Contents

1 Bool

```
(******************************************************************************)
(* Boolean                                                                    *)
(******************************************************************************)

(* rename module to clash with existing list modules of targets *)

declare {isabelle; hol; ocaml} rename module = lem_bool

(* The type bool is hard-coded, so are true and false *)

declare tex target_rep type $\MATHBB{B}$ = '$\mathbb{B}$'

(* ----------------------- *)
(* not                     *)
(* ----------------------- *)

val not : 𝔹 → 𝔹
let not b = match b with
  | true → false
  | false → true
end

declare hol target_rep function not x = '~' x
declare ocaml target_rep function not = 'not'
declare isabelle target_rep function not x = '\<not>' x
declare html target_rep function not = '&not;'
declare coq target_rep function not = 'negb'
declare tex target_rep function not b = '$\neg$' b

assert not1 : ¬ (¬ true)
assert not2 : ¬ false

(* ----------------------- *)
(* and                     *)
(* ----------------------- *)

val && [and] : 𝔹 → 𝔹 → 𝔹
let && b1 b2 = match (b1, b2) with
  | (true, true) → true
  | _ → false
end

declare hol target_rep function and = infix '/\'
declare ocaml target_rep function and = infix '&&'
declare isabelle target_rep function and = infix '\<and>'
declare coq target_rep function and = infix '&&'
declare html target_rep function and = infix '&and;'
declare tex target_rep function and = infix '$\wedge$'

assert and1 : (¬ (true ∧ false))
assert and2 : (¬ (false ∧ true))
assert and3 : (¬ (false ∧ false))
assert and4 : (true ∧ true)


(* ----------------------- *)
```

```
(* or                     *)
(* ---------------------- *)
```

val || [or] : $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$
let || b_1 b_2 = match $(b_1,\ b_2)$ with
  | (false, false) $\rightarrow$ false
  | _ $\rightarrow$ true
end

declare *hol* target_rep function or = infix '\/'
declare *ocaml* target_rep function or = infix '||'
declare *isabelle* target_rep function or = infix '\<or>'
declare *coq* target_rep function or = infix '||'
declare *html* target_rep function or = infix '∨'
declare *tex* target_rep function or = infix '$\vee$'

assert or_1 : (true $\vee$ false)
assert or_2 : (false $\vee$ true)
assert or_3 : (true $\vee$ true)
assert or_4 : ($\neg$ (false $\vee$ false))

```
(* ---------------------- *)
(* implication            *)
(* ---------------------- *)
```

val $--\!>$ [imp] : $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$
let $--\!>$ b_1 b_2 = match $(b_1,\ b_2)$ with
  | (true, false) $\rightarrow$ false
  | _ $\rightarrow$ true
end

declare *hol* target_rep function imp = infix '==>'
declare *isabelle* target_rep function imp = infix '\<longrightarrow>'
```
(* declare coq      target_rep function (-->) = `imp` *)
```
declare *html* target_rep function imp = infix '→'
declare *tex* target_rep function imp = infix '$\longrightarrow$'

let inline {*ocaml*; *coq*} *imp* x y = $((\neg\ x) \vee y)$

assert imp_1 : ($\neg$ (true $\longrightarrow$ false))
assert imp_2 : (false $\longrightarrow$ true)
assert imp_3 : (false $\longrightarrow$ false)
assert imp_4 : (true $\longrightarrow$ true)

```
(* ---------------------- *)
(* equivalence            *)
(* ---------------------- *)
```

val $<\!->$ [equiv] : $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$
let $<\!->$ b_1 b_2 = match $(b_1,\ b_2)$ with
  | (true, true) $\rightarrow$ true
  | (false, false) $\rightarrow$ true
  | _ $\rightarrow$ false
end

declare *hol* target_rep function equiv = infix '<=>'

declare *isabelle* target_rep function equiv = infix '\<longleftrightarrow>'
declare *coq* target_rep function equiv = 'eqb'
declare *ocaml* target_rep function equiv = infix '='
declare *html* target_rep function equiv = infix '↔'
declare *tex* target_rep function equiv = infix '$\longleftrightarrow$'

assert $equiv_1$: ($\neg$ (true $\longleftrightarrow$ false))
assert $equiv_2$: ($\neg$ (false $\longleftrightarrow$ true))
assert $equiv_3$: (false $\longleftrightarrow$ false)
assert $equiv_4$: (true $\longleftrightarrow$ true)

2 Basic_classes

```
(******************************************************************************)
(* Basic Type Classes                                                         *)
(******************************************************************************)
```

open import *Bool*

declare {*isabelle*; *ocaml*; *hol*} rename module = lem_basic_classes

```
(* ========================================================================== *)
(* Equality                                                                   *)
(* ========================================================================== *)

(* Lem`s default equality (=) is defined by the following type-class Eq.
   This typeclass should define equality on an abstract datatype 'a. It should
   always coincide with the default equality of Coq, HOL and Isabelle.
   For OCaml, it might be different, since abstract datatypes like sets
   might have fancy equalities. *)
```

class (*Eq* α)
  val = [isEqual] : $\alpha \rightarrow \alpha \rightarrow \mathbb{B}$
  val <> [isInequal] : $\alpha \rightarrow \alpha \rightarrow \mathbb{B}$
end

declare *coq* target_rep function isEqual = infix '='

```
(* declare coq target_rep function isEqual = infix `=`
declare coq target_rep function isInequal = infix `<>` *)
```

declare *tex* target_rep function isInequal = infix '$\neq$'

```
(* (=) should for all instances be an equivalence relation
   The isEquivalence predicate of relations could be used here.
   However, this would lead to a cyclic dependency. *)

(* TODO: add later, once lemmata can be assigned to classes
lemma eq_equiv: ((forall x. (x = x)) &&
                 (forall x y. (x = y) <-> (y = x)) &&
                 (forall x y z. ((x = y) && (y = z)) --> (x = z)))
*)

(* Structural equality *)

(* Sometimes, it is also handy to be able to use structural equality.
   This equality is mapped to the build-in equality of backends. This equality
   differs significantly for each backend. For example, OCaml can`t check equality
   of function types, whereas HOL can.  When using structural equality, one should
   know what one is doing. The only guarentee is that is behaves like
   the native backend equality.

   A lengthy name for structural equality is used to discourage its direct use.
   It also ensures that users realise it is unsafe (e.g. OCaml can`t check two functions
   for equality *)
```

val *unsafe_structural_equality* : $\forall\ \alpha.\ \alpha \rightarrow \alpha \rightarrow \mathbb{B}$

declare *hol* target_rep function unsafe_structural_equality = infix '='
declare *ocaml* target_rep function unsafe_structural_equality = infix '='
declare *isabelle* target_rep function unsafe_structural_equality = infix '='

```
declare coq target_rep function unsafe_structural_equality = 'classical_boolean_equivalence'

val unsafe_structural_inequality : ∀ α. α → α → 𝔹
let unsafe_structural_inequality x y = ¬ (unsafe_structural_equality x y)
declare isabelle target_rep function unsafe_structural_inequality = infix '\<noteq>'
declare hol target_rep function unsafe_structural_inequality = infix '<>'

(* The default for equality is the unsafe structural one. It can
   (and should) be overriden for concrete types later. *)
default_instance ∀ α. (Eq α)
  let = = unsafe_structural_equality
  let <> = unsafe_structural_inequality
end

(* for HOL and Isabelle, be even stronger and always(!) use
   standard equality *)
let inline {hol; isabelle} = = unsafe_structural_equality
let inline {hol; isabelle} <> = unsafe_structural_inequality


(* ========================================================================== *)
(* Orderings                                                                  *)
(* ========================================================================== *)

(* The type-class Ord represents total orders (also called linear orders) *)
type ORDERING = LT | EQ | GT

declare ocaml target_rep type ORDERING = 'int'
declare ocaml target_rep function LT = '(-1)'
declare ocaml target_rep function EQ = '0'
declare ocaml target_rep function GT = '1'

declare coq target_rep type ORDERING = 'ordering'
declare coq target_rep function LT = 'LT'
declare coq target_rep function EQ = 'EQ'
declare coq target_rep function GT = 'GT'

let orderingIsLess r = (match r with LT → true | _ → false end)
let orderingIsGreater r = (match r with GT → true | _ → false end)
let orderingIsEqual r = (match r with EQ → true | _ → false end)
let inline orderingIsLessEqual r = ¬ (orderingIsGreater r)
let inline orderingIsGreaterEqual r = ¬ (orderingIsLess r)

let ordering_cases r lt eq gt =
  if orderingIsLess r then lt else
  if orderingIsEqual r then eq else gt

declare ocaml target_rep function orderingIsLess = 'Lem.orderingIsLess'
declare ocaml target_rep function orderingIsGreater = 'Lem.orderingIsGreater'
declare ocaml target_rep function orderingIsEqual = 'Lem.orderingIsEqual'

declare ocaml target_rep function ordering_cases = 'Lem.ordering_cases'

declare {ocaml} pattern_match exhaustive ORDERING = [ LT; EQ ; GT ] ordering_cases

assert ordering_cases_0 : (ordering_cases LT true false false)
assert ordering_cases_1 : (ordering_cases EQ false true false)
assert ordering_cases_2 : (ordering_cases GT false false true)
```

assert *ordering_match*$_1$: (match LT with GT $\rightarrow$ false $\wedge$ false | _ $\rightarrow$ true end)
assert *ordering_match*$_2$: (match EQ with GT $\rightarrow$ false | _ $\rightarrow$ true end)
assert *ordering_match*$_3$: (match GT with GT $\rightarrow$ true $\wedge$ true | _ $\rightarrow$ false end)
assert *ordering_match*$_4$: ((fun r $\rightarrow$ (match r with GT $\rightarrow$ false | _ $\rightarrow$ true end)) LT)
assert *ordering_match*$_5$: ((fun r $\rightarrow$ (match r with GT $\rightarrow$ false | _ $\rightarrow$ true end)) EQ)
assert *ordering_match*$_6$: ((fun r $\rightarrow$ (match r with GT $\rightarrow$ true $\wedge$ true | _ $\rightarrow$ false end)) GT)

val *orderingEqual* : ORDERING $\rightarrow$ ORDERING $\rightarrow$ $\mathbb{B}$
let inline $\sim$\{*ocaml*; *coq*\} *orderingEqual* = unsafe_structural_equality
declare *coq* target_rep function orderingEqual = `'ordering_equal'`
declare *ocaml* target_rep function orderingEqual = `'Lem.orderingEqual'`

instance (*Eq* ORDERING)
  let = = orderingEqual
  let $<>$ x y = $\neg$ (orderingEqual x y)
end

class (*Ord* α)
  val *compare* : $\alpha \rightarrow \alpha \rightarrow$ ORDERING
  val $<$ [isLess] : $\alpha \rightarrow \alpha \rightarrow \mathbb{B}$
  val $<=$ [isLessEqual] : $\alpha \rightarrow \alpha \rightarrow \mathbb{B}$
  val $>$ [isGreater] : $\alpha \rightarrow \alpha \rightarrow \mathbb{B}$
  val $>=$ [isGreaterEqual] : $\alpha \rightarrow \alpha \rightarrow \mathbb{B}$
end

declare *coq* target_rep function isLess = `'isLess'`
declare *coq* target_rep function isLessEqual = `'isLessEqual'`
declare *coq* target_rep function isGreater = `'isGreater'`
declare *coq* target_rep function isGreaterEqual = `'isGreaterEqual'`
declare *tex* target_rep function isLess = infix `'$<$'`
declare *tex* target_rep function isLessEqual = infix `'$\le$'`
declare *tex* target_rep function isGreater = infix `'$>$'`
declare *tex* target_rep function isGreaterEqual = infix `'$\ge$'`

```
(* Ocaml provides default, polymorphic compare functions. Let's use them
   as the default. However, because used perhaps in a typeclass they must be
   defined for all targets. So, explicitly declare them as undefined for
   all other targets. If explictly declare undefined, the type-checker won't complain and
   an error will only be raised when trying to actually output the function for a certain
   target. *)
```
val *defaultCompare* : $\forall\, \alpha.\ \alpha \rightarrow \alpha \rightarrow$ ORDERING
val *defaultLess* : $\forall\, \alpha.\ \alpha \rightarrow \alpha \rightarrow \mathbb{B}$
val *defaultLessEq* : $\forall\, \alpha.\ \alpha \rightarrow \alpha \rightarrow \mathbb{B}$
val *defaultGreater* : $\forall\, \alpha.\ \alpha \rightarrow \alpha \rightarrow \mathbb{B}$
val *defaultGreaterEq* : $\forall\, \alpha.\ \alpha \rightarrow \alpha \rightarrow \mathbb{B}$

declare *ocaml* target_rep function defaultCompare = `'compare'`
declare *hol* target_rep function defaultCompare =
declare *isabelle* target_rep function defaultCompare =
declare *coq* target_rep function defaultCompare x y = EQ

declare *ocaml* target_rep function defaultLess = infix `'<'`
declare *hol* target_rep function defaultLess =
declare *isabelle* target_rep function defaultLess =
declare *coq* target_rep function defaultLess =

```
declare ocaml target_rep function defaultLessEq = infix '<='
declare hol target_rep function defaultLessEq =
declare isabelle target_rep function defaultLessEq =
declare coq target_rep function defaultLessEq =

declare ocaml target_rep function defaultGreater = infix '>'
declare hol target_rep function defaultGreater =
declare isabelle target_rep function defaultGreater =
declare coq target_rep function defaultGreater =

declare ocaml target_rep function defaultGreaterEq = infix '>='
declare hol target_rep function defaultGreaterEq =
declare isabelle target_rep function defaultGreaterEq =
declare coq target_rep function defaultGreaterEq =
;;

let genericCompare (less : α → α → 𝔹) (equal : α → α → 𝔹) (x : α) (y : α) =
  if less x y then
    LT
  else if equal x y then
    EQ
  else
    GT


(*
(* compare should really be a total order *)
lemma ord_OK_1: (
  (forall x y. (compare x y = EQ) <-> (compare y x = EQ)) &&
  (forall x y. (compare x y = LT) <-> (compare y x = GT)))

lemma ord_OK_2: (
  (forall x y z. (x <= y) && (y <= z) --> (x <= z)) &&
  (forall x y. (x <= y) || (y <= x))
)
*)

(* let's derive a compare function from the Ord type-class *)
val ordCompare : ∀ α. Eq α, Ord α ⇒ α → α → ORDERING
let ordCompare x y =
  if (x < y) then LT else
  if (x = y) then EQ else GT

class ( OrdMaxMin α )
  val max : α → α → α
  val min : α → α → α
end

val defaultMin : ∀ α. Ord α ⇒ α → α → α
let defaultMin x y = if (x ≤ y) then x else y
declare ocaml target_rep function defaultMin = 'min'

val defaultMax : ∀ α. Ord α ⇒ α → α → α
let defaultMax x y = if (y ≤ x) then x else y
declare ocaml target_rep function defaultMax = 'max'


default_instance ∀ α. Ord α ⇒ ( OrdMaxMin α)
```

```
  let max = defaultMax
  let min = defaultMin
end


(* ========================================================================== *)
(* SetTypes                                                                   *)
(* ========================================================================== *)

(* Set implementations use often an order on the elements. This allows the OCaml implementation
   to use trees for implementing them. At least, one needs to be able to check equality on
sets.
   One could use the Ord type-class for sets. However, defining a special typeclass is cleaner
   and allows more flexibility. One can make e.g. sure, that this type-class is ignored for
   backends like HOL or Isabelle, which don't need it. Moreover, one is not forced to also
instantiate
   the functions "<", "<=" ... *)

class ( SetType α )
  val {ocaml; coq} setElemCompare : α → α → ORDERING
end

default_instance ∀ α. ( SetType α )
  let setElemCompare = defaultCompare
end

(* ========================================================================== *)
(* Instantiations                                                             *)
(* ========================================================================== *)

instance (Eq 𝔹)
  let = = (⟷)
  let <> x y = ¬ ((⟷) x y)
end

let boolCompare b1 b2 = match (b1, b2) with
  | (true, true) → EQ
  | (true, false) → GT
  | (false, true) → LT
  | (false, false) → EQ
end

instance (SetType 𝔹)
  let setElemCompare = boolCompare
end

val pairEqual : ∀ α β. Eq α, Eq β ⇒ (α ∗ β) → (α ∗ β) → 𝔹
let pairEqual (a1, b1) (a2, b2) = (a1 = a2) ∧ (b1 = b2)

val pairEqualBy : ∀ α β. (α → α → 𝔹) → (β → β → 𝔹) → (α ∗ β) → (α ∗ β) → 𝔹
declare ocaml target_rep function pairEqualBy = 'Lem.pair_equal'
declare coq target_rep function pairEqualBy = 'tuple_equal_by'

let inline {hol; isabelle} pairEqual = unsafe_structural_equality
let inline {ocaml; coq} pairEqual = pairEqualBy (=) (=)

instance ∀ α β. Eq α, Eq β ⇒ (Eq (α ∗ β))
  let = = pairEqual
```

