lemma *fdom-zero* [*simp*]: $fdom\ \bot = \{\}$

$\langle proof \rangle$

lemma *fdom-plus* [*simp*]: $fdom\ (f \oplus g) = fdom\ f \cup fdom\ g$
$\langle proof \rangle$

lemma *fdom-inter*: $fdom\ (f \cap_f g) \subseteq fdom\ f \cap fdom\ g$
$\langle proof \rangle$

lemma *fdom-comp* [*simp*]: $fdom\ (g \circ_f f) = fdom\ (f \rhd_f fdom\ g)$
$\langle proof \rangle$

lemma *fdom-upd* [*simp*]: $fdom\ (f(k \mapsto v)_f) = insert\ k\ (fdom\ f)$
$\langle proof \rangle$

lemma *fdom-fdom-res* [*simp*]: $fdom\ (A \lhd_f f) = A \cap fdom(f)$
$\langle proof \rangle$

lemma *ffun-fdom-antires-upd* [*simp*]:
$k \in A \Longrightarrow ((-\ A) \lhd_f m)(k \mapsto v)_f = ((-\ (A - \{k\})) \lhd_f m)(k \mapsto v)_f$
$\langle proof \rangle$

lemma *fdom-res-UNIV* [*simp*]: $UNIV \lhd_f f = f$
$\langle proof \rangle$

lemma *fdom-graph-ffun* [*simp*]:
$[\![functional\ R;\ finite\ (Domain\ R)]\!] \Longrightarrow fdom\ (graph\text{-}ffun\ R) = Domain\ R$
$\langle proof \rangle$

lemma *pdom-pfun-of* [*simp*]: $pdom\ (pfun\text{-}of\ f) = fdom\ f$
$\langle proof \rangle$

lemma *finite-pdom-ffun* [*simp*]: $finite\ (pdom\ (pfun\text{-}of\ f))$
$\langle proof \rangle$

16.5 Range laws

lemma *fran-zero* [*simp*]: $fran\ \bot = \{\}$
$\langle proof \rangle$

lemma *fran-upd* [*simp*]: $fran\ (f(k \mapsto v)_f) = insert\ v\ (fran\ ((-\ \{k\}) \lhd_f f))$
$\langle proof \rangle$

lemma *fran-fran-res* [*simp*]: $fran\ (f \rhd_f A) = fran(f) \cap A$
$\langle proof \rangle$

lemma *fran-comp* [*simp*]: $fran\ (g \circ_f f) = fran\ (fran\ f \lhd_f g)$
$\langle proof \rangle$

16.6 Domain restriction laws

lemma *fdom-res-zero* [*simp*]: $A \lhd_f \{\}_f = \{\}_f$
$\langle proof \rangle$

lemma *fdom-res-empty* [*simp*]:
$(\{\} \lhd_f f) = \{\}_f$
$\langle proof \rangle$

lemma *fdom-res-fdom* [*simp*]:
$fdom(f) \lhd_f f = f$
$\langle proof \rangle$

lemma *fdom-res-upd-in* [*simp*]:
$k \in A \Longrightarrow A \lhd_f f(k \mapsto v)_f = (A \lhd_f f)(k \mapsto v)_f$
$\langle proof \rangle$

lemma *fdom-res-upd-out* [*simp*]:
$k \notin A \Longrightarrow A \lhd_f f(k \mapsto v)_f = A \lhd_f f$
$\langle proof \rangle$

lemma *fdom-res-override* [*simp*]: $A \lhd_f (f \oplus g) = (A \lhd_f f) \oplus (A \lhd_f g)$
$\langle proof \rangle$

lemma *fdom-res-minus* [*simp*]: $A \lhd_f (f - g) = (A \lhd_f f) - g$
$\langle proof \rangle$

lemma *fdom-res-swap*: $(A \lhd_f f) \rhd_f B = A \lhd_f (f \rhd_f B)$
$\langle proof \rangle$

lemma *fdom-res-twice* [*simp*]: $A \lhd_f (B \lhd_f f) = (A \cap B) \lhd_f f$
$\langle proof \rangle$

lemma *fdom-res-comp* [*simp*]: $A \lhd_f (g \circ_f f) = \ g \circ_f (A \lhd_f f)$
$\langle proof \rangle$

lemma *ffun-split-domain*: $A \lhd_f f \oplus (-\ A) \lhd_f f = f$
$\langle proof \rangle$