let $<>\ x\ y\ =\ \neg\,(\mathrm{pairEqual}\ x\ y)$
end

val $\mathit{pairCompare}\ :\ \forall\ \alpha\ \beta.\ (\alpha\ \to\ \alpha\ \to\ \textsc{ordering})\ \to\ (\beta\ \to\ \beta\ \to\ \textsc{ordering})\ \to\ (\alpha\ *\ \beta)\ \to\ (\alpha\ *\ \beta)\ \to\ \textsc{ordering}$
let $\mathit{pairCompare}\ \mathit{cmpa}\ \mathit{cmpb}\ (a_1,\ b_1)\ (a_2,\ b_2)\ =$
  match $\mathit{cmpa}\ a_1\ a_2$ with
    $|\ \mathrm{LT}\ \to\ \mathrm{LT}$
    $|\ \mathrm{GT}\ \to\ \mathrm{GT}$
    $|\ \mathrm{EQ}\ \to\ \mathit{cmpb}\ b_1\ b_2$
  end

let $\mathit{pairLess}\ (x_1,\ x_2)\ (y_1,\ y_2)\ =\ (x_1\ <\ y_1) \lor ((x_1\ \leq\ y_1) \land (x_2\ <\ y_2))$
let $\mathit{pairLessEq}\ (x_1,\ x_2)\ (y_1,\ y_2)\ =\ (x_1\ <\ y_1) \lor ((x_1\ \leq\ y_1) \land (x_2\ <\ y_2))$

let $\mathit{pairGreater}\ x_{12}\ y_{12}\ =\ \mathrm{pairLess}\ y_{12}\ x_{12}$
let $\mathit{pairGreaterEq}\ x_{12}\ y_{12}\ =\ \mathrm{pairLessEq}\ y_{12}\ x_{12}$

instance $\forall\ \alpha\ \beta.\ \mathit{Ord}\ \alpha,\ \mathit{Ord}\ \beta\ \Rightarrow\ (\mathit{Ord}\ (\alpha\ *\ \beta))$
  let $\mathit{compare}\ =\ \mathrm{pairCompare\ compare\ compare}$
  let $<\ =\ \mathrm{pairLess}$
  let $<=\ =\ \mathrm{pairLessEq}$
  let $>\ =\ \mathrm{pairGreater}$
  let $>=\ =\ \mathrm{pairGreaterEq}$
end

val $\mathit{test}\ :\ \forall\ \alpha\ \beta.\ \mathit{SetType}\ \alpha,\ \mathit{SetType}\ \beta\ \Rightarrow\ (\alpha\ *\ \beta)\ \to\ (\alpha\ *\ \beta)\ \to\ \textsc{ordering}$
let $\{\mathit{ocaml}\}\ \mathit{test}\ =\ \mathrm{pairCompare\ setElemCompare\ setElemCompare}$

instance $\forall\ \alpha\ \beta.\ \mathit{SetType}\ \alpha,\ \mathit{SetType}\ \beta\ \Rightarrow\ (\mathit{SetType}\ (\alpha\ *\ \beta))$
  let $\mathit{setElemCompare}\ =\ \mathrm{pairCompare\ setElemCompare\ setElemCompare}$
end

3 Function

```
(**************************************************************************)
(* A library for common operations on functions                          *)
(**************************************************************************)
```

open import *Bool Basic_classes*

declare {*isabelle*; *hol*; *ocaml*} rename module = lem_function

```
(* ----------------------- *)
(* identity function       *)
(* ----------------------- *)
```

val *id* : $\forall\, \alpha.\ \alpha \ \to\ \alpha$
let *id x* = *x*

let inline {*coq*} *id x* = *x*
declare *isabelle* target_rep function id = `'id'`
declare *hol* target_rep function id = `'I'`

```
(* ----------------------- *)
(* constant function       *)
(* ----------------------- *)
```

val *const* : $\forall\, \alpha\ \beta.\ \alpha \ \to\ \beta \ \to\ \alpha$
let inline *const x y* = *x*

declare *coq* target_rep function const = `'const'`
declare *hol* target_rep function const = `'K'`

```
(* ----------------------- *)
(* function composition    *)
(* ----------------------- *)
```

val *comb* : $\forall\, \alpha\ \beta\ \gamma.\ (\beta \ \to\ \gamma) \ \to\ (\alpha \ \to\ \beta) \ \to\ (\alpha \ \to\ \gamma)$
let *comb f g* = (fun *x* → *f* (*g x*))

declare *coq* target_rep function comb = `'compose'`
declare *isabelle* target_rep function comb = infix `'o'`
declare *hol* target_rep function comb = infix `'o'`

```
(* ----------------------- *)
(* function application    *)
(* ----------------------- *)
```

val $ [apply] : $\forall\, \alpha\ \beta.\ (\alpha \ \to\ \beta) \ \to\ (\alpha \ \to\ \beta)$
let $ *f* = (fun *x* → *f x*)

declare *coq* target_rep function apply = `'apply'`
let inline {*isabelle*; *ocaml*; *hol*} *apply f x* = *f x*

```
(* ----------------------- *)
(* flipping argument order *)
(* ----------------------- *)
```

val *flip* : $\forall\ \alpha\ \beta\ \gamma.\ (\alpha\ \rightarrow\ \beta\ \rightarrow\ \gamma)\ \rightarrow\ (\beta\ \rightarrow\ \alpha\ \rightarrow\ \gamma)$
let *flip* f = (fun $x\ y\ \rightarrow\ f\ y\ x$)

declare *coq* target_rep function flip = `'flip'`
let inline $\{isabelle\}$ *flip* $f\ x\ y\ =\ f\ y\ x$
declare *hol* target_rep function flip = `'combin$C'`

4 Maybe

```
(******************************************************************************)
(* A library for option                                                       *)
(*                                                                            *)
(* It mainly follows the Haskell Maybe-library                                *)
(******************************************************************************)

declare {hol; isabelle; ocaml} rename module = lem_maybe

open import Bool Basic_classes Function

(* ========================================================================== *)
(* Basic stuff                                                                *)
(* ========================================================================== *)

type MAYBE α =
  | NOTHING
  | JUST of α

declare hol target_rep type MAYBE α = 'option' α
declare isabelle target_rep type MAYBE α = 'option' α
declare coq target_rep type MAYBE α = 'option' α
declare ocaml target_rep type MAYBE α = 'option' α

declare hol target_rep function Just = 'SOME'
declare ocaml target_rep function Just = 'Some'
declare isabelle target_rep function Just = 'Some'
declare coq target_rep function Just = 'Some'

declare hol target_rep function Nothing = 'NONE'
declare ocaml target_rep function Nothing = 'None'
declare isabelle target_rep function Nothing = 'None'
declare coq target_rep function Nothing = 'None'


val maybeEqual : ∀ α. Eq α ⇒ MAYBE α → MAYBE α → 𝔹
val maybeEqualBy : ∀ α. (α → α → 𝔹) → MAYBE α → MAYBE α → 𝔹

let maybeEqualBy eq x y = match (x, y) with
  | (Nothing, Nothing) → true
  | (Nothing, Just _) → false
  | (Just _, Nothing) → false
  | (Just x', Just y') → (eq x' y')
end
let inline maybeEqual = maybeEqualBy (=)

declare ocaml target_rep function maybeEqualBy = 'Lem.option_equal'
let inline {hol; isabelle} maybeEqual = unsafe_structural_equality

instance ∀ α. Eq α ⇒ (Eq (MAYBE α))
  let = = maybeEqual
  let <> x y = ¬ (maybeEqual x y)
end

assert maybe_eq1 : ((Nothing : MAYBE 𝔹) = Nothing)
assert maybe_eq2 : ((Just true) ≠ Nothing)
assert maybe_eq3 : ((Just false) ≠ (Just true))
```

```
assert maybe_eq_4 : ((Just false) = (Just false))


let maybeCompare cmp x y = match (x, y) with
  | (Nothing, Nothing) → EQ
  | (Nothing, Just _) → LT
  | (Just _, Nothing) → GT
  | (Just x′, Just y′) → cmp x′ y′
end

instance ∀ α. SetType α ⇒ (SetType (MAYBE α))
  let setElemCompare = maybeCompare setElemCompare
end



(* ----------------------- *)
(* maybe                   *)
(* ----------------------- *)

val maybe : ∀ α β. β → (α → β) → MAYBE α → β
let maybe d f mb = match mb with
  | Just a → f a
  | Nothing → d
end

declare ocaml target_rep function maybe = 'Lem.option_case'
declare isabelle target_rep function maybe = 'option_case'
declare hol target_rep function maybe d f mb = 'option_CASE' mb d f

assert maybe_1 : (maybe true (fun b → ¬ b) Nothing = true)
assert maybe_2 : (maybe false (fun b → ¬ b) Nothing = false)
assert maybe_3 : (maybe true (fun b → ¬ b) (Just true) = false)
assert maybe_4 : (maybe true (fun b → ¬ b) (Just false) = true)

(* ----------------------- *)
(* isJust / isNothing      *)
(* ----------------------- *)

val isJust : ∀ α. MAYBE α → 𝔹
let isJust mb = match mb with
  | Just _ → true
  | Nothing → false
end

declare hol target_rep function isJust = 'IS_SOME'
declare ocaml target_rep function isJust = 'Lem.is_some'
declare isabelle target_rep function isJust x = '$\neg$' (unsafe_structural_equality x Nothing)

assert isJust_1 : (isJust (Just true))
assert isJust_2 : (¬ (isJust (Nothing : MAYBE 𝔹)))

val isNothing : ∀ α. MAYBE α → 𝔹
let isNothing mb = match mb with
  | Just _ → false
  | Nothing → true
end

declare hol target_rep function isNothing = 'IS_NONE'
```

```
declare ocaml target_rep function isNothing  =  'Lem.is_none'
declare isabelle target_rep function isNothing x  =  (unsafe_structural_equality x Nothing)

assert isNothing_1 :  (¬ (isNothing (Just true)))
assert isNothing_2 :  (isNothing (Nothing  :  MAYBE 𝔹))

lemma isJustNothing :  (
  (∀ x. isNothing x = ¬ (isJust x)) ∧
  (∀ v. isJust (Just v)) ∧
  (isNothing Nothing))

(* ----------------------- *)
(* fromMaybe               *)
(* ----------------------- *)

val fromMaybe  :  ∀ α. α  →  MAYBE α  →  α
let fromMaybe  d  mb  =  match mb with
   | Just v  →  v
   | Nothing  →  d
end

declare ocaml target_rep function fromMaybe  =  'Lem.option_default'
let inline {isabelle; hol} fromMaybe  d  =  maybe d id

lemma fromMaybe :  (
  (∀ d v. fromMaybe d (Just v) = v) ∧
  (∀ d. fromMaybe d Nothing = d))

assert fromMaybe_1 :  (fromMaybe true Nothing = true)
assert fromMaybe_2 :  (fromMaybe false Nothing = false)
assert fromMaybe_3 :  (fromMaybe true (Just true) = true)
assert fromMaybe_4 :  (fromMaybe true (Just false) = false)

(* ----------------------- *)
(* map                     *)
(* ----------------------- *)

val map  :  ∀ α β. (α  →  β)  →  MAYBE α  →  MAYBE β
let map f  =  maybe Nothing (fun v  →  Just (f  v))

declare hol target_rep function map  =  'OPTION_MAP'
declare ocaml target_rep function map  =  'Lem.option_map'
declare isabelle target_rep function map  =  'Option.map'
declare coq target_rep function map  =  'option_map'

lemma maybe_map :  (
  (∀ f. map f  Nothing = Nothing) ∧
  (∀ f  v. map f  (Just v) = Just (f  v)))

assert map_1 :  (map (fun b → ¬ b) Nothing = Nothing)
assert map_2 :  (map (fun b → ¬ b) (Just true) = Just false)
assert map_3 :  (map (fun b → ¬ b) (Just false) = Just true)


(* ----------------------- *)
(* bind                    *)
(* ----------------------- *)
```

```
val bind : ∀ α β. (α → MAYBE β) → MAYBE α → MAYBE β
let bind f = maybe Nothing f

declare isabelle target_rep function bind f mb = 'Option.bind' mb f
declare ocaml target_rep function bind = 'Lem.option_bind'
declare hol target_rep function bind f mb = 'OPTION_BIND' mb f

lemma maybe_bind : (
  (∀ f. bind f Nothing = Nothing) ∧
  (∀ f v. bind f (Just v) = (f v)))

assert bind_1 : (bind (fun b → Just (¬ b)) Nothing = Nothing)
assert bind_2 : (bind (fun b → Just (¬ b)) (Just true) = Just false)
assert bind_3 : (bind (fun b → Just (¬ b)) (Just false) = Just true)
assert bind_4 : (bind (fun b → (Nothing : MAYBE 𝔹)) (Just false) = Nothing)
```

5 Num

```
(******************************************************************************)
(* A library for numbers                                                      *)
(*                                                                            *)
(* It mainly follows the Haskell Maybe-library                                *)
(******************************************************************************)

(* rename module to clash with existing list modules of targets
   problem: renaming from inside the module itself! *)
```

declare {*isabelle*; *ocaml*; *hol*} rename module = lem_num

open import *Bool Basic_classes*
open import {*hol*} *integerTheory intReduce*
open import {*coq*} *Coq.ZArith.BinInt Coq.ZArith.Zpower Coq.ZArith.Zdiv Coq.ZArith.Zmax*

```
(* ========================================================================== *)
(* Numerals                                                                   *)
(* ========================================================================== *)

(* Numerals like 0, 1, 2, 42, 4543 are build-in. That's the only use
   of numerals. The following type-class is used to convert numerals into
   verious number types. The type of numerals differs form backend to backend.
   Essentially they are just printed as "0", "1", ... and the backend decides
   then. For Ocaml, they are integers. For HOL of type "num". Isabelle thinks
   they are polymorphic. ...
*)
```

declare *hol* target_rep type NUMERAL = `'num'`
declare *coq* target_rep type NUMERAL = `'nat'`
declare *ocaml* target_rep type NUMERAL = `'int'`

class inline (*Numeral* α)
  val *fromNumeral* : NUMERAL $\rightarrow$ α
end

```
(* ========================================================================== *)
(* Syntactic type-classes for common operations                               *)
(* ========================================================================== *)

(* Typeclasses can be used as a mean to overload constants like "+", "-", etc *)
```

class (*NumNegate* α)
  val $\sim$ [numNegate] : α $\rightarrow$ α
end
declare *tex* target_rep function numNegate = `'$-$'`

class (*NumAdd* α)
  val $+$ [numAdd] : α $\rightarrow$ α $\rightarrow$ α
end
declare *tex* target_rep function numAdd = infix `'$+$'`

class (*NumMinus* α)
  val $-$ [numMinus] : α $\rightarrow$ α $\rightarrow$ α
end
declare *tex* target_rep function numMinus = infix `'$-$'`

class (*NumMult* α)
 val $*$ [numMult] : $\alpha \rightarrow \alpha \rightarrow \alpha$
end
declare *tex* target_rep function numMult = infix `'$*$'`

class (*NumPow* α)
 val $**$ [numPow] : $\alpha \rightarrow$ NAT $\rightarrow \alpha$
end
declare *tex* target_rep function numPow n m = special "{%e}↑{%e}" n m

class (*NumDivision* α)
 val / [numDivision] : $\alpha \rightarrow \alpha \rightarrow \alpha$
end

class (*NumIntegerDivision* α)
 val *div* [numIntegerDivision] : $\alpha \rightarrow \alpha \rightarrow \alpha$
end

class (*NumRemainder* α)
 val *mod* [numRemainder] : $\alpha \rightarrow \alpha \rightarrow \alpha$
end

class (*NumSucc* α)
 val *succ* : $\alpha \rightarrow \alpha$
end

class (*NumPred* α)
 val *pred* : $\alpha \rightarrow \alpha$
end

```
(* ========================================================================== *)
(* Basic number types                                                         *)
(* ========================================================================== *)

(* ----------------------- *)
(* nat                     *)
(* ----------------------- *)

(* bounded size natural numbers, i.e. positive integers *)

(* "nat" is the old type "num". It represents natural numbers.
   These numbers might be bounded, however no checks of the boundedness are
   provided. The theorem prover backends map nat to unbounded size
   natural numbers. However, OCaml uses the type "int", which is bounded.
   Using "int" allows using many functions like "List.length" without wrappers.
   This leeds to nice readable code, but a slightly fuzzy concept what
   "nat" represents. If you want to use unbounded natural numbers, use "natural"
   instead. *)
```

declare *hol* target_rep type NAT = `'num'`
declare *isabelle* target_rep type NAT = `'nat'`
declare *coq* target_rep type NAT = `'nat'`
declare *ocaml* target_rep type NAT = `'int'`

```
(* ----------------------- *)
(* natural                 *)
```

```
(* ----------------------- *)

(* unbounded size natural numbers *)
type NATURAL
declare hol target_rep type $\MATHBB{N}$ = 'num'
declare isabelle target_rep type $\MATHBB{N}$ = 'nat'
declare coq target_rep type $\MATHBB{N}$ = 'nat'
declare ocaml target_rep type $\MATHBB{N}$ = 'Big_int.big_int'
declare tex target_rep type $\MATHBB{N}$ = '$\mathbb{N}$'


(* ----------------------- *)
(* int                     *)
(* ----------------------- *)

(* bounded size integers with uncertain length *)

type INT
declare ocaml target_rep type INT = 'int'
declare isabelle target_rep type INT = 'int'
declare hol target_rep type INT = 'int'
declare coq target_rep type INT = 'Z'


(* ----------------------- *)
(* integer                 *)
(* ----------------------- *)

(* unbounded size integers *)

type INTEGER
declare ocaml target_rep type $\MATHBB{Z}$ = 'Big_int.big_int'
declare isabelle target_rep type $\MATHBB{Z}$ = 'int'
declare hol target_rep type $\MATHBB{Z}$ = 'int'
declare coq target_rep type $\MATHBB{Z}$ = 'Z'
declare tex target_rep type $\MATHBB{Z}$ = '$\mathbb{Z}$'

(* ----------------------- *)
(* bint                    *)
(* ----------------------- *)

(* 32 bit integers *)
type INT32
declare ocaml target_rep type INT32 = 'int'32
declare coq target_rep type INT32 = 'Z' (* ???: better type for this in Coq? *)
declare isabelle target_rep type INT32 = 'int' (* ???: better type for this in Isa? *)
declare hol target_rep type INT32 = 'int' (* ???: better type for this in HOL? *)

(* 64 bit integers *)
type INT64
declare ocaml target_rep type INT64 = 'int'64
declare coq target_rep type INT64 = 'Z' (* ???: better type for this in Coq? *)
declare isabelle target_rep type INT64 = 'int' (* ???: better type for this in Isa? *)
declare hol target_rep type INT64 = 'int' (* ???: better type for this in HOL? *)


(* ----------------------- *)
(* rational                *)
```

```
(* ----------------------- *)

(* unbounded size and precision rational numbers *)

type RATIONAL
declare ocaml target_rep type RATIONAL = 'Num.num'
declare coq target_rep type RATIONAL = 'Q' (* ???: better type for this in Coq? *)
declare isabelle target_rep type RATIONAL = 'rat' (* ???: better type for this in Isa? *)
declare hol target_rep type RATIONAL = 'XXX' (* ???: better type for this in HOL? *)


(* ----------------------- *)
(* double                  *)
(* ----------------------- *)

(* double precision floating point (64 bits) *)

type FLOAT64
declare ocaml target_rep type FLOAT64 = 'double'
declare coq target_rep type FLOAT64 = 'Q' (* ???: better type for this in Coq? *)
declare isabelle target_rep type FLOAT64 = '???' (* ???: better type for this in Isa? *)
declare hol target_rep type FLOAT64 = 'XXX' (* ???: better type for this in HOL? *)

type FLOAT32
declare ocaml target_rep type FLOAT32 = 'float'
declare coq target_rep type FLOAT32 = 'Q' (* ???: better type for this in Coq? *)
declare isabelle target_rep type FLOAT32 = '???' (* ???: better type for this in Isa? *)
declare hol target_rep type FLOAT32 = 'XXX' (* ???: better type for this in HOL? *)


(* ========================================================================== *)
(* Binding the standard operations for the number types                      *)
(* ========================================================================== *)


(* ----------------------- *)
(* nat                     *)
(* ----------------------- *)

val natFromNumeral : NUMERAL → NAT
declare hol target_rep function natFromNumeral = '' (* remove natFromNumeral, as it is the identify
function *)
declare ocaml target_rep function natFromNumeral = ''
declare isabelle target_rep function natFromNumeral n = (''n : NAT)
declare coq target_rep function natFromNumeral = 'id'

instance (Numeral NAT)
  let fromNumeral n = natFromNumeral n
end

val natEq : NAT → NAT → 𝔹
let inline natEq = unsafe_structural_equality
instance (Eq NAT)
  let = = natEq
  let <> n1 n2 = ¬ (natEq n1 n2)
end

val natLess : NAT → NAT → 𝔹
```

```
val natLessEqual : NAT → NAT → 𝔹
val natGreater : NAT → NAT → 𝔹
val natGreaterEqual : NAT → NAT → 𝔹

declare hol target_rep function natLess = infix '<'
declare ocaml target_rep function natLess = infix '<'
declare isabelle target_rep function natLess = infix '<'
declare coq target_rep function natLess = 'nat_ltb'

declare hol target_rep function natLessEqual = infix '<='
declare ocaml target_rep function natLessEqual = infix '<='
declare isabelle target_rep function natLessEqual = infix '\<le>'
declare coq target_rep function natLessEqual = 'nat_lteb'

declare hol target_rep function natGreater = infix '>'
declare ocaml target_rep function natGreater = infix '>'
declare isabelle target_rep function natGreater = infix '>'
declare coq target_rep function natGreater = 'nat_gtb'

declare hol target_rep function natGreaterEqual = infix '>='
declare ocaml target_rep function natGreaterEqual = infix '>='
declare isabelle target_rep function natGreaterEqual = infix '\<ge>'
declare coq target_rep function natGreaterEqual = 'nat_gteb'

val natCompare : NAT → NAT → ORDERING
let inline natCompare = defaultCompare
let inline {coq; hol; isabelle} natCompare = genericCompare natLess natEq

instance (Ord NAT)
  let compare = natCompare
  let < = natLess
  let <= = natLessEqual
  let > = natGreater
  let >= = natGreaterEqual
end

instance (SetType NAT)
  let setElemCompare = natCompare
end

val natAdd : NAT → NAT → NAT
declare hol target_rep function natAdd = infix '+'
declare ocaml target_rep function natAdd = infix '+'
declare isabelle target_rep function natAdd = infix '+'
declare coq target_rep function natAdd = 'Coq.Init.Peano.plus'

instance (NumAdd NAT)
  let + = natAdd
end

val natMinus : NAT → NAT → NAT
declare hol target_rep function natMinus = infix '-'
declare ocaml target_rep function natMinus = 'Nat_num.nat_monus'
declare isabelle target_rep function natMinus = infix '-'
declare coq target_rep function natMinus = 'Coq.Init.Peano.minus'

instance (NumMinus NAT)
  let − = natMinus
```

```
end

val natSucc : NAT → NAT
let natSucc n = n + 1
declare hol target_rep function natSucc = 'SUC'
declare isabelle target_rep function natSucc = 'Suc'
declare ocaml target_rep function natSucc = 'succ'
declare coq target_rep function natSucc = 'S'
instance (NumSucc NAT)
  let succ = natSucc
end

val natPred : NAT → NAT
let inline natPred n = n − 1
declare hol target_rep function natPred = 'PRE'
declare ocaml target_rep function natPred = 'Nat_num.nat_pred'
declare coq target_rep function natPred = 'Coq.Init.Peano.pred'
instance (NumPred NAT)
  let pred = natPred
end

val natMult : NAT → NAT → NAT
declare hol target_rep function natMult = infix '*'
declare ocaml target_rep function natMult = infix '*'
declare isabelle target_rep function natMult = infix '*'
declare coq target_rep function natMult = 'Coq.Init.Peano.mult'

instance (NumMult NAT)
  let * = natMult
end


val natPow : NAT → NAT → NAT
declare hol target_rep function natPow = infix '**'
declare ocaml target_rep function natPow = 'Nat_num.nat_pow'
declare isabelle target_rep function natPow = infix '↑'
declare coq target_rep function natPow = 'nat_power'

instance ( NumPow NAT )
  let ** = natPow
end

val natDiv : NAT → NAT → NAT
declare hol target_rep function natDiv = infix 'DIV'
declare ocaml target_rep function natDiv = infix '/'
declare isabelle target_rep function natDiv = infix 'div'
declare coq target_rep function natDiv = 'nat_div'

instance ( NumIntegerDivision NAT )
  let div = natDiv
end

instance ( NumDivision NAT )
  let / = natDiv
end

val natMod : NAT → NAT → NAT
declare hol target_rep function natMod = infix 'MOD'
```

```
declare ocaml target_rep function natMod = infix 'mod'
declare isabelle target_rep function natMod = infix 'mod'
declare coq target_rep function natMod = 'nat_mod'

instance ( NumRemainder NAT )
  let mod = natMod
end

val natMin : NAT → NAT → NAT
let inline natMin = defaultMin
declare ocaml target_rep function natMin = 'min'
declare isabelle target_rep function natMin = 'min'
declare hol target_rep function natMin = 'MIN'
declare coq target_rep function natMin = 'nat_min'

val natMax : NAT → NAT → NAT
let inline natMax = defaultMax
declare isabelle target_rep function natMax = 'max'
declare ocaml target_rep function natMax = 'max'
declare hol target_rep function natMax = 'MAX'
declare coq target_rep function natMax = 'nat_max'

instance ( OrdMaxMin NAT )
  let max = natMax
  let min = natMin
end


(* ----------------------- *)
(* natural                 *)
(* ----------------------- *)

val naturalFromNumeral : NUMERAL → ℕ
declare hol target_rep function naturalFromNumeral = '' (* remove naturalFromNumeral, as it is
the identify function *)
declare ocaml target_rep function naturalFromNumeral = 'Big_int.big_int_of_int'
declare isabelle target_rep function naturalFromNumeral n = (''n : ℕ)
declare coq target_rep function naturalFromNumeral = 'id'

instance (Numeral ℕ)
  let fromNumeral n = naturalFromNumeral n
end

val naturalEq : ℕ → ℕ → 𝔹
let inline naturalEq = unsafe_structural_equality
declare ocaml target_rep function naturalEq = 'Big_int.eq_big_int'
instance (Eq ℕ)
  let = = naturalEq
  let <> n1 n2 = ¬ (naturalEq n1 n2)
end

val naturalLess : ℕ → ℕ → 𝔹
val naturalLessEqual : ℕ → ℕ → 𝔹
val naturalGreater : ℕ → ℕ → 𝔹
val naturalGreaterEqual : ℕ → ℕ → 𝔹

declare hol target_rep function naturalLess = infix '<'
declare ocaml target_rep function naturalLess = 'Big_int.lt_big_int'
```

```
declare isabelle target_rep function naturalLess = infix '<'
declare coq target_rep function naturalLess = 'nat_ltb'

declare hol target_rep function naturalLessEqual = infix '<='
declare ocaml target_rep function naturalLessEqual = 'Big_int.le_big_int'
declare isabelle target_rep function naturalLessEqual = infix '\<le>'
declare coq target_rep function naturalLessEqual = 'nat_lteb'

declare hol target_rep function naturalGreater = infix '>'
declare ocaml target_rep function naturalGreater = 'Big_int.gt_big_int'
declare isabelle target_rep function naturalGreater = infix '>'
declare coq target_rep function naturalGreater = 'nat_gtb'

declare hol target_rep function naturalGreaterEqual = infix '>='
declare ocaml target_rep function naturalGreaterEqual = 'Big_int.ge_big_int'
declare isabelle target_rep function naturalGreaterEqual = infix '\<ge>'
declare coq target_rep function naturalGreaterEqual = 'nat_gteb'

val naturalCompare : ℕ → ℕ → ORDERING
let inline naturalCompare = defaultCompare
let inline {coq; isabelle; hol} naturalCompare = genericCompare naturalLess naturalEq
declare ocaml target_rep function naturalCompare = 'Big_int.compare_big_int'

instance (Ord ℕ)
  let compare = naturalCompare
  let < = naturalLess
  let <= = naturalLessEqual
  let > = naturalGreater
  let >= = naturalGreaterEqual
end

instance (SetType ℕ)
  let setElemCompare = naturalCompare
end

val naturalAdd : ℕ → ℕ → ℕ
declare hol target_rep function naturalAdd = infix '+'
declare ocaml target_rep function naturalAdd = 'Big_int.add_big_int'
declare isabelle target_rep function naturalAdd = infix '+'
declare coq target_rep function naturalAdd = 'Coq.Init.Peano.plus'

instance (NumAdd ℕ)
  let + = naturalAdd
end

val naturalMinus : ℕ → ℕ → ℕ
declare hol target_rep function naturalMinus = infix '-'
declare ocaml target_rep function naturalMinus = 'Nat_num.natural_monus'
declare isabelle target_rep function naturalMinus = infix '-'
declare coq target_rep function naturalMinus = 'Coq.Init.Peano.minus'

instance (NumMinus ℕ)
  let − = naturalMinus
end

val naturalSucc : ℕ → ℕ
let naturalSucc n = n + 1
declare hol target_rep function naturalSucc = 'SUC'
```

```
declare isabelle target_rep function naturalSucc = 'Suc'
declare ocaml target_rep function naturalSucc = 'Big_int.succ_big_int'
declare coq target_rep function naturalSucc = 'S'
instance (NumSucc ℕ)
  let succ = naturalSucc
end

val naturalPred : ℕ → ℕ
let inline naturalPred n = n − 1
declare hol target_rep function naturalPred = 'PRE'
declare ocaml target_rep function naturalPred = 'Nat_num.natural_pred'
declare coq target_rep function naturalPred = 'Coq.Init.Peano.pred'
instance (NumPred ℕ)
  let pred = naturalPred
end

val naturalMult : ℕ → ℕ → ℕ
declare hol target_rep function naturalMult = infix '*'
declare ocaml target_rep function naturalMult = 'Big_int.mult_big_int'
declare isabelle target_rep function naturalMult = infix '*'
declare coq target_rep function naturalMult = 'Coq.Init.Peano.mult'

instance (NumMult ℕ)
  let ∗ = naturalMult
end


val naturalPow : ℕ → NAT → ℕ
declare hol target_rep function naturalPow = infix '**'
declare ocaml target_rep function naturalPow = 'Big_int.power_big_int_positive_int'
declare isabelle target_rep function naturalPow = infix '↑'
declare coq target_rep function naturalPow = 'nat_power'

instance ( NumPow ℕ )
  let ∗∗ = naturalPow
end

val naturalDiv : ℕ → ℕ → ℕ
declare hol target_rep function naturalDiv = infix 'DIV'
declare ocaml target_rep function naturalDiv = 'Big_int.div_big_int'
declare isabelle target_rep function naturalDiv = infix 'div'
declare coq target_rep function naturalDiv = 'nat_div'

instance ( NumIntegerDivision ℕ )
  let div = naturalDiv
end

instance ( NumDivision ℕ )
  let / = naturalDiv
end

val naturalMod : ℕ → ℕ → ℕ
declare hol target_rep function naturalMod = infix 'MOD'
declare ocaml target_rep function naturalMod = 'Big_int.mod_big_int'
declare isabelle target_rep function naturalMod = infix 'mod'
declare coq target_rep function naturalMod = 'nat_mod'

instance ( NumRemainder ℕ )
```

let mod = naturalMod
end

val $naturalMin$: $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$
let inline $naturalMin$ = defaultMin
declare $isabelle$ target_rep function naturalMin = `'min'`
declare $ocaml$ target_rep function naturalMin = `'Big_int.min_big_int'`
declare hol target_rep function naturalMin = `'MIN'`
declare coq target_rep function naturalMin = `'nat_min'`

val $naturalMax$: $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$
let inline $naturalMax$ = defaultMax
declare $isabelle$ target_rep function naturalMax = `'max'`
declare $ocaml$ target_rep function naturalMax = `'Big_int.max_big_int'`
declare hol target_rep function naturalMax = `'MAX'`
declare coq target_rep function naturalMax = `'nat_max'`

instance ($OrdMaxMin$ $\mathbb{N}$)
let max = naturalMax
let min = naturalMin
end

```
(* ----------------------- *)
(* int                     *)
(* ----------------------- *)
```

val $intFromNumeral$: NUMERAL $\to$ INT
declare $ocaml$ target_rep function intFromNumeral = `''`
declare $isabelle$ target_rep function intFromNumeral n = (`''`n : INT)
declare hol target_rep function intFromNumeral n = (`''`n : INT)
declare coq target_rep function intFromNumeral n = (`'Zpos'` (`'P_of_succ_nat'` n))