16.7 Range restriction laws

lemma *fran-res-zero* [*simp*]: $\{\}_f \rhd_f A = \{\}_f$
$\langle proof \rangle$

lemma *fran-res-upd-1* [*simp*]: $v \in A \Longrightarrow f(x \mapsto v)_f \rhd_f A = (f \rhd_f A)(x \mapsto v)_f$
$\langle proof \rangle$

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

lemma *fran-res-override*: $(f \oplus g) \rhd_f A \subseteq_f (f \rhd_f A) \oplus (g \rhd_f A)$
⟨*proof*⟩

16.8 Graph laws

lemma *ffun-graph-inv*: *graph-ffun* (*ffun-graph* f) = f
⟨*proof*⟩

lemma *ffun-graph-zero*: *ffun-graph* $\bot = \{\}$
⟨*proof*⟩

lemma *ffun-graph-minus*: *ffun-graph* $(f - g)$ = *ffun-graph* f − *ffun-graph* g
⟨*proof*⟩

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

16.9 Conversions

lift-definition *list-ffun* :: *'a list* ⇒ *nat* ⇸ *'a* **is**
list-pfun ⟨*proof*⟩

lemma *fdom-list-ffun* [*simp*]: *fdom* (*list-ffun xs*) = {*1..length xs*}
⟨*proof*⟩

lemma *fran-list-ffun* [*simp*]: *fran* (*list-ffun xs*) = *set xs*
⟨*proof*⟩

lemma *ffun-app-list-ffun*: ⟦ *0* < *i*; *i* < *length xs* ⟧ ⟹ (*list-ffun xs*)$(i)_f$ = *xs* ! (*i*
− *1*)
⟨*proof*⟩

lemma *range-list-ffun*: *range list-ffun* = {*f*. ∃ *i*. *fdom*(*f*) = {*1..i*}}
⟨*proof*⟩

16.10 Finite Function Lens

definition *ffun-lens* :: *'a* ⇒ (*'b* ⟹ *'a* ⇸ *'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*)
⟨*proof*⟩

lemma *ffun-lens-src*: $\mathcal{S}_{\textit{ffun-lens } i}$ = {*f*. *i* ∈ *fdom*(*f*)}
⟨*proof*⟩

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** $\trianglelefteq$ *86*)

notation *fran-res* (**infixl** $\rhd$ *86*)
notation *fran-nres* (**infixl** $\trianglerighteq$ *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* $= \{f :: 'a \rightarrowtail\!\!\!\!\!\mapsto 'b.\ finite(pidom(f))\}$
morphisms *pinj-of-finj finj-of-pinj*
⟨*proof*⟩

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

type-notation *finj* (**infixr** $\rightarrowtail\!\!\!\!\!\Mapsto$ *1*)

lift-definition *ffun-of-finj* :: $'a \rightarrowtail\!\!\!\!\!\Mapsto 'b \Rightarrow 'a \twoheadrightarrow\!\!\!\!\!\Mapsto 'b$ **is** $\lambda\ x.$ *pfun-of-pinj* (*pinj-of-finj* *x*)
⟨*proof*⟩

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*
  $\langle proof \rangle$

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**
$\lambda\ f.$ *fun-pfun f* $\langle proof \rangle$

lemma *pfun-of-tfun-mk-tfun* [*simp*]: *pfun-of-tfun* (*mk-tfun f*) = *fun-pfun f*
  $\langle proof \rangle$