instance ($Numeral$ INT)
let $fromNumeral$ n = intFromNumeral n
end

val $intEq$: INT $\to$ INT $\to$ $\mathbb{B}$
let inline $intEq$ = unsafe_structural_equality
instance (Eq INT)
let = = intEq
let <> n_1 n_2 = $\neg$ (intEq n_1 n_2)
end

val $intLess$: INT $\to$ INT $\to$ $\mathbb{B}$
val $intLessEqual$: INT $\to$ INT $\to$ $\mathbb{B}$
val $intGreater$: INT $\to$ INT $\to$ $\mathbb{B}$
val $intGreaterEqual$: INT $\to$ INT $\to$ $\mathbb{B}$

declare hol target_rep function intLess = infix `'<'`
declare $ocaml$ target_rep function intLess = infix `'<'`
declare $isabelle$ target_rep function intLess = infix `'<'`
declare coq target_rep function intLess = `'int_ltb'`

declare hol target_rep function intLessEqual = infix `'<='`
declare $ocaml$ target_rep function intLessEqual = infix `'<='`
declare $isabelle$ target_rep function intLessEqual = infix `'\<le>'`
declare coq target_rep function intLessEqual = `'int_lteb'`

```
declare hol target_rep function intGreater = infix '>'
declare ocaml target_rep function intGreater = infix '>'
declare isabelle target_rep function intGreater = infix '>'
declare coq target_rep function intGreater = 'int_gtb'

declare hol target_rep function intGreaterEqual = infix '>='
declare ocaml target_rep function intGreaterEqual = infix '>='
declare isabelle target_rep function intGreaterEqual = infix '\<ge>'
declare coq target_rep function intGreaterEqual = 'int_gteb'

val intCompare : INT → INT → ORDERING
let inline intCompare = defaultCompare
let inline {coq; isabelle; hol} intCompare = genericCompare intLess intEq
declare ocaml target_rep function intCompare = 'compare'

instance (Ord INT)
  let compare = intCompare
  let < = intLess
  let <= = intLessEqual
  let > = intGreater
  let >= = intGreaterEqual
end

instance (SetType INT)
  let setElemCompare = intCompare
end

val intNegate : INT → INT
declare hol target_rep function intNegate i = '~' i
declare ocaml target_rep function intNegate i = ('~-' i)
declare isabelle target_rep function intNegate i = '-' i
declare coq target_rep function intNegate i = ('Coq.ZArith.BinInt.Zminus' 'Z'_0 i)

instance (NumNegate INT)
  let ~ = intNegate
end


val intAdd : INT → INT → INT
declare hol target_rep function intAdd = infix '+'
declare ocaml target_rep function intAdd = infix '+'
declare isabelle target_rep function intAdd = infix '+'
declare coq target_rep function intAdd = 'Coq.ZArith.BinInt.Zplus'

instance (NumAdd INT)
  let + = intAdd
end

val intMinus : INT → INT → INT
declare hol target_rep function intMinus = infix '-'
declare ocaml target_rep function intMinus = infix '-'
declare isabelle target_rep function intMinus = infix '-'
declare coq target_rep function intMinus = 'Coq.ZArith.BinInt.Zminus'

instance (NumMinus INT)
  let − = intMinus
end
```

```
val intSucc : INT → INT
let inline intSucc n = n + 1
declare ocaml target_rep function intSucc = 'succ'
instance (NumSucc INT)
  let succ = intSucc
end

val intPred : INT → INT
let inline intPred n = n − 1
declare ocaml target_rep function intPred = 'pred'
instance (NumPred INT)
  let pred = intPred
end

val intMult : INT → INT → INT
declare hol target_rep function intMult = infix '*'
declare ocaml target_rep function intMult = infix '*'
declare isabelle target_rep function intMult = infix '*'
declare coq target_rep function intMult = 'Coq.ZArith.BinInt.Zmult'

instance (NumMult INT)
  let * = intMult
end


val intPow : INT → NAT → INT
declare hol target_rep function intPow = infix '**'
declare ocaml target_rep function intPow = 'Nat_num.int_pow'
declare isabelle target_rep function intPow = infix '↑'
declare coq target_rep function intPow = 'Coq.ZArith.Zpower.Zpower_nat'

instance ( NumPow INT )
  let ** = intPow
end

val intDiv : INT → INT → INT
declare hol target_rep function intDiv = infix '/'
declare ocaml target_rep function intDiv = 'Nat_num.int_div'
declare isabelle target_rep function intDiv = infix 'div'
declare coq target_rep function intDiv = 'Coq.ZArith.Zdiv.Zdiv'

instance ( NumIntegerDivision INT )
  let div = intDiv
end

instance ( NumDivision INT )
  let / = intDiv
end

val intMod : INT → INT → INT
declare hol target_rep function intMod = infix '%'
declare ocaml target_rep function intMod = 'Nat_num.int_mod'
declare isabelle target_rep function intMod = infix 'mod'
declare coq target_rep function intMod = 'Coq.ZArith.Zdiv.Zmod'

instance ( NumRemainder INT )
  let mod = intMod
```

```
end

val intMin : int → int → int
let inline intMin = defaultMin
declare isabelle target_rep function intMin = 'min'
declare ocaml target_rep function intMin = 'min'
declare hol target_rep function intMin = 'int_min'
declare coq target_rep function intMin = 'Zmin'

val intMax : int → int → int
let inline intMax = defaultMax
declare isabelle target_rep function intMax = 'max'
declare ocaml target_rep function intMax = 'max'
declare hol target_rep function intMax = 'int_max'
declare coq target_rep function intMax = 'Zmax'

instance ( OrdMaxMin int )
  let max = intMax
  let min = intMin
end


(* ----------------------- *)
(* integer                 *)
(* ----------------------- *)

val integerFromNumeral : numeral → ℤ
declare ocaml target_rep function integerFromNumeral = 'Big_int.big_int_of_int'
declare isabelle target_rep function integerFromNumeral n = (''n : ℤ)
declare hol target_rep function integerFromNumeral n = (''n : ℤ)
declare coq target_rep function integerFromNumeral n = ('Zpos' ('P_of_succ_nat' n))

instance (Numeral ℤ)
  let fromNumeral n = integerFromNumeral n
end

val integerEq : ℤ → ℤ → 𝔹
let inline integerEq = unsafe_structural_equality
declare ocaml target_rep function integerEq = 'Big_int.eq_big_int'
instance (Eq ℤ)
  let = = integerEq
  let <> n1 n2 = ¬ (integerEq n1 n2)
end

val integerLess : ℤ → ℤ → 𝔹
val integerLessEqual : ℤ → ℤ → 𝔹
val integerGreater : ℤ → ℤ → 𝔹
val integerGreaterEqual : ℤ → ℤ → 𝔹

declare hol target_rep function integerLess = infix '<'
declare ocaml target_rep function integerLess = 'Big_int.lt_big_int'
declare isabelle target_rep function integerLess = infix '<'
declare coq target_rep function integerLess = 'int_ltb'

declare hol target_rep function integerLessEqual = infix '<='
declare ocaml target_rep function integerLessEqual = 'Big_int.le_big_int'
declare isabelle target_rep function integerLessEqual = infix '\<le>'
declare coq target_rep function integerLessEqual = 'int_lteb'
```

```
declare hol target_rep function integerGreater = infix '>'
declare ocaml target_rep function integerGreater = 'Big_int.gt_big_int'
declare isabelle target_rep function integerGreater = infix '>'
declare coq target_rep function integerGreater = 'int_gtb'

declare hol target_rep function integerGreaterEqual = infix '>='
declare ocaml target_rep function integerGreaterEqual = 'Big_int.ge_big_int'
declare isabelle target_rep function integerGreaterEqual = infix '\<ge>'
declare coq target_rep function integerGreaterEqual = 'int_gteb'

val integerCompare : ℤ → ℤ → ORDERING
let inline integerCompare = defaultCompare
let inline {coq; isabelle; hol} integerCompare = genericCompare integerLess integerEq
declare ocaml target_rep function integerCompare = 'Big_int.compare_big_int'

instance (Ord ℤ)
  let compare = integerCompare
  let < = integerLess
  let <= = integerLessEqual
  let > = integerGreater
  let >= = integerGreaterEqual
end

instance (SetType ℤ)
  let setElemCompare = integerCompare
end

val integerNegate : ℤ → ℤ
declare hol target_rep function integerNegate i = '~' i
declare ocaml target_rep function integerNegate = 'Big_int.minus_big_int'
declare isabelle target_rep function integerNegate i = '-' i
declare coq target_rep function integerNegate i = ('Coq.ZArith.BinInt.Zminus' 'Z'0 i)

instance (NumNegate ℤ)
  let ~ = integerNegate
end


val integerAdd : ℤ → ℤ → ℤ
declare hol target_rep function integerAdd = infix '+'
declare ocaml target_rep function integerAdd = 'Big_int.add_big_int'
declare isabelle target_rep function integerAdd = infix '+'
declare coq target_rep function integerAdd = 'Coq.ZArith.BinInt.Zplus'

instance (NumAdd ℤ)
  let + = integerAdd
end

val integerMinus : ℤ → ℤ → ℤ
declare hol target_rep function integerMinus = infix '-'
declare ocaml target_rep function integerMinus = 'Big_int.sub_big_int'
declare isabelle target_rep function integerMinus = infix '-'
declare coq target_rep function integerMinus = 'Coq.ZArith.BinInt.Zminus'

instance (NumMinus ℤ)
  let − = integerMinus
end
```

```
val integerSucc : ℤ → ℤ
let inline integerSucc n = n + 1
declare ocaml target_rep function integerSucc = 'Big_int.succ_big_int'
instance (NumSucc ℤ)
  let succ = integerSucc
end

val integerPred : ℤ → ℤ
let inline integerPred n = n − 1
declare ocaml target_rep function integerPred = 'Big_int.pred_big_int'
instance (NumPred ℤ)
  let pred = integerPred
end

val integerMult : ℤ → ℤ → ℤ
declare hol target_rep function integerMult = infix '*'
declare ocaml target_rep function integerMult = 'Big_int.mult_big_int'
declare isabelle target_rep function integerMult = infix '*'
declare coq target_rep function integerMult = 'Coq.ZArith.BinInt.Zmult'

instance (NumMult ℤ)
  let ∗ = integerMult
end


val integerPow : ℤ → NAT → ℤ
declare hol target_rep function integerPow = infix '**'
declare ocaml target_rep function integerPow = 'Big_int.power_big_int_positive_int'
declare isabelle target_rep function integerPow = infix '↑'
declare coq target_rep function integerPow = 'Coq.ZArith.Zpower.Zpower_nat'

instance ( NumPow ℤ )
  let ∗∗ = integerPow
end

val integerDiv : ℤ → ℤ → ℤ
declare hol target_rep function integerDiv = infix '/'
declare ocaml target_rep function integerDiv = 'Big_int.div_big_int'
declare isabelle target_rep function integerDiv = infix 'div'
declare coq target_rep function integerDiv = 'Coq.ZArith.Zdiv.Zdiv'

instance ( NumIntegerDivision ℤ )
  let div = integerDiv
end

instance ( NumDivision ℤ )
  let / = integerDiv
end

val integerMod : ℤ → ℤ → ℤ
declare hol target_rep function integerMod = infix '%'
declare ocaml target_rep function integerMod = 'Big_int.mod_big_int'
declare isabelle target_rep function integerMod = infix 'mod'
declare coq target_rep function integerMod = 'Coq.ZArith.Zdiv.Zmod'

instance ( NumRemainder ℤ )
  let mod = integerMod
```

```
end

val integerMin : ℤ → ℤ → ℤ
let inline integerMin = defaultMin
declare isabelle target_rep function integerMin = 'min'
declare ocaml target_rep function integerMin = 'Big_int.min_big_int'
declare hol target_rep function integerMin = 'int_min'
declare coq target_rep function integerMin = 'Zmin'

val integerMax : ℤ → ℤ → ℤ
let inline integerMax = defaultMax
declare isabelle target_rep function integerMax = 'max'
declare ocaml target_rep function integerMax = 'Big_int.max_big_int'
declare hol target_rep function integerMax = 'int_max'
declare coq target_rep function integerMax = 'Zmax'

instance ( OrdMaxMin ℤ )
  let max = integerMax
  let min = integerMin
end




(* ========================================================================== *)
(* Tests                                                                      *)
(* ========================================================================== *)

assert nat_test1 : (2 + (5 : NAT) = 7)
assert nat_test2 : (8 − (7 : NAT) = 1)
assert nat_test3 : (7 − (8 : NAT) = 0)
assert nat_test4 : (7 ∗ (8 : NAT) = 56)
assert nat_test5 : ((7 : NAT)^2 = 49)
assert nat_test6 : (div 11 (4 : NAT) = 2)
assert nat_test7 : (11 / (4 : NAT) = 2)
assert nat_test8 : (11 mod (4 : NAT) = 3)
assert nat_test9 : (11 < (12 : NAT))
assert nat_test10 : (11 ≤ (12 : NAT))
assert nat_test11 : (12 ≤ (12 : NAT))
assert nat_test12 : (¬ (12 < (12 : NAT)))
assert nat_test13 : (12 > (11 : NAT))
assert nat_test14 : (12 ≥ (11 : NAT))
assert nat_test15 : (12 ≥ (12 : NAT))
assert nat_test16 : (¬ (12 > (12 : NAT)))
assert nat_test17 : (min 12 (12 : NAT) = 12)
assert nat_test18 : (min 10 (12 : NAT) = 10)
assert nat_test19 : (min 12 (10 : NAT) = 10)
assert nat_test20 : (max 12 (12 : NAT) = 12)
assert nat_test21 : (max 10 (12 : NAT) = 12)
assert nat_test22 : (max 12 (10 : NAT) = 12)
assert nat_test23 : (succ 12 = (13 : NAT))
assert nat_test24 : (succ 0 = (1 : NAT))
assert nat_test25 : (pred 12 = (11 : NAT))
assert nat_test26 : (pred 0 = (0 : NAT))
assert nat_test27 : (match (27 : NAT) with
    | 0 → false
    | x + 2 → (x = 25)
    | x + 1 → (x = 26)
  end)
```

assert $nat\_test28a$: (match $(27 : \text{NAT})$ with
| $n + 50 \rightarrow$ "$50 <= x$"
| $40 \rightarrow$ "$x = 40$"
| $n + 31 \rightarrow$ "$x <> 40 \&\& 31 <= x < 50$"
| $29 \rightarrow$ "$x = 29$"
| $n + 30 \rightarrow$ "$x = 30$"
| $4 \rightarrow$ "$x = 4$"
| $\_ \rightarrow$ "$x <> 4 \&\& x <> 29 \&\& x < 30$"
end = "$x <> 4 \&\& x <> 29 \&\& x < 30$")
assert $nat\_test28b$: (match $(30 : \text{NAT})$ with
| $n + 50 \rightarrow$ "$50 <= x$"
| $40 \rightarrow$ "$x = 40$"
| $n + 31 \rightarrow$ "$x <> 40 \&\& 31 <= x < 50$"
| $29 \rightarrow$ "$x = 29$"
| $n + 30 \rightarrow$ "$x = 30$"
| $4 \rightarrow$ "$x = 4$"
| $\_ \rightarrow$ "$x <> 4 \&\& x <> 29 \&\& x < 30$"
end = "$x = 30$")

assert $natural\_test_1$: $(2 + (5 : \mathbb{N}) = 7)$
assert $natural\_test_2$: $(8 - (7 : \mathbb{N}) = 1)$
assert $natural\_test_3$: $(7 - (8 : \mathbb{N}) = 0)$
assert $natural\_test_4$: $(7 * (8 : \mathbb{N}) = 56)$
assert $natural\_test_5$: $((7 : \mathbb{N})^2 = 49)$
assert $natural\_test_6$: $(\text{div } 11\ (4 : \mathbb{N}) = 2)$
assert $natural\_test_7$: $(11 \ / \ (4 : \mathbb{N}) = 2)$
assert $natural\_test_8$: $(11 \text{ mod } (4 : \mathbb{N}) = 3)$
assert $natural\_test_9$: $(11 < (12 : \mathbb{N}))$
assert $natural\_test_{10}$: $(11 \le (12 : \mathbb{N}))$
assert $natural\_test_{11}$: $(12 \le (12 : \mathbb{N}))$
assert $natural\_test_{12}$: $(\neg (12 < (12 : \mathbb{N})))$
assert $natural\_test_{13}$: $(12 > (11 : \mathbb{N}))$
assert $natural\_test_{14}$: $(12 \ge (11 : \mathbb{N}))$
assert $natural\_test_{15}$: $(12 \ge (12 : \mathbb{N}))$
assert $natural\_test_{16}$: $(\neg (12 > (12 : \mathbb{N})))$
assert $natural\_test_{17}$: $(\text{min } 12\ (12 : \mathbb{N}) = 12)$
assert $natural\_test_{18}$: $(\text{min } 10\ (12 : \mathbb{N}) = 10)$
assert $natural\_test_{19}$: $(\text{min } 12\ (10 : \mathbb{N}) = 10)$
assert $natural\_test_{20}$: $(\text{max } 12\ (12 : \mathbb{N}) = 12)$
assert $natural\_test_{21}$: $(\text{max } 10\ (12 : \mathbb{N}) = 12)$
assert $natural\_test_{22}$: $(\text{max } 12\ (10 : \mathbb{N}) = 12)$
assert $natural\_test_{23}$: $(\text{succ } 12 = (13 : \mathbb{N}))$
assert $natural\_test_{24}$: $(\text{succ } 0 = (1 : \mathbb{N}))$
assert $natural\_test_{25}$: $(\text{pred } 12 = (11 : \mathbb{N}))$
assert $natural\_test_{26}$: $(\text{pred } 0 = (0 : \mathbb{N}))$
assert $natural\_test_{27}$: (match $(27 : \mathbb{N})$ with
| $0 \rightarrow$ false
| $x + 2 \rightarrow (x = 25)$
| $x + 1 \rightarrow (x = 26)$
end)
assert $natural\_test28a$: (match $(27 : \mathbb{N})$ with
| $n + 50 \rightarrow$ "$50 <= x$"
| $40 \rightarrow$ "$x = 40$"
| $n + 31 \rightarrow$ "$x <> 40 \&\& 31 <= x < 50$"
| $29 \rightarrow$ "$x = 29$"
| $n + 30 \rightarrow$ "$x = 30$"

$\mid 4 \ \rightarrow\ "x = 4"$
$\mid \_ \ \rightarrow\ "x<>4\&\&x<>29\&\&x<30"$
end $= "x<>4\&\&x<>29\&\&x<30")$
assert $natural\_test28b$: (match $(30 : \mathbb{N})$ with
$\mid n\ +\ 50 \ \rightarrow\ "50<=x"$
$\mid 40 \ \rightarrow\ "x = 40"$
$\mid n\ +\ 31 \ \rightarrow\ "x<>40\&\&31<=x<50"$
$\mid 29 \ \rightarrow\ "x = 29"$
$\mid n\ +\ 30 \ \rightarrow\ "x = 30"$
$\mid 4 \ \rightarrow\ "x = 4"$
$\mid \_ \ \rightarrow\ "x<>4\&\&x<>29\&\&x<30"$
end $= "x = 30")$

assert int_test_1 : $(2 + (5\ :\ \text{INT}) = 7)$
assert int_test_2 : $(8 - (7\ :\ \text{INT}) = 1)$
assert int_test_3 : $(7 - (8\ :\ \text{INT}) = -1)$
assert int_test_4 : $(7 * (8\ :\ \text{INT}) = 56)$
assert int_test_5 : $((7 : \text{INT})^2 = 49)$
assert int_test_6 : $(\text{div } 11\ (4\ :\ \text{INT}) = 2)$
assert int_test6a : $(\text{div } (-\ 11)\ (4\ :\ \text{INT}) = -3)$
assert int_test_7 : $(11\ /\ (4\ :\ \text{INT}) = 2)$
assert int_test7a : $(-11\ /\ (4\ :\ \text{INT}) = -3)$
assert int_test_8 : $(11 \bmod (4\ :\ \text{INT}) = 3)$
assert $int\_test8at$: $(-11 \bmod (4\ :\ \text{INT}) = 1)$
assert int_test_9 : $(11 < (12\ :\ \text{INT}))$
assert int_test_{10} : $(11 \leq (12\ :\ \text{INT}))$
assert int_test_{11} : $(12 \leq (12\ :\ \text{INT}))$
assert int_test_{12} : $(\neg\ (12 < (12\ :\ \text{INT})))$
assert int_test_{13} : $(12 > (11\ :\ \text{INT}))$
assert int_test_{14} : $(12 \geq (11\ :\ \text{INT}))$
assert int_test_{15} : $(12 \geq (12\ :\ \text{INT}))$
assert int_test_{16} : $(\neg\ (12 > (12\ :\ \text{INT})))$
assert int_test_{17} : $(\min 12\ (12\ :\ \text{INT}) = 12)$
assert int_test_{18} : $(\min 10\ (12\ :\ \text{INT}) = 10)$
assert int_test_{19} : $(\min 12\ (10\ :\ \text{INT}) = 10)$
assert int_test_{20} : $(\max 12\ (12\ :\ \text{INT}) = 12)$
assert int_test_{21} : $(\max 10\ (12\ :\ \text{INT}) = 12)$
assert int_test_{22} : $(\max 12\ (10\ :\ \text{INT}) = 12)$
assert int_test_{23} : $(\text{succ } 12 = (13\ :\ \text{INT}))$
assert int_test_{24} : $(\text{succ } 0 = (1\ :\ \text{INT}))$
assert int_test_{25} : $(\text{pred } 12 = (11\ :\ \text{INT}))$
assert int_test_{26} : $(\text{pred } 0 = -(1\ :\ \text{INT}))$

assert $integer\_test_1$: $(2 + (5\ :\ \mathbb{Z}) = 7)$
assert $integer\_test_2$: $(8 - (7\ :\ \mathbb{Z}) = 1)$
assert $integer\_test_3$: $(7 - (8\ :\ \mathbb{Z}) = -1)$
assert $integer\_test_4$: $(7 * (8\ :\ \mathbb{Z}) = 56)$
assert $integer\_test_5$: $((7 : \mathbb{Z})^2 = 49)$
assert $integer\_test_6$: $(\text{div } 11\ (4\ :\ \mathbb{Z}) = 2)$
assert $integer\_test6a$: $(\text{div } (-\ 11)\ (4\ :\ \mathbb{Z}) = -3)$
assert $integer\_test_7$: $(11\ /\ (4\ :\ \mathbb{Z}) = 2)$
assert $integer\_test7a$: $(-11\ /\ (4\ :\ \mathbb{Z}) = -3)$
assert $integer\_test_8$: $(11 \bmod (4\ :\ \mathbb{Z}) = 3)$
assert $integer\_test8a$: $(-11 \bmod (4\ :\ \mathbb{Z}) = 1)$
assert $integer\_test_9$: $(11 < (12\ :\ \mathbb{Z}))$
assert $integer\_test_{10}$: $(11 \leq (12\ :\ \mathbb{Z}))$
assert $integer\_test_{11}$: $(12 \leq (12\ :\ \mathbb{Z}))$

assert $integer\_test_{12}$: $(\neg\ (12 < (12\ :\ \mathbb{Z})))$
assert $integer\_test_{13}$: $(12 > (11\ :\ \mathbb{Z}))$
assert $integer\_test_{14}$: $(12 \geq (11\ :\ \mathbb{Z}))$
assert $integer\_test_{15}$: $(12 \geq (12\ :\ \mathbb{Z}))$
assert $integer\_test_{16}$: $(\neg\ (12 > (12\ :\ \mathbb{Z})))$
assert $integer\_test_{17}$: $(\min 12\ (12\ :\ \mathbb{Z}) = 12)$
assert $integer\_test_{18}$: $(\min 10\ (12\ :\ \mathbb{Z}) = 10)$
assert $integer\_test_{19}$: $(\min 12\ (10\ :\ \mathbb{Z}) = 10)$
assert $integer\_test_{20}$: $(\max 12\ (12\ :\ \mathbb{Z}) = 12)$
assert $integer\_test_{21}$: $(\max 10\ (12\ :\ \mathbb{Z}) = 12)$
assert $integer\_test_{22}$: $(\max 12\ (10\ :\ \mathbb{Z}) = 12)$
assert $integer\_test_{23}$: $(\mathrm{succ}\ 12 = (13\ :\ \mathbb{Z}))$
assert $integer\_test_{24}$: $(\mathrm{succ}\ 0 = (1\ :\ \mathbb{Z}))$
assert $integer\_test_{25}$: $(\mathrm{pred}\ 12 = (11\ :\ \mathbb{Z}))$
assert $integer\_test_{26}$: $(\mathrm{pred}\ 0 = -(1\ :\ \mathbb{Z}))$

```
(* ========================================================================== *)
(* Translation between number types                                           *)
(* ========================================================================== *)
```

val $naturalFromNat$: NAT $\rightarrow$ $\mathbb{N}$
declare hol target_rep function naturalFromNat = `''` `(* remove natFromNumeral, as it is the identify function *)`
declare $ocaml$ target_rep function naturalFromNat = `'Big_int.big_int_of_int'`
declare $isabelle$ target_rep function naturalFromNat = `''`
declare coq target_rep function naturalFromNat = `'id'`

assert $natural\_from\_nat_0$: naturalFromNat $0 = 0$
assert $natural\_from\_nat_1$: naturalFromNat $1 = 1$
assert $natural\_from\_nat_2$: naturalFromNat $2 = 2$

val $natFromNatural$: $\mathbb{N}$ $\rightarrow$ NAT
declare hol target_rep function natFromNatural = `''` `(* remove natFromNumeral, as it is the identify function *)`
declare $ocaml$ target_rep function natFromNatural = `'Big_int.int_of_big_int'`
declare $isabelle$ target_rep function natFromNatural = `''`
declare coq target_rep function natFromNatural = `'id'`

assert $nat\_from\_natural_0$: natFromNatural $0 = 0$
assert $nat\_from\_natural_1$: natFromNatural $1 = 1$
assert $nat\_from\_natural_2$: natFromNatural $2 = 2$

val $intFromNat$: NAT $\rightarrow$ INT
declare hol target_rep function intFromNat = `'int_of_num'`
declare $ocaml$ target_rep function intFromNat n = `''`n
declare $isabelle$ target_rep function intFromNat = `'int'`
declare coq target_rep function intFromNat n = (`'Zpos'` (`'P_of_succ_nat'` n))

assert $int\_from\_nat_0$: intFromNat $0 = 0$
assert $int\_from\_nat_1$: intFromNat $1 = 1$
assert $int\_from\_nat_2$: intFromNat $2 = 2$

val $natFromInt$: INT $\rightarrow$ NAT
declare hol target_rep function natFromInt i = `'Num'` (`'ABS'` i)
declare $ocaml$ target_rep function natFromInt = `'abs'`
declare coq target_rep function natFromInt = `'Zabs_nat'`

declare *isabelle* target_rep function natFromInt i = `'nat'` (`'abs'` i)

assert $\mathit{nat\_from\_int}_0$: natFromInt $0 = 0$
assert $\mathit{nat\_from\_int}_1$: natFromInt $1 = 1$
assert $\mathit{nat\_from\_int}_2$: natFromInt $(-\ 2) = 2$

6 Function_extra

declare {*isabelle*; *hol*; *ocaml*} rename module = lem_function_extra

open import *Maybe Bool Basic_classes Num Function*

```
(* ----------------------- *)
(* Tests for function      *)
(* ----------------------- *)

(* These tests are not written in function itself, because the nat type
   is not available there, yet *)
```

assert id_0 : id $(2 : \text{NAT}) = 2$
assert id_1 : id $(5 : \text{NAT}) = 5$
assert id_2 : id $(2 : \text{NAT}) = 2$

assert $const_0$: $(\text{const } (2 : \text{NAT}))$ true $= 2$
assert $const_1$: $(\text{const } (5 : \text{NAT}))$ false $= 5$
assert $const_2$: $(\text{const } (2 : \text{NAT})) \; (3 : \text{NAT}) = 2$

assert $comb_0$: (comb (fun $(x : \text{NAT}) \rightarrow 3 * x)$ succ $2 = 9)$
assert $comb_1$: (comb succ (fun $(x : \text{NAT}) \rightarrow 3 * x)$ $2 = 7)$

assert $apply_0$: (\$) (fun $(x : \text{NAT}) \rightarrow 3 * x)$ $2 = 6$
assert $apply_1$: (fun $(x : \text{NAT}) \rightarrow 3 * x)$ \$ $2 = 6$

assert $flip_0$: flip (fun $(x : \text{NAT})\; y \rightarrow x - y)$ $3\;5 = 2$
assert $flip_1$: flip (fun $(x : \text{NAT})\; y \rightarrow x - y)$ $5\;3 = 0$

```
(* ----------------------- *)
(* getting a unique value  *)
(* ----------------------- *)
```

val *THE* : $\forall\, \alpha.\, (\alpha \rightarrow \mathbb{B}) \rightarrow \text{MAYBE } \alpha$
declare *hol* target_rep function THE = `'$THE'`
declare *ocaml* target_rep function THE = `'THE'`
declare *isabelle* target_rep function THE = `'The_opt'`

lemma ~{*coq*} *THE_spec* : $(\forall\, p\; x.\, (\text{THE } p = \text{Just } x) \longleftrightarrow ((p\; x) \wedge (\forall\, y.\, p\; y \longrightarrow (x = y))))$

7 Tuple

```
(******************************************************************************)
(* Tuples                                                                     *)
(******************************************************************************)

(* The type for tuples (pairs) is hard-coded, so here only a few functions are used *)
```

declare {*isabelle*; *hol*; *ocaml*} rename module = lem_tuple

open import *Bool Basic_classes*

```
(* ----------------------- *)
(* fst                     *)
(* ----------------------- *)
```

val *fst* : $\forall\, \alpha\, \beta.\ \alpha * \beta \rightarrow \alpha$
let *fst* $(v_1,\ v_2) = v_1$

declare *hol* target_rep function fst = `'FST'`
declare *ocaml* target_rep function fst = `'fst'`
declare *isabelle* target_rep function fst = `'fst'`
declare *coq* target_rep function fst = `'fst'`

assert fst_1 : (fst (true, false) = true)
assert fst_2 : (fst (false, true) = false)

```
(* ----------------------- *)
(* snd                     *)
(* ----------------------- *)
```

val *snd* : $\forall\, \alpha\, \beta.\ \alpha * \beta \rightarrow \beta$
let *snd* $(v_1,\ v_2) = v_2$

declare *hol* target_rep function snd = `'SND'`
declare *ocaml* target_rep function snd = `'snd'`
declare *isabelle* target_rep function snd = `'snd'`
declare *coq* target_rep function snd = `'snd'`

lemma *fst_snd* : $(\forall\, v.\ v = (\mathrm{fst}\ v,\ \mathrm{snd}\ v))$

assert snd_1 : (snd (true, false) = false)
assert snd_2 : (snd (false, true) = true)

```
(* ----------------------- *)
(* curry                   *)
(* ----------------------- *)
```

val *curry* : $\forall\, \alpha\, \beta\, \gamma.\ (\alpha * \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma)$
let inline *curry* $f\ v_1\ v_2 = f\ (v_1,\ v_2)$

declare *hol* target_rep function curry = `'CURRY'`
declare *isabelle* target_rep function curry = `'curry'`
declare *ocaml* target_rep function curry = `'Lem.curry'`
declare *coq* target_rep function curry = `'prod_curry'`

assert $curry_1$: (curry (fun $(x,\ y) \rightarrow x \wedge y$) true false = false)

```
(* ----------------------- *)
(* uncurry                 *)
(* ----------------------- *)
```

val $uncurry$: $\forall\ \alpha\ \beta\ \gamma.\ (\alpha\ \rightarrow\ \beta\ \rightarrow\ \gamma)\ \rightarrow\ (\alpha\ *\ \beta\ \rightarrow\ \gamma)$
let inline $uncurry\ f\ =$ (fun $(v_1,\ v_2)\ \rightarrow\ f\ v_1\ v_2)$

declare hol target_rep function uncurry = `'UNCURRY'`
declare $isabelle$ target_rep function uncurry = `'split'`
declare $ocaml$ target_rep function uncurry = `'Lem.uncurry'`
declare coq target_rep function uncurry = `'prod_uncurry'`

lemma $curry\_uncurry$: $(\forall\ f\ xy.\ \mathrm{uncurry}\ (\mathrm{curry}\ f)\ xy = f\ xy)$
lemma $uncurry\_curry$: $(\forall\ f\ x\ y.\ \mathrm{curry}\ (\mathrm{uncurry}\ f)\ x\ y = f\ x\ y)$

assert $uncurry_1$: (uncurry (fun $x\ y\ \rightarrow\ x \wedge y$) (true, false) = false)

```
(* ----------------------- *)
(* swap                    *)
(* ----------------------- *)
```

val $swap$: $\forall\ \alpha\ \beta.\ (\alpha\ *\ \beta)\ \rightarrow\ (\beta\ *\ \alpha)$
let $swap\ (v_1,\ v_2)\ =\ (v_2,\ v_1)$

let inline $\{isabelle;\ coq\}$ $swap$ = (fun $(v_1,\ v_2)\ \rightarrow\ (v_2,\ v_1)$)
declare hol target_rep function swap = `'SWAP'`
declare $ocaml$ target_rep function swap = `'Lem.pair_swap'`

assert $swap_1$: (swap (false, true) = (true, false))