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*
$\langle proof \rangle$

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* ⟨*proof*⟩

lift-definition *bnth* :: *'a blist['n::finite]* ⇒ *nat* ⇒ *'a* **is** *nth* ⟨*proof*⟩

lift-definition *bappend* :: *'a blist['m::finite]* ⇒ *'a blist['n::finite]* ⇒ *'a blist['m + 'n]* (**infixr** $\bullet_s$ *65*) **is** *append*
⟨*proof*⟩

lift-definition *bmake* :: *'n itself* ⇒ *'a list* ⇒ *'a blist['n::finite]* **is** λ -. *take CARD('n)*
⟨*proof*⟩

code-datatype *bmake*

lemma *bmake-length-card*:
blength (bmake TYPE('n::finite) xs) = (if length xs ≤ CARD('n) then length xs else CARD('n))
⟨*proof*⟩

lemma *blist-always-bounded*:
length (list-of-blist (bl::'a blist['n::finite])) ≤ CARD('n)
⟨*proof*⟩

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*
⟨*proof*⟩

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 • take CARD('b) ys)
⟨*proof*⟩

instantiation *blist* :: (*type*, *finite*) *equal*
begin

definition *equal-blist* :: *'a blist['b]* ⇒ *'a blist['b]* ⇒ *bool* **where**
equal-blist m1 m2 ⟷ (*list-of-blist m1 = list-of-blist m2*)

instance ⟨*proof*⟩

end

lemma *list-of-blist-code* [*code*]:
list-of-blist (bmake TYPE('n::finite) xs) = take CARD('n) xs

$\langle proof \rangle$

definition *blists* :: *'n::finite itself* ⇒ *'a list* ⇒ (*'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*)
$\langle proof \rangle$

lemma *set-blists-enum-UNIV*: *set* (*blists TYPE*(*'n::finite*) (*enum-class.enum* ::*'a::enum list*)) = *UNIV*
$\langle proof \rangle$

lemma *distinct-blists*: *distinct xs* ⟹ *distinct* (*blists n xs*)
$\langle proof \rangle$

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
$\langle proof \rangle$

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* :: $('a{::}enum \Rightarrow 'b) \Rightarrow ('a \times 'b)$ *list* **where**
tabulate f = *map* $(\lambda\ x.\ (x, f\ x))$ (*rev Enum.enum*)

$\langle ML \rangle$

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*!]: $[\![\ (\mathit{if}\ b\ \mathit{then}\ e\ \mathit{else}\ f) = v;\ [\![\ b;\ e = v\ ]\!] \Longrightarrow P;\ [\![\ \neg\ b;\ f = v\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
$\langle proof \rangle$

bundle *Z-Type-Syntax*
begin

type-notation *bool* ($\mathbb{B}$)
type-notation *nat* ($\mathbb{N}$)
type-notation *int* ($\mathbb{Z}$)
type-notation *rat* ($\mathbb{Q}$)
type-notation *real* ($\mathbb{R}$)

type-notation *set* ($\mathbb{P}$ - [*999*] *999*)

type-notation *tfun* (**infixr** $\rightarrow$ *0*)

notation *Pow* ($\mathbb{P}$)
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* ($\sqsubseteq$) ($\sqsubset$)
$\langle proof \rangle$

lemma *ref-by-trans* [*trans*]: $[\![P \sqsubseteq Q;\ \ Q \sqsubseteq R]\!] \Longrightarrow P \sqsubseteq R$
$\langle proof \rangle$

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 $\langle proof \rangle$

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
$\langle proof \rangle$
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* :: $'f \Rightarrow 'a\ set$

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* :: $'f \Rightarrow 'a\ set$

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* :: $'f \Rightarrow 'g \Rightarrow 'h$