8 List

```
(******************************************************************************)
(* A library for lists                                                        *)
(*                                                                            *)
(* It mainly follows the Haskell List-library                                 *)
(******************************************************************************)



(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

declare {*isabelle*; *ocaml*; *hol*} rename module = Lem_list

open import *Bool Maybe Basic_classes Tuple Num*

open import {*coq*} *Coq.Lists.TheoryList*
open import {*isabelle*} *$LIB_DIR/Lem*
open import {*hol*} *listTheory rich_listTheory sortingTheory*

```
(* ========================================================================== *)
(* Basic list functions                                                       *)
(* ========================================================================== *)

(* The type of lists as well as list literals like [], [1;2], ... are hardcoded.
   Thus, we can directly dive into derived definitions. *)


(* ----------------------- *)
(* cons                    *)
(* ----------------------- *)
```

val :: : $\forall\ \alpha.\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha$

declare ascii_rep function :: = cons
declare *hol* target_rep function cons = infix `'::'`
declare *ocaml* target_rep function cons = infix `'::'`
declare *isabelle* target_rep function cons = infix `'#'`
declare *coq* target_rep function cons = infix `'::'`

```
(* ----------------------- *)
(* Emptyness check         *)
(* ----------------------- *)
```

val *null* : $\forall\ \alpha.\ \text{LIST}\ \alpha\ \rightarrow\ \mathbb{B}$
let *null* *l* = match *l* with [] → true | _ → false end

declare *hol* target_rep function null = `'NULL'`
declare {*ocaml*} rename function null = list_null
`(* let inline {isabelle} null l = (l = []) *)`

assert $null\_simple_1$: (null ([] : LIST NAT))
assert $null\_simple_2$: (¬ (null [(2 : NAT); 3; 4]))
assert $null\_simple_3$: (¬ (null [(2 : NAT)]))

```
(* ----------------------- *)
(* Length                  *)
(* ----------------------- *)
```

val length : $\forall\ \alpha.\ \text{LIST}\ \alpha\ \rightarrow\ \text{NAT}$
let rec $\mathit{length}\ l\ =$
  match l with
    $|\ []\ \rightarrow\ 0$
    $|\ x\ ::\ xs\ \rightarrow\ \text{length}\ xs + 1$
  end

declare termination_argument length = automatic

declare hol target_rep function length = `'LENGTH'`
declare ocaml target_rep function length = `'List.length'`
declare $\mathit{isabelle}$ target_rep function length = `'List.length'`
declare coq target_rep function length = `'Length_l'`

assert length_0 : $(\text{length}\ ([] : \text{LIST NAT}) = 0)$
assert length_1 : $(\text{length}\ ([2] : \text{LIST NAT}) = 1)$
assert length_2 : $(\text{length}\ ([2;3] : \text{LIST NAT}) = 2)$

lemma $\mathit{length\_spec}$: $((\text{length}\ [] = 0) \wedge (\forall\ x\ xs.\ \text{length}\ (x :: xs) = \text{length}\ xs + 1))$

```
(* ----------------------- *)
(* Equality                *)
(* ----------------------- *)
```

val $\mathit{listEqual}$: $\forall\ \alpha.\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \mathbb{B}$
val $\mathit{listEqualBy}$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \mathbb{B}$

let rec $\mathit{listEqualBy}\ \mathit{eq}\ l_1\ l_2\ =$ match $(l_1,\ l_2)$ with
  $|\ ([],\ [])\ \rightarrow$ true
  $|\ ([],\ (\_ :: \_))\ \rightarrow$ false
  $|\ ((\_ :: \_),\ [])\ \rightarrow$ false
  $|\ (x :: xs,\ y\ ::\ ys)\ \rightarrow\ (\mathit{eq}\ x\ y \wedge \text{listEqualBy}\ \mathit{eq}\ xs\ ys)$
end
declare termination_argument listEqualBy = automatic

let inline $\mathit{listEqual}$ = listEqualBy (=)
declare hol target_rep function listEqual = infix `'='`
declare $\mathit{isabelle}$ target_rep function listEqual = infix `'='`
declare ocaml target_rep function listEqualBy = `'List.for_all'`$_2$
declare coq target_rep function listEqualBy = `'list_equal_by'`

instance $\forall\ \alpha.\ \mathit{Eq}\ \alpha\ \Rightarrow\ (\mathit{Eq}\ (\text{LIST}\ \alpha))$
  let = = listEqual
  let <> $l_1\ l_2\ =\ \neg\ (\text{listEqual}\ l_1\ l_2)$
end

```
(* ----------------------- *)
(* compare                 *)
(* ----------------------- *)
```

val $\mathit{lexicographicCompare}$: $\forall\ \alpha.\ \mathit{Ord}\ \alpha\ \Rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{ORDERING}$

```
val lexicographicCompareBy : ∀ α. (α → α → ORDERING) → LIST α → LIST α → ORDERING

let rec lexicographicCompareBy cmp l1 l2 = match (l1, l2) with
  | ([], []) → EQ
  | ([], _ :: _) → LT
  | (_ :: _, []) → GT
  | (x :: xs, y :: ys) → begin
      match cmp x y with
        | LT → LT
        | GT → GT
        | EQ → lexicographicCompareBy cmp xs ys
      end
    end
end
declare termination_argument lexicographicCompareBy = automatic

let inline lexicographicCompare = lexicographicCompareBy compare
declare {ocaml; hol} rename function lexicographicCompareBy = lexicographic_compare

val lexicographicLess : ∀ α. Ord α ⇒ LIST α → LIST α → 𝔹
val lexicographicLessBy : ∀ α. (α → α → 𝔹) → (α → α → 𝔹) → LIST α → LIST α → 𝔹
let rec lexicographicLessBy less less_eq l1 l2 = match (l1, l2) with
  | ([], []) → false
  | ([], _ :: _) → true
  | (_ :: _, []) → false
  | (x :: xs, y :: ys) → ((less x y) ∨ ((less_eq x y) ∧ (lexicographicLessBy less less_eq xs ys)))
end
declare termination_argument lexicographicLessBy = automatic

let inline lexicographicLess = lexicographicLessBy (<) (≤)
declare {ocaml; hol} rename function lexicographicLessBy = lexicographic_less

val lexicographicLessEq : ∀ α. Ord α ⇒ LIST α → LIST α → 𝔹
val lexicographicLessEqBy : ∀ α. (α → α → 𝔹) → (α → α → 𝔹) → LIST α → LIST α → 𝔹
let rec lexicographicLessEqBy less less_eq l1 l2 = match (l1, l2) with
  | ([], []) → true
  | ([], _ :: _) → true
  | (_ :: _, []) → false
  | (x :: xs, y :: ys) → (less x y ∨ (less_eq x y ∧ lexicographicLessEqBy less less_eq xs ys))
end
declare termination_argument lexicographicLessEqBy = automatic

let inline lexicographicLessEq = lexicographicLessEqBy (<) (≤)
declare {ocaml; hol} rename function lexicographicLessEqBy = lexicographic_less_eq


instance ∀ α. Ord α ⇒ (Ord (LIST α))
  let compare = lexicographicCompare
  let < = lexicographicLess
  let <= = lexicographicLessEq
  let > x y = lexicographicLess y x
  let >= x y = lexicographicLessEq y x
end


assert list_ord1 : ([] < [(2 : NAT)])
assert list_ord2 : ([] ≤ [(2 : NAT)])
assert list_ord3 : ([1] ≤ [(2 : NAT)])
```

assert $list\_ord_4$: $([2] \leq [(2 : \text{NAT})])$
assert $list\_ord_5$: $([2; 3] > [(2 : \text{NAT})])$
assert $list\_ord_6$: $([2; 3; 4; 5] > [(2 : \text{NAT})])$
assert $list\_ord_7$: $([2; 3; 4] > [(2 : \text{NAT}); 1; 5; 67])$
assert $list\_ord_8$: $([4] > [(3 : \text{NAT}); 56])$
assert $list\_ord_9$: $([5] \geq [(5 : \text{NAT})])$

```
(* ----------------------- *)
(* Append                  *)
(* ----------------------- *)
```

val $++$: $\forall\ \alpha.\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha$ `(* originally append *)`
let rec $++\ xs\ ys$ = match xs with
$\quad\quad |\ []\ \rightarrow\ ys$
$\quad\quad |\ x\ ::\ xs'\ \rightarrow\ x :: (\text{append}\ xs'\ ys)$
$\quad$ end

declare ascii_rep function $++$ = append
declare termination_argument append = automatic

declare *hol* target_rep function append = infix `'++'`
declare *ocaml* target_rep function append = `'List.append'`
declare *isabelle* target_rep function append = infix `'@'`
declare *coq* target_rep function append = `'app'`
declare *tex* target_rep function append = infix `'$+\!+$'`

assert $append_1$: $([0; 1; 2; 3] ++ [4; 5] = [(0 : \text{NAT}); 1; 2; 3; 4; 5])$
lemma $append\_nil_1$: $(\forall\ l.\ l ++ [] = l)$
lemma $append\_nil_2$: $(\forall\ l.\ [] ++ l = l)$

```
(* ----------------------- *)
(* snoc                    *)
(* ----------------------- *)
```

val $snoc$: $\forall\ \alpha.\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha$
let $snoc\ e\ l\ =\ l ++ [e]$

declare *hol* target_rep function snoc = `'SNOC'`
let inline $\{isabelle;\ coq\}$ $snoc\ e\ l\ =\ l ++ [e]$

assert $snoc_1$: $\text{snoc}\ (2 : \text{NAT})\ [] = [2]$
assert $snoc_2$: $\text{snoc}\ (2 : \text{NAT})\ [3; 4] = [3; 4; 2]$
assert $snoc_3$: $\text{snoc}\ (2 : \text{NAT})\ [1] = [1; 2]$
lemma $snoc\_length$: $\forall\ e\ l.\ \text{length}\ (\text{snoc}\ e\ l) = \text{succ}\ (\text{length}\ l)$
lemma $snoc\_append$: $\forall\ e\ l_1\ l_2.\ (\text{snoc}\ e\ (l_1\ ++\ l_2) = l_1\ ++\ (\text{snoc}\ e\ l_2))$

```
(* ----------------------- *)
(* Map                     *)
(* ----------------------- *)
```

val map : $\forall\ \alpha\ \beta.\ (\alpha\ \rightarrow\ \beta)\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \beta$
let rec $map\ f\ l$ = match l with
$\ \ |\ []\ \rightarrow\ []$
$\ \ |\ x\ ::\ xs\ \rightarrow\ (f\ x) :: \text{map}\ f\ xs$
end
declare termination_argument map = automatic

declare *hol* target_rep function map = 'MAP'
declare *ocaml* target_rep function map = 'List.map'
declare *isabelle* target_rep function map = 'List.map'
declare *coq* target_rep function map = 'List.map'

assert *map_nil* : $(\text{map} (\text{fun } x \rightarrow x + (1 : \text{NAT})) [] = [])$
assert map_1 : $(\text{map} (\text{fun } x \rightarrow x + (1 : \text{NAT})) [0] = [1])$
assert map_4 : $(\text{map} (\text{fun } x \rightarrow x + (1 : \text{NAT})) [0; 1; 2; 3] = [1; 2; 3; 4])$

```
(* ----------------------- *)
(* Reverse                 *)
(* ----------------------- *)

(* First lets define the function [reverse_append], which is
   closely related to reverse. [reverse_append l1 l2] appends the list [l2] to the reverse
of [l1].
   This can be implemented more efficienctly than appending and is
   used to implement reverse. *)
```

val *reverseAppend* : $\forall\ \alpha.\ \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$ `(* originally named rev_append *)`
let rec *reverseAppend* $l_1\ l_2$ = match l_1 with
| $[] \rightarrow l_2$
| $x :: xs \rightarrow \text{reverseAppend}\ xs\ (x :: l_2)$
end
declare termination_argument reverseAppend = automatic

declare *hol* target_rep function reverseAppend = 'REV'
declare *ocaml* target_rep function reverseAppend = 'List.rev_append'

assert $reverseAppend_1$: $(\text{reverseAppend}\ [(0 : \text{NAT}); 1; 2; 3]\ [4; 5] = [3; 2; 1; 0; 4; 5])$

`(* Reversing a list *)`
val *reverse* : $\forall\ \alpha.\ \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$ `(* originally named rev *)`
let *reverse* l = reverseAppend l []

declare *hol* target_rep function reverse = 'REVERSE'
declare *ocaml* target_rep function reverse = 'List.rev'
declare *isabelle* target_rep function reverse = 'List.rev'
declare *coq* target_rep function reverse = 'List.rev'

assert *reverse_nil* : $(\text{reverse}\ ([] : \text{LIST NAT}) = [])$
assert $reverse_1$: $(\text{reverse}\ [(1 : \text{NAT})] = [1])$
assert $reverse_2$: $(\text{reverse}\ [(1 : \text{NAT}); 2] = [2; 1])$
assert $reverse_5$: $(\text{reverse}\ [(1 : \text{NAT}); 2; 3; 4; 5] = [5; 4; 3; 2; 1])$

lemma *reverseAppend* : $(\forall\ l_1\ l_2.\ \text{reverseAppend}\ l_1\ l_2 = (++)\ (\text{reverse}\ l_1)\ l_2)$
let inline {*isabelle*} *reverseAppend* $l_1\ l_2$ = $((\text{reverse}\ l_1) ++ l_2)$

```
(* ----------------------- *)
(* Reverse Map             *)
(* ----------------------- *)
```

val *reverseMap* : $\forall\ \alpha\ \beta.\ (\alpha \rightarrow \beta) \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \beta$
let inline *reverseMap* $f\ l$ = reverse (map $f\ l$)

declare *ocaml* target_rep function reverseMap = 'List.rev_map'

```
(* ========================================================================== *)
(* Folding                                                                    *)
(* ========================================================================== *)

(* ----------------------- *)
(* fold left               *)
(* ----------------------- *)
```

val *foldl* : ∀ α β. (α → β → α) → α → LIST β → α `(* originally foldl *)`

let rec *foldl* f b l = match l with
  | [] → b
  | x :: xs → foldl f (f b x) xs
end
declare termination_argument foldl = automatic

declare *hol* target_rep function foldl = `'FOLDL'`
declare *ocaml* target_rep function foldl = `'List.fold_left'`
declare *isabelle* target_rep function foldl = `'List.foldl'`
declare *coq* target_rep function foldl f e l = `'List.fold_left'` f l e

assert $foldl_0$: (foldl (+) (0 : NAT) [] = 0)
assert $foldl_1$: (foldl (+) (0 : NAT) [4] = 4)
assert $foldl_4$: (foldl (fun l e → e::l) [] [(1 : NAT); 2; 3; 4] = [4; 3; 2; 1])

```
(* ----------------------- *)
(* fold right              *)
(* ----------------------- *)
```

val *foldr* : ∀ α β. (α → β → β) → β → LIST α → β `(* originally foldr with different argument order *)`
let rec *foldr* f b l = match l with
  | [] → b
  | x :: xs → f x (foldr f b xs)
end
declare termination_argument foldr = automatic

declare *hol* target_rep function foldr = `'FOLDR'`
declare *ocaml* target_rep function foldr f b l = `'List.fold_right'` f l b
declare *isabelle* target_rep function foldr f b l = `'List.foldr'` f l b
declare *coq* target_rep function foldr = `'List.fold_right'`

assert $foldr_0$: (foldr (+) (0 : NAT) [] = 0)
assert $foldr_1$: (foldr (+) 1 [(4 : NAT)] = 5)
assert $foldr_4$: (foldr (fun e l → e::l) [] [(1 : NAT); 2; 3; 4] = [1; 2; 3; 4])