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** $\circ$ *55*)
notation *zcomp* (**infixl** $\circ$ *55*)

end

22.10 Domain restriction and subtraction

consts *dom-res* :: *'a set* $\Rightarrow$ *'r* $\Rightarrow$ *'r* (**infixr** $\lhd$ *85*)

abbreviation *ndres* (**infixr** $⩤$ *85*) **where** *ndres A P* $\equiv$ *CONST dom-res* ($-$ *A*) *P*

adhoc-overloading
  dom-res $\rightleftharpoons$ *rel-domres*
  and *dom-res* $\rightleftharpoons$ *pdom-res*
  and *dom-res* $\rightleftharpoons$ *fdom-res*
  and *dom-res* $\rightleftharpoons$ *pinj-dres*

syntax *-ndres* :: *logic* $\Rightarrow$ *logic* $\Rightarrow$ *logic*
translations *-ndres A P* == *CONST dom-res* ($-$ *A*) *P*

22.11 Range restriction and subtraction

consts *ran-res* :: *'r* $\Rightarrow$ *'a set* $\Rightarrow$ *'r* (**infixl** $\rhd$ *86*)

abbreviation *nrres* (**infixl** $⩥$ *86*) **where** *nrres P A* $\equiv$ *CONST ran-res P* ($-$ *A*)

adhoc-overloading
  ran-res $\rightleftharpoons$ *rel-ranres*
  and *ran-res* $\rightleftharpoons$ *pran-res*
  and *ran-res* $\rightleftharpoons$ *fran-res*
  and *ran-res* $\rightleftharpoons$ *pinj-rres*

22.12 Relational inversion

notation *converse* ((-$^\sim$) [*1000*] *999*)

lemma *relational-inverse*: $r^\sim = \{(p.2,\ p.1) \mid p.\ p \in r\}$
  $\langle proof \rangle$

22.13 Relational image

notation *Image* (-(|-|) [*990*] *990*)

lemma *Image-eq*: *Image r a* $= \{p.2 \mid p.\ p \in r \wedge p.1 \in a\}$
  $\langle proof \rangle$

22.14 Overriding

lemma *dom-override*: $dom\ ((Q :: 'a \leftrightarrow 'b) \oplus R) = (dom\ Q) \cup (dom\ R)$
⟨*proof*⟩

lemma *override-Un*: $dom\ Q \cap dom\ R = \{\} \Longrightarrow Q \oplus R = Q \cup R$
⟨*proof*⟩

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*]: $[f]_{\nrightarrow} = [g]_{\nrightarrow} \longleftrightarrow f = g$
⟨*proof*⟩

lemma *rel-of-pfun-le-iff* [*simp*]: $[f]_{\nrightarrow} \subseteq [g]_{\nrightarrow} \longleftrightarrow f \leq g$
⟨*proof*⟩

lemma *rel-of-pfun-pabs*: $[pabs\ A\ P\ f]_{\nrightarrow} = \{(k, v).\ k \in A \wedge P\ k \wedge v = f\ k\}$
⟨*proof*⟩

lemma *rel-of-pfun-apply* [*simp*]: $[f]_{\nrightarrow}\ x = f\ x$
⟨*proof*⟩

lemma *rel-of-pfun-functional* [*simp*]: *functional* $[f]_{\nrightarrow}$
⟨*proof*⟩

lemma *rel-of-pfun-override* [*simp*]: $[f]_{\nrightarrow} \oplus [g]_{\nrightarrow} = [f \oplus g]_{\nrightarrow}$
⟨*proof*⟩

lemma *rel-of-pfun-comp* [*simp*]: $[f]_{\nrightarrow}\ O\ [g]_{\nrightarrow} = [g \circ_p f]_{\nrightarrow}$
⟨*proof*⟩

lemma *pfun-comp-inv*: $[f \circ_p g]_{\nrightarrow}{}^{\sim} = [f]_{\nrightarrow}{}^{\sim}\ O\ [g]_{\nrightarrow}{}^{\sim}$
⟨*proof*⟩

lemma *rel-of-pfun-dom* [*simp*]: $Domain\ [f]_{\nrightarrow} = pdom\ f$
⟨*proof*⟩

lemma *rel-of-pfun-ran* [*simp*]: $Range\ [f]_{\nrightarrow} = pran\ f$
⟨*proof*⟩

lemma *rel-of-pfun-domres* [*simp*]: $A \lhd [f]_{\nrightarrow} = [A \lhd f]_{\nrightarrow}$
⟨*proof*⟩

lemma *rel-of-pfun-ranres* [*simp*]: $[f]_{\nrightarrow} \rhd A = [f \rhd A]_{\nrightarrow}$
⟨*proof*⟩

lemma *rel-of-pfun-image* [*simp*]: $[f]_{\rightarrowtail\!\!\!\!\leftrightarrow} (\!| A |\!) = pfun\text{-}image\ f\ A$
⟨*proof*⟩

lemma *rel-of-pfun-member-iff* [*simp*]:
$(k,\ v) \in [f]_{\leftrightarrow} \longleftrightarrow (k \in dom\ f \wedge f\ k = v)$
⟨*proof*⟩

lemma *rel-of-pinj-conv* [*simp*]: $[[f]_{\rightarrowtail\leftrightarrow}]_{\leftrightarrow}{}^{-1} = [[pinv\ f]_{\rightarrowtail\leftrightarrow}]_{\leftrightarrow}$
⟨*proof*⟩

lemma *dom-pinv* [*simp*]: $dom\ [pinv\ f]_{\rightarrowtail\leftrightarrow} = ran\ [f]_{\rightarrowtail\leftrightarrow}$
⟨*proof*⟩

lemma *ran-pinv* [*simp*]: $ran\ [pinv\ f]_{\rightarrowtail\leftrightarrow} = dom\ [f]_{\rightarrowtail\leftrightarrow}$
⟨*proof*⟩

lemma *pfun-inj-rel-conv* [*simp*]: $pfun\text{-}inj\ f \Longrightarrow [f]_{\leftrightarrow}{}^{\sim} = [pfun\text{-}inv\ f]_{\leftrightarrow}$
⟨*proof*⟩

end

23 Function Toolkit

theory *Function-Toolkit*
imports *Relation-Toolkit*
begin

23.1 Partial Functions

lemma *partial-functions*: $X \rightarrow_p Y = \{f \in X \leftrightarrow Y.\ \forall\ p \in f.\ \forall\ q \in f.\ p.1 = q.1 \longrightarrow p.2 = q.2\}$
⟨*proof*⟩

notation *size* (#- [*999*] *999*)

instantiation *set* :: (*type*) *size*
begin
definition [*simp*]: *size A* = *card A*
instance ⟨*proof*⟩
end

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

definition *size-pfun* :: $('a \nrightarrow 'b) \Rightarrow nat$ **where**
size-pfun f = *card* (*pfun-graph f*)

instance ⟨*proof*⟩

end

lemma *size-finite-pfun*: *finite* $(pdom\ f) \Longrightarrow \#f = \#(dom\ f)$
  ⟨*proof*⟩

lemma *card-pfun-empty* [*simp*]: $\#\{\}_p = 0$
  ⟨*proof*⟩

lemma *card-pfun-update* [*simp*]: *finite* $(dom\ f) \Longrightarrow \#(f(k \mapsto v)_p) = (if\ (k \in dom\ f)\ then\ \#f\ else\ \#f + 1)$
  ⟨*proof*⟩

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 ⇌ *rel-disjoint* **and**
  disjoint ⇌ *pfun-disjoint* **and**
  disjoint ⇌ *list-disjoint*

23.4 Partitions

consts *partitions* :: $'f \Rightarrow 'a\ set \Rightarrow bool$ (**infix** *partitions 65*)

adhoc-overloading
  partitions ⇌ *rel-partitions* **and**
  partitions ⇌ *pfun-partitions* **and**
  partitions ⇌ *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* :: $'a\ list \Rightarrow nat \leftrightarrow 'a$ ($[\text{-}]_s$) **where**
rel-of-list xs $\equiv$ $[list\text{-}pfun\ xs]_{\nrightarrow}$

abbreviation *seq-nth* $(\text{-}'(\text{-}')_s\ [999,0]\ 999)$ **where**
seq-nth xs i $\equiv$ *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*: $\{i..j\} = \{k :: \mathbb{Z}.\ i \leq k \wedge k \leq j\}$
$\langle proof \rangle$