```
(* ----------------------- *)
(* concatenating lists     *)
(* ----------------------- *)
```

val *concat* : ∀ α. LIST (LIST α) → LIST α `(* before also called "flatten" *)`
let *concat* = foldr (++) []

declare *hol* target_rep function concat = `'FLAT'`
declare *ocaml* target_rep function concat = `'List.concat'`
declare *isabelle* target_rep function concat = `'List.concat'`
declare *coq* target_rep function concat = `'List.flat_map'` (fun $x \to x$)

assert *concat_nil* : $(\text{concat } ([] : \text{LIST } (\text{LIST NAT})) = [])$
assert $concat_1$: $(\text{concat } [[(1 : \text{NAT})]] = [1])$
assert $concat_2$: $(\text{concat } [[(1 : \text{NAT})]; [2]] = [1; 2])$
assert $concat_3$: $(\text{concat } [[(1 : \text{NAT})]; []; [2]] = [1; 2])$

lemma *concat_emp_thm* : $(\text{concat } [] = [])$
lemma *concat_cons_thm* : $(\forall\ l\ ll.\ (\text{concat } (l{::}ll) = (+\!\!+)\ l\ (\text{concat } ll)))$

```
(* -------------------------- *)
(* concatenating with mapping *)
(* -------------------------- *)
```

val *concatMap* : $\forall\ \alpha\ \beta.\ (\alpha \to \text{LIST } \beta) \to \text{LIST } \alpha \to \text{LIST } \beta$
let inline *concatMap* $f\ l$ = concat (map $f\ l$)

assert *concatMap_nil* : (concatMap (fun $(x : \text{NAT}) \to [x; x]$) $[] = []$)
assert $concatMap_1$: (concatMap (fun $x \to [x; x]$) $[(1 : \text{NAT})] = [1; 1]$)
assert $concatMap_2$: (concatMap (fun $x \to [x; x]$) $[(1 : \text{NAT}); 2] = [1; 1; 2; 2]$)
assert $concatMap_3$: (concatMap (fun $x \to [x; x]$) $[(1 : \text{NAT}); 2; 3] = [1; 1; 2; 2; 3; 3]$)
lemma *concatMap_concat* : ($\forall\ ll.$ concat ll = concatMap (fun $l \to l$) ll)
lemma *concatMap_alt_def* : ($\forall\ f\ l.$ concatMap $f\ l$ = foldr (fun $l\ ll \to f\ l +\!\!+ ll$) $[]\ l$)

```
(* ------------------------- *)
(* universal qualification   *)
(* ------------------------- *)
```

val *all* : $\forall\ \alpha.\ (\alpha \to \mathbb{B}) \to \text{LIST } \alpha \to \mathbb{B}$ `(* originally for_all *)`
let *all* $P\ l$ = foldl (fun $r\ e \to P\ e \wedge r$) true l

declare *hol* target_rep function all = `'EVERY'`
declare *ocaml* target_rep function all = `'List.for_all'`
declare *isabelle* target_rep function all $P\ l$ = ($\forall\ x \in$ (`'set'` l). $P\ x$)
declare *coq* target_rep function all = `'List.forallb'`

assert all_0 : (all (fun $x \to x > (2 : \text{NAT})$) $[]$)
assert all_4 : (all (fun $x \to x > (2 : \text{NAT})$) $[4; 5; 6; 7]$)
assert *all_4_neg* : ($\neg$ (all (fun $x \to x > (2 : \text{NAT})$) $[4; 5; 2; 7]$))

lemma *all_nil_thm* : ($\forall\ P.$ all $P\ []$)
lemma *all_cons_thm* : ($\forall\ P\ e\ l.$ all $P\ (e{::}l) = (P\ e \wedge \text{all } P\ l)$)

```
(* ------------------------- *)
(* existential qualification *)
(* ------------------------- *)
```

val *any* : $\forall\ \alpha.\ (\alpha \to \mathbb{B}) \to \text{LIST } \alpha \to \mathbb{B}$ `(* originally exist *)`
let *any* $P\ l$ = foldl (fun $r\ e \to P\ e \vee r$) false l

declare *hol* target_rep function any = `'EXISTS'`

declare *ocaml* target_rep function any = `'List.exists'`
declare *isabelle* target_rep function any $P\ l\ =\ (\exists\, x \in (\texttt{'set'}\ l).\ P\ x)$
declare *coq* target_rep function any = `'List.existsb'`

assert any_0 : $(\neg\ (\text{any}\ (\textsf{fun}\ x\ \rightarrow\ (x < (3 : \text{NAT})))\ []))$
assert any_4 : $(\neg\ (\text{any}\ (\textsf{fun}\ x\ \rightarrow\ (x < (3 : \text{NAT})))\ [4; 5; 6; 7]))$
assert any_4_neg : $(\text{any}\ (\textsf{fun}\ x\ \rightarrow\ (x < (3 : \text{NAT})))\ [4; 5; 2; 7])$

lemma any_nil_thm : $(\forall\ P.\ \neg\ (\text{any}\ P\ []))$
lemma any_cons_thm : $(\forall\ P\ e\ l.\ \text{any}\ P\ (e{::}l) = (P\ e \lor \text{any}\ P\ l))$

```
(* ========================================================================== *)
(* Indexing lists                                                             *)
(* ========================================================================== *)

(* ------------------------- *)
(* index / nth with maybe   *)
(* ------------------------- *)
```

val *index* : $\forall\ \alpha.\ \text{LIST}\ \alpha\ \rightarrow\ \text{NAT}\ \rightarrow\ \text{MAYBE}\ \alpha$

let rec *index* $l\ n$ = match l with
  | $[]\ \rightarrow$ Nothing
  | $x\ ::\ xs\ \rightarrow$ if $n = 0$ then Just x else index $xs\ (n{-}1)$
end

declare termination_argument index = automatic

declare *isabelle* target_rep function index = `'index'`
declare $\{ocaml;\ hol\}$ rename function index = list_index

assert $index_0$: (index $[(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 0 =$ Just 0)
assert $index_1$: (index $[(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 1 =$ Just 1)
assert $index_2$: (index $[(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 2 =$ Just 2)
assert $index_3$: (index $[(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 3 =$ Just 3)
assert $index_4$: (index $[(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 4 =$ Just 4)
assert $index_5$: (index $[(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 5 =$ Just 5)
assert $index_6$: (index $[(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 6 =$ Nothing)

lemma $index\_is\_none$: $(\forall\ l\ n.\ (\text{index}\ l\ n = \text{Nothing}) \longleftrightarrow (n \geq \text{length}\ l))$
lemma $index\_list\_eq$: $(\forall\ l_1\ l_2.\ ((\forall\ n.\ \text{index}\ l_1\ n = \text{index}\ l_2\ n) \longleftrightarrow (l_1 = l_2)))$

```
(* ------------------------- *)
(* findIndices               *)
(* ------------------------- *)

(* [findIndices P l] returns the indices of all elements of list [l] that satisfy predicate
[P].
   Counting starts with 0, the result list is sorted ascendingly *)
```
val *findIndices* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST NAT}$

let rec *findIndices_aux* $(i : \text{NAT})\ P\ l$ =
  match l with
    | $[]\ \rightarrow\ []$
    | $x\ ::\ xs\ \rightarrow$ if $P\ x$ then i :: findIndices_aux $(i + 1)\ P\ xs$ else findIndices_aux $(i + 1)\ P\ xs$
  end
let *findIndices* $P\ l$ = findIndices_aux $0\ P\ l$

declare termination_argument findIndices_aux = automatic

declare *isabelle* target_rep function findIndices = `'find_indices'`
declare {*ocaml*; *hol*} rename function findIndices = find_indices
declare {*ocaml*; *hol*} rename function findIndices_aux = find_indices_aux

assert $findIndices_1$: (findIndices (fun (n : NAT) → $n > 3$) [] = [])
assert $findIndices_2$: (findIndices (fun (n : NAT) → $n > 3$) [4] = [0])
assert $findIndices_3$: (findIndices (fun (n : NAT) → $n > 3$) [1; 5; 3; 1; 2; 6] = [1; 5])

```
(* ------------------------ *)
(* findIndex                *)
(* ------------------------ *)

(* findIndex returns the first index of a list that satisfies a given predicate. *)
```

val *findIndex* : ∀ α. (α → 𝔹) → LIST α → MAYBE NAT
let *findIndex* P l = match findIndices P l with
 | [] → Nothing
 | x :: _ → Just x
end

declare *isabelle* target_rep function findIndex = `'find_index'`
declare {*ocaml*; *hol*} rename function findIndex = find_index

assert $find\_index_0$: (findIndex (fun (n : NAT) → $n > 3$) [1; 2] = Nothing)
assert $find\_index_1$: (findIndex (fun (n : NAT) → $n > 3$) [1; 2; 4] = Just 2)
assert $find\_index_2$: (findIndex (fun (n : NAT) → $n > 3$) [1; 2; 4; 5; 67; 1] = Just 2)

```
(* ------------------------ *)
(* elemIndices              *)
(* ------------------------ *)
```

val *elemIndices* : ∀ α. *Eq* α ⇒ α → LIST α → LIST NAT
let inline *elemIndices* e l = findIndices ((=) e) l

assert $elemIndices_0$: (elemIndices (2 : NAT) [] = [])
assert $elemIndices_1$: (elemIndices (2 : NAT) [2] = [0])
assert $elemIndices_2$: (elemIndices (2 : NAT) [2; 3; 4; 2; 4; 2] = [0; 3; 5])

```
(* ------------------------ *)
(* elemIndex                *)
(* ------------------------ *)
```

val *elemIndex* : ∀ α. *Eq* α ⇒ α → LIST α → MAYBE NAT
let inline *elemIndex* e l = findIndex ((=) e) l

assert $elemIndex_0$: (elemIndex (2 : NAT) [] = Nothing)
assert $elemIndex_1$: (elemIndex (2 : NAT) [2] = Just 0)
assert $elemIndex_2$: (elemIndex (2 : NAT) [3; 4; 2; 4; 2] = Just 2)

```
(* ========================================================================== *)
(* Creating lists                                                             *)
(* ========================================================================== *)

(* ------------------------ *)
```

```
(* genlist                    *)
(* ------------------------- *)

(* [genlist f n] generates the list [f 0; f 1; ... (f (n-1))] *)
```

val *genlist* : $\forall\, \alpha.\ (\text{NAT} \ \rightarrow\ \alpha) \ \rightarrow\ \text{NAT} \ \rightarrow\ \text{LIST}\ \alpha$

let rec *genlist* $f\ n$ =
  match n with
    $|\ 0\ \rightarrow\ []$
    $|\ n'\ +\ 1\ \rightarrow\ \text{snoc}\ (f\ n')\ (\text{genlist}\ f\ n')$
  end
declare termination_argument genlist = automatic

assert $genlist_0$: (genlist (fun n → n) $0 = []$)
assert $genlist_1$: (genlist (fun n → n) $1 = [0]$)
assert $genlist_2$: (genlist (fun n → n) $2 = [0; 1]$)
assert $genlist_3$: (genlist (fun n → n) $3 = [0; 1; 2]$)
lemma *genlist_length* : $(\forall\ f\ n.\ (\text{length}\ (\text{genlist}\ f\ n) = n))$
lemma *genlist_index* : $(\forall\ f\ n\ i.\ i < n \longrightarrow \text{index}\ (\text{genlist}\ f\ n)\ i = \text{Just}\ (f\ i))$

declare *hol* target_rep function genlist = `'GENLIST'`
declare *isabelle* target_rep function genlist = `'genlist'`

```
(* ------------------------- *)
(* replicate                  *)
(* ------------------------- *)
```

val *replicate* : $\forall\, \alpha.\ \text{NAT}\ \rightarrow\ \alpha\ \rightarrow\ \text{LIST}\ \alpha$
let rec *replicate* $n\ x$ =
  match n with
    $|\ 0\ \rightarrow\ []$
    $|\ n'\ +\ 1\ \rightarrow\ x :: \text{replicate}\ n'\ x$
  end
declare termination_argument replicate = automatic

declare *isabelle* target_rep function replicate = `'List.replicate'`
declare *hol* target_rep function replicate = `'REPLICATE'`

assert $replicate_0$: (replicate 0 $(2 : \text{NAT}) = []$)
assert $replicate_1$: (replicate 1 $(2 : \text{NAT}) = [2]$)
assert $replicate_2$: (replicate 2 $(2 : \text{NAT}) = [2; 2]$)
assert $replicate_3$: (replicate 3 $(2 : \text{NAT}) = [2; 2; 2]$)
lemma *replicate_length* : $(\forall\ n\ x.\ (\text{length}\ (\text{replicate}\ n\ x) = n))$
lemma *replicate_index* : $(\forall\ n\ x\ i.\ i < n \longrightarrow \text{index}\ (\text{replicate}\ n\ x)\ i = \text{Just}\ x)$

```
(* ========================================================================== *)
(* Sublists                                                                   *)
(* ========================================================================== *)

(* ------------------------- *)
(* splitAt                    *)
(* ------------------------- *)

(* [splitAt n xs] returns a tuple (xs1, xs2), with "append xs1 xs2 = xs" and
```

```
   "length xs1 = n". If there are not enough elements
   in [xs], the original list and the empty one are returned. *)
val splitAt : ∀ α. NAT → LIST α → (LIST α * LIST α)
let rec splitAt n l =
  match l with
    | [] → ([], [])
    | x :: xs →
      if n ≤ 0 then ([], l) else
      begin
        let (l1, l2) = splitAt (n−1) xs in
        (x::l1, l2)
      end
  end
declare termination_argument splitAt = automatic

declare isabelle target_rep function splitAt = 'split_at'
declare {ocaml; hol} rename function splitAt = split_at


assert splitAt1 : (splitAt 0 [(1 : NAT); 2; 3; 4; 5; 6] = ([], [1; 2; 3; 4; 5; 6]))
assert splitAt2 : (splitAt 2 [(1 : NAT); 2; 3; 4; 5; 6] = ([1; 2], [3; 4; 5; 6]))
assert splitAt3 : (splitAt 100 [(1 : NAT); 2; 3; 4; 5; 6] = ([1; 2; 3; 4; 5; 6], []))

lemma splitAt_append : (∀ n xs.
  let (xs1, xs2) = splitAt n xs in
  (xs = xs1 ++ xs2))

lemma splitAt_length : (∀ n xs.
  let (xs1, xs2) = splitAt n xs in
  ((length xs1 = n) ∨
   ((length xs1 = length xs) ∧ null xs2)))


(* ------------------------- *)
(* take                      *)
(* ------------------------- *)

(* take n xs returns the prefix of xs of length n, or xs itself if n > length xs *)
val take : ∀ α. NAT → LIST α → LIST α
let take n l = fst (splitAt n l)

declare hol target_rep function take = 'TAKE'
declare isabelle target_rep function take = 'List.take'

assert take1 : (take 0 [(1 : NAT); 2; 3; 4; 5; 6] = [])
assert take2 : (take 2 [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2])
assert take3 : (take 100 [(1 : NAT); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])




(* ------------------------- *)
(* drop                      *)
(* ------------------------- *)

(* [drop n xs] drops the first [n] elements of [xs]. It returns the empty list, if [n] > [length
xs]. *)
val drop : ∀ α. NAT → LIST α → LIST α
let drop n l = snd (splitAt n l)
```

declare *hol* target_rep function drop = 'DROP'
declare *isabelle* target_rep function drop = 'List.drop'

assert $drop_1$: (drop 0 $[(1 : \text{NAT}); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6]$)
assert $drop_2$: (drop 2 $[(1 : \text{NAT}); 2; 3; 4; 5; 6] = [3; 4; 5; 6]$)
assert $drop_3$: (drop 100 $[(1 : \text{NAT}); 2; 3; 4; 5; 6] = []$)

lemma *splitAt_take_drop* : ($\forall$ n xs. splitAt n xs = (take n xs, drop n xs))

let inline {*hol*} *splitAt* n xs = (take n xs, drop n xs)

```
(* ------------------------- *)
(* update                    *)
(* ------------------------- *)
```

val *update* : $\forall\, \alpha$. LIST α $\rightarrow$ NAT $\rightarrow$ α $\rightarrow$ LIST α
let rec *update* l n e =
  match l with
    | [] $\rightarrow$ []
    | x :: xs $\rightarrow$ if $n = 0$ then e :: xs else x :: (update xs $(n-1)$ e)
end
declare termination_argument update = automatic

declare *isabelle* target_rep function update = 'List.list_update'
declare *hol* target_rep function update l n e = 'LUPDATE' e n l
declare {*ocaml*} rename function update = list_update

assert $list\_update_1$: (update [] 2 $(3 : \text{NAT}) = []$)
assert $list\_update_2$: (update $[1; 2; 3; 4; 5]$ 0 $(0 : \text{NAT}) = [0; 2; 3; 4; 5]$)
assert $list\_update_3$: (update $[1; 2; 3; 4; 5]$ 1 $(0 : \text{NAT}) = [1; 0; 3; 4; 5]$)
assert $list\_update_4$: (update $[1; 2; 3; 4; 5]$ 2 $(0 : \text{NAT}) = [1; 2; 0; 4; 5]$)
assert $list\_update_5$: (update $[1; 2; 3; 4; 5]$ 5 $(0 : \text{NAT}) = [1; 2; 3; 4; 5]$)

lemma *list_update_length* : ($\forall$ l n e. length (update l n e) = length l)
lemma *list_update_index* : ($\forall$ i l n e.
  (index (update l n e) i = ((if $i = n \wedge n <$ length l then Just e else index l e))))

```
(* ========================================================================== *)
(* Searching lists                                                            *)
(* ========================================================================== *)

(* ------------------------- *)
(* Membership test           *)
(* ------------------------- *)

(* The membership test, one of the basic list functions, is actually tricky for
   Lem, because it is tricky, which equality to use. From Lem‘s point of
   perspective, we want to use the equality provided by the equality type - class.
   This allows for example to check whether a set is in a list of sets.

   However, in order to use the equality type class, elem essentially becomes
   existential quantification over lists. For types, which implement semantic
   equality (=) with syntactic equality, this is overly complicated. In
   our theorem prover backend, we would end up with overly complicated, harder
   to read definitions and some of the automation would be harder to apply.
```

```
   Moreover, nearly all the old Lem generated code would change and require
   (hopefully minor) adaptions of proofs.

   For now, we ignore this problem and just demand, that all instances of
   the equality type class do the right thing for the theorem prover backends.
*)
```

val $elem$: $\forall\ \alpha.\ Eq\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \mathbb{B}$
val $elemBy$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \mathbb{B}$

let $elemBy\ eq\ e\ l\ =\ \text{any}\ (eq\ e)\ l$
let $elem\ =\ \text{elemBy}\ (=)$

declare hol target_rep function $\text{elem}\ =$ `'MEM'`
declare $ocaml$ target_rep function $\text{elem}\ =$ `'List.mem'`
declare $isabelle$ target_rep function $\text{elem}\ e\ l\ =$ `'Set.member'` e (`'set'` l)

assert $elem_1$: $(\text{elem}\ (2 : \text{NAT})\ [3; 1; 2; 4])$
assert $elem_2$: $(\text{elem}\ (3 : \text{NAT})\ [3; 1; 2; 4])$
assert $elem_3$: $(\text{elem}\ (4 : \text{NAT})\ [3; 1; 2; 4])$
assert $elem_4$: $(\neg\ (\text{elem}\ (5 : \text{NAT})\ [3; 1; 2; 4]))$

lemma $elem\_spec$: $((\forall\ e.\ \neg\ (\text{elem}\ e\ []))\ \wedge$
$(\forall\ e\ x\ xs.\ (\text{elem}\ e\ (x :: xs)) = ((e = x) \vee (\text{elem}\ e\ xs))))$

```
(* ------------------------- *)
(* Find                      *)
(* ------------------------- *)
```

val $find$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{MAYBE}\ \alpha$ `(* previously not of maybe type *)`
let rec $find\ P\ l\ =$ match l with
  | $[]\ \rightarrow\ \text{Nothing}$
  | $x\ ::\ xs\ \rightarrow$ if $P\ x$ then $\text{Just}\ x$ else $\text{find}\ P\ xs$
end
declare termination_argument $\text{find}\ =$ automatic

declare $isabelle$ target_rep function $\text{find}\ =$ `'List.find'`
declare $\{ocaml;\ hol\}$ rename function $\text{find}\ =\ \text{list\_find\_opt}$

assert $find_1$: $((\text{find}\ (\text{fun}\ n\ \rightarrow\ n > (3 : \text{NAT}))\ []) = \text{Nothing})$
assert $find_2$: $((\text{find}\ (\text{fun}\ n\ \rightarrow\ n > (3 : \text{NAT}))\ [2; 1; 3]) = \text{Nothing})$
assert $find_3$: $((\text{find}\ (\text{fun}\ n\ \rightarrow\ n > (3 : \text{NAT}))\ [2; 1; 5; 4]) = \text{Just}\ 5)$
assert $find_4$: $((\text{find}\ (\text{fun}\ n\ \rightarrow\ n > (3 : \text{NAT}))\ [2; 1; 4; 5; 4]) = \text{Just}\ 4)$

lemma $find\_in$: $(\forall\ P\ l\ x.\ (\text{find}\ P\ l = \text{Just}\ x) \longrightarrow P\ x \wedge \text{elem}\ x\ l)$
lemma $find\_not\_in$: $(\forall\ P\ l.\ (\text{find}\ P\ l = \text{Nothing}) = (\neg\ (\text{any}\ P\ l)))$

```
(* ----------------------------- *)
(* Lookup in an associative list *)
(* ----------------------------- *)
```

val $lookup$: $\forall\ \alpha\ \beta.\ Eq\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \text{LIST}\ (\alpha\ *\ \beta)\ \rightarrow\ \text{MAYBE}\ \beta$
val $lookupBy$: $\forall\ \alpha\ \beta.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \alpha\ \rightarrow\ \text{LIST}\ (\alpha\ *\ \beta)\ \rightarrow\ \text{MAYBE}\ \beta$

`(* DPM: eta-expansion for Coq backend type-inference. *)`
let $lookupBy\ eq\ k\ m\ =\ \text{Maybe.map}\ (\text{fun}\ x\ \rightarrow\ \text{snd}\ x)\ (\text{find}\ (\text{fun}\ (k',\ \_)\ \rightarrow\ eq\ k\ k')\ m)$
let inline $lookup\ =\ \text{lookupBy}\ (=)$

declare $isabelle$ target_rep function $\text{lookup}\ x\ l\ =$ `'Map.map_of'` $l\ x$

```
declare {ocaml; hol} rename function lookup = list_assoc_opt

assert lookup_1 : (lookup (3 : NAT) ([(4, (5 : NAT)); (3, 4); (1, 2); (3, 5)]) = Just 4)
assert lookup_2 : (lookup (8 : NAT) ([(4, (5 : NAT)); (3, 4); (1, 2); (3, 5)]) = Nothing)
assert lookup_3 : (lookup (1 : NAT) ([(4, (5 : NAT)); (3, 4); (1, 2); (3, 5)]) = Just 2)

(* ------------------------- *)
(* filter                    *)
(* ------------------------- *)
val filter : ∀ α. (α → 𝔹) → LIST α → LIST α
let rec filter P l = match l with
                   | [] → []
                   | x :: xs → if (P x) then x :: (filter P xs) else filter P xs
                 end
declare termination_argument filter = automatic

declare hol target_rep function filter = 'FILTER'
declare ocaml target_rep function filter = 'List.filter'
declare isabelle target_rep function filter = 'List.filter'
declare coq target_rep function filter = 'List.filter'

assert filter_0 : (filter (fun x → x > (4 : NAT)) [] = [])
assert filter_1 : (filter (fun x → x > (4 : NAT)) [1; 2; 4; 5; 2; 7; 6] = [5; 7; 6])
lemma filter_nil_thm : (∀ P. filter P [] = [])
lemma filter_cons_thm : (∀ P x xs. filter P (x::xs) = (let l' = filter P xs in (if (P x) then x :: l' else l')))


(* ------------------------- *)
(* partition                 *)
(* ------------------------- *)
val partition : ∀ α. (α → 𝔹) → LIST α → LIST α * LIST α
let partition P l = (filter P l, filter (fun x → ¬ (P x)) l)

val reversePartition : ∀ α. (α → 𝔹) → LIST α → LIST α * LIST α
let reversePartition P l = partition P (reverse l)

let inline {hol} partition P l = reversePartition P (reverse l)
declare hol target_rep function reversePartition = 'PARTITION'
declare ocaml target_rep function partition = 'List.partition'
declare isabelle target_rep function partition = 'List.partition'

assert partition_0 : (partition (fun x → x > (4 : NAT)) [] = ([], []))
assert partition_1 : (partition (fun x → x > (4 : NAT)) [1; 2; 4; 5; 2; 7; 6] = ([5; 7; 6], [1; 2; 4; 2]))
lemma partition_fst : (∀ P l. fst (partition P l) = filter P l)
lemma partition_snd : (∀ P l. snd (partition P l) = filter (fun x → ¬ (P x)) l)


(* ------------------------- *)
(* delete first element      *)
(* with certain property     *)
(* ------------------------- *)

val deleteFirst : ∀ α. (α → 𝔹) → LIST α → MAYBE (LIST α)
let rec deleteFirst P l = match l with
                        | [] → Nothing
                      | x :: xs → if (P x) then Just xs else Maybe.map (fun xs' → x :: xs') (deleteFirst P xs)
                        end
declare termination_argument deleteFirst = automatic
```

```
declare isabelle target_rep function deleteFirst = 'delete_first'
declare {ocaml; hol} rename function deleteFirst = list_delete_first

assert deleteFirst1 : (deleteFirst (fun x -> x > (5 : NAT)) [3; 6; 7; 1] = Just [3; 7; 1])
assert deleteFirst2 : (deleteFirst (fun x -> x > (15 : NAT)) [3; 6; 7; 1] = Nothing)
assert deleteFirst3 : (deleteFirst (fun x -> x > (2 : NAT)) [3; 6; 7; 1] = Just [6; 7; 1])


val delete : ∀ α. Eq α => α -> LIST α -> LIST α
val deleteBy : ∀ α. (α -> α -> 𝔹) -> α -> LIST α -> LIST α

let deleteBy eq x l = fromMaybe l (deleteFirst (eq x) l)
let inline delete = deleteBy (=)

declare isabelle target_rep function delete = 'remove'1
declare {ocaml; hol} rename function delete = list_remove1
declare {ocaml; hol} rename function deleteBy = list_delete

assert delete1 : (delete (6 : NAT) [(3 : NAT); 6; 7; 1] = [3; 7; 1])
assert delete2 : (delete (4 : NAT) [(3 : NAT); 6; 7; 1] = [3; 6; 7; 1])
assert delete3 : (delete (3 : NAT) [(3 : NAT); 6; 7; 1] = [6; 7; 1])
assert delete4 : (delete (3 : NAT) [(3 : NAT); 3; 6; 7; 1] = [3; 6; 7; 1])


(* ========================================================================== *)
(* Zipping and unzipping lists                                                *)
(* ========================================================================== *)

(* ------------------------- *)
(* zip                       *)
(* ------------------------- *)

(* zip takes two lists and returns a list of corresponding pairs. If one input list is short,
excess elements of the longer list are discarded. *)
val zip : ∀ α β. LIST α -> LIST β -> LIST (α * β) (* before combine *)
let rec zip l1 l2 = match (l1, l2) with
  | (x :: xs, y :: ys) -> (x, y) :: zip xs ys
  | _ -> []
end
declare termination_argument zip = automatic

declare isabelle target_rep function zip = 'List.zip'
declare {ocaml; hol} rename function zip = list_combine

assert zip1 : (zip [(1 : NAT); 2; 3; 4; 5] [(2 : NAT); 3; 4; 5; 6] = [(1, 2); (2, 3); (3, 4); (4, 5); (5, 6)])

(* this test rules out List.combine for ocaml and ZIP for HOL, but it's needed to make it a
total function *)
assert zip2 : (zip [(1 : NAT); 2; 3] [(2 : NAT); 3; 4; 5; 6] = [(1, 2); (2, 3); (3, 4)])

(* ------------------------- *)
(* unzip                     *)
(* ------------------------- *)

val unzip : ∀ α β. LIST (α * β) -> (LIST α * LIST β)
let rec unzip l = match l with
  | [] -> ([], [])
```

| $(x,\ y)$ :: xys $\rightarrow$ let $(xs,\ ys)$ = unzip xys in $(x :: xs,\ y :: ys)$
end
declare termination_argument unzip = automatic

declare *hol* target_rep function unzip = 'UNZIP'
declare *isabelle* target_rep function unzip = 'list_unzip'
declare *ocaml* target_rep function unzip = 'List.split'

assert $unzip_1$: $(\text{unzip } ([] : \text{LIST } (\text{NAT} * \text{NAT})) = ([],\ []))$
assert $unzip_2$: $(\text{unzip } [((1 : \text{NAT}), (2 : \text{NAT})); (2, 3); (3, 4)] = ([1; 2; 3],\ [2; 3; 4]))$

```
(* ========================================================================== *)
(* Comments (not clean yet, please ignore the rest of the file)               *)
(* ========================================================================== *)

(* ----------------------- *)
(* skipped from Haskell Lib*)
(* -----------------------

intersperse :: a -> [a] -> [a]
intercalate :: [a] -> [[a]] -> [a]
transpose :: [[a]] -> [[a]]
subsequences :: [a] -> [[a]]
permutations :: [a] -> [[a]]
foldl` :: (a -> b -> a) -> a -> [b] -> aSource
foldl1` :: (a -> a -> a) -> [a] -> aSource

and
or
sum
product
maximum
minimum
scanl
scanr
scanl1
scanr1
Accumulating maps

mapAccumL :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])Source
mapAccumR :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])Source

iterate :: (a -> a) -> a -> [a]
repeat :: a -> [a]
cycle :: [a] -> [a]
unfoldr


takeWhile :: (a -> Bool) -> [a] -> [a]Source
dropWhile :: (a -> Bool) -> [a] -> [a]Source
dropWhileEnd :: (a -> Bool) -> [a] -> [a]Source
span :: (a -> Bool) -> [a] -> ([a], [a])Source
break :: (a -> Bool) -> [a] -> ([a], [a])Source
break p is equivalent to span (not . p).
stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]Source
group :: Eq a => [a] -> [[a]]Source
inits :: [a] -> [[a]]Source
tails :: [a] -> [[a]]Source
```

```
isPrefixOf :: Eq a => [a] -> [a] -> BoolSource
isSuffixOf :: Eq a => [a] -> [a] -> BoolSource
isInfixOf :: Eq a => [a] -> [a] -> BoolSource
```

```
notElem :: Eq a => a -> [a] -> BoolSource
```

```
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]Source
zip4 :: [a] -> [b] -> [c] -> [d] -> [(a, b, c, d)]Source
zip5 :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a, b, c, d, e)]Source
zip6 :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [(a, b, c, d, e, f)]Source
zip7 :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [(a, b, c, d, e, f, g)]Source
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]Source
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]Source
zipWith4 :: (a -> b -> c -> d -> e) -> [a] -> [b] -> [c] -> [d] -> [e]Source
zipWith5 :: (a -> b -> c -> d -> e -> f) -> [a] -> [b] -> [c] -> [d] -> [e] -> [f]Source
zipWith6 :: (a -> b -> c -> d -> e -> f -> g) -> [a] -> [b] -> [c] -> [d] -> [e] -> [f]
-> [g]Source
zipWith7 :: (a -> b -> c -> d -> e -> f -> g -> h) -> [a] -> [b] -> [c] -> [d] -> [e]
-> [f] -> [g] -> [h]Source
```

```
unzip3 :: [(a, b, c)] -> ([a], [b], [c])Source
unzip4 :: [(a, b, c, d)] -> ([a], [b], [c], [d])Source
unzip5 :: [(a, b, c, d, e)] -> ([a], [b], [c], [d], [e])Source
unzip6 :: [(a, b, c, d, e, f)] -> ([a], [b], [c], [d], [e], [f])Source
unzip7 :: [(a, b, c, d, e, f, g)] -> ([a], [b], [c], [d], [e], [f], [g])Source
```

```
lines :: String -> [String]Source
words :: String -> [String]Source
unlines :: [String] -> StringSource
unwords :: [String] -> StringSource
nub :: Eq a => [a] -> [a]Source
delete :: Eq a => a -> [a] -> [a]Source
```

```
(\\) :: Eq a => [a] -> [a] -> [a]Source
union :: Eq a => [a] -> [a] -> [a]Source
intersect :: Eq a => [a] -> [a] -> [a]Source
sort :: Ord a => [a] -> [a]Source
insert :: Ord a => a -> [a] -> [a]Source
```

```
nubBy :: (a -> a -> Bool) -> [a] -> [a]Source
deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]Source
deleteFirstsBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]Source
unionBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]Source
intersectBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]Source
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]Source
sortBy :: (a -> a -> Ordering) -> [a] -> [a]Source
insertBy :: (a -> a -> Ordering) -> a -> [a] -> [a]Source
maximumBy :: (a -> a -> Ordering) -> [a] -> aSource
minimumBy :: (a -> a -> Ordering) -> [a] -> aSource
genericLength :: Num i => [b] -> iSource
```

```
genericTake :: Integral i => i -> [a] -> [a]Source
genericDrop :: Integral i => i -> [a] -> [a]Source
genericSplitAt :: Integral i => i -> [b] -> ([b], [b])Source
genericIndex