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* :: $\mathbb{Z} \Rightarrow ('X \leftrightarrow 'X) \Rightarrow ('X \leftrightarrow 'X)$ **where**
$iter\ n\ R = (if\ (n \geq 0)\ then\ R\ \hat{}\hat{}\ (nat\ n)\ else\ (R^{\sim})\ \hat{}\hat{}\ (nat\ n))$

lemma *iter-eqs*:
$iter\ 0\ r = Id$
$n \geq 0 \Longrightarrow iter\ (n + 1)\ r = r\ \mathbf{;}\ (iter\ n\ r)$
$n < 0 \Longrightarrow iter\ (n + 1)\ r = iter\ n\ (r^{\sim})$
$\langle proof \rangle$

25.4 Number of members of a set

lemma *size-rel-of-list*:
$\#xs = length\ xs$
$\langle proof \rangle$

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 $seq\text{-}iff\ [simp]: xs \in seq\ A \longleftrightarrow set\ xs \subseteq A$
$\langle proof \rangle$

lemma $seq\text{-}ffun\text{-}set: range\ list\text{-}ffun = \{f :: \mathbb{N} \twoheadrightarrow 'X.\ dom(f) = \{1..\#f\}\}$
$\langle proof \rangle$

25.8 Non-empty finite sequences

definition $seq_1\ A = seq\ A - \{[]\}$

lemma $seq_1\text{-}iff\ [simp]: xs \in seq_1(A) \longleftrightarrow (xs \in seq\ A \wedge \#xs > 0)$
$\langle proof \rangle$

25.9 Injective sequences

definition $iseq\ A = seq\ A \cap Collect\ distinct$

lemma $iseq\text{-}iff\ [simp]: xs \in iseq(A) \longleftrightarrow (xs \in seq\ A \wedge distinct\ xs)$
$\langle proof \rangle$

25.10 Bounded sequences

definition $bseq :: \mathbb{N} \Rightarrow 'a\ set \Rightarrow 'a\ list\ set\ (bseq[\text{-}])$ **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 $head :: 'a\ list \rightarrow 'a$ **where**
$head = (\lambda\ xs :: 'a\ list \mid \#xs > 0 \cdot hd\ xs)$

lemma $dom\text{-}head: dom\ head = \{xs.\ \#xs > 0\}$
$\langle proof \rangle$

lemma *head-app*: $\#xs > 0 \Longrightarrow head\ xs = hd\ xs$
⟨*proof*⟩

lemma *head-z-def*: $xs \in seq_1(A) \Longrightarrow head\ xs = xs\ 1$
⟨*proof*⟩

25.15 Last of a sequence

definition $slast :: 'a\ list \rightharpoonup 'a$ **where**
$slast = (\lambda\ xs :: 'a\ list \mid \#xs > 0 \cdot List.last\ xs)$

lemma *dom-last*: $dom\ slast = \{xs.\ \#xs > 0\}$
⟨*proof*⟩

lemma *slast-app*: $\#xs > 0 \Longrightarrow slast\ xs = last\ xs$
⟨*proof*⟩

lemma *slast-eq*: $\#s > 0 \Longrightarrow last\ s = s\ (\#s)$
⟨*proof*⟩

25.16 Tail of a sequence

definition $tail :: 'a\ list \rightharpoonup 'a\ list$ **where**
$tail = (\lambda\ xs :: 'a\ list \mid \#xs > 0 \cdot tl\ xs)$

lemma *dom-tail*: $dom\ tail = \{xs.\ \#xs > 0\}$
⟨*proof*⟩

lemma *tail-app*: $\#xs > 0 \Longrightarrow tail\ xs = tl\ xs$
⟨*proof*⟩

25.17 Domain

definition $dom\text{-}seq :: 'a\ list \Rightarrow \mathbb{N}\ set$ **where**
$[simp]$: $dom\text{-}seq\ xs = \{0..<\#xs\}$

adhoc-overloading $dom \rightleftharpoons dom\text{-}seq$

25.18 Range

definition $ran\text{-}seq :: 'a\ list \Rightarrow 'a\ set$ **where**
$[simp]$: $ran\text{-}seq\ xs = set\ xs$

adhoc-overloading $ran \rightleftharpoons ran\text{-}seq$

25.19 Filter

notation *seq-filter* (**infix** $\upharpoonright$ *80*)

lemma *seq-filter-Nil*: [] ↾ *V* = [] ⟨*proof*⟩

lemma *seq-filter-append*: (*s* • *t*) ↾ *V* = (*s* ↾ *V*) • (*t* ↾ *V*)
  ⟨*proof*⟩

lemma *seq-filter-subset-iff*: *ran s* ⊆ *V* ⟷ (*s* ↾ *V* = *s*)
  ⟨*proof*⟩

lemma *seq-filter-empty*: *s* ↾ {} = [] ⟨*proof*⟩

lemma *seq-filter-size*: #(*s* ↾ *V*) ≤ #*s*
  ⟨*proof*⟩

lemma *seq-filter-twice*: (*s* ↾ *V*) ↾ *W* = *s* ↾ (*V* ∩ *W*) ⟨*proof*⟩

25.20 Examples

lemma ([*1*,*2*,*3*] **;** (λ *x* · *x* + *1*)) *1* = *2*
  ⟨*proof*⟩

end

26 Pretty Notation for Z

theory *Z-Toolkit-Pretty*
  imports *Relation-Toolkit Number-Toolkit*
  abbrevs +> = ⇸ **and** +> = ⤀ **and** ++> = ⇻
    and >–> = ↣ **and** >–> = ⤖ **and** >+> = ⤔ **and** >++> = ⤕
    and <| = ◁ **and** <| = ◁ **and** <| = ⩤ **and** <| = ⦇
    and |> = ▷ **and** |> = ▷ **and** |> = ⩥ **and** |> = ⦈
    and |‘ = ↾ **and** |‘ = ↿ **and** |‘ = ↾
    and *O*+ = ⊕
    and ; ; = ⨾ **and** ; ; = ⨟
    and *PP* = ℙ **and** *FF* = 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** $\rightarrow$ *60*)

notation *rel-tfun* (**infixr** $\rightarrow$ *60*)
notation *rel-pfun* (**infixr** $\rightarrow\!\!\!\!\!+$ *60*)
notation *rel-ffun* (**infixr** $\twoheadrightarrow\!\!\!\!\!\!+$ *60*)

end

context
  includes *Z-RFun-Syntax*
begin

26.1 Examples

typ $\mathbb{P}\ \mathbb{N} \rightarrow \mathbb{N}$
typ $\mathbb{P}\ \mathbb{N} \rightarrow\!\!\!\!\!+ \mathbb{B}$
term $\{\}$
term $P ⨾ Q$
term $A \lhd B \unlhd (P :: \mathbb{P}(\mathbb{N}) \rightarrow\!\!\!\!\!+ \mathbb{B})$

term $\mathbb{P}\ \mathbb{N} \rightarrow \mathbb{N}$
term $\mathbb{N} \twoheadrightarrow\!\!\!\!\!\!+ \mathbb{N}$

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* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \rightarrow\!\!\!\!\!+ 'b$ **where**
$\textit{pfun-spec}\ P\ Q = (\lambda\ x \mid P\ x \wedge (\exists\ y.\ Q\ x\ y) \cdot \textit{SOME}\ y.\ Q\ x\ y)$

lemma *pfun-spec-app-eqI* [*intro*]: $[\![\ P\ x;\ \bigwedge y.\ Q\ x\ y \longleftrightarrow y = f\ x\ ]\!] \Longrightarrow (\textit{pfun-spec}\ P\ Q)(x)_p = f\ x$
  $\langle proof \rangle$

$\langle ML \rangle$

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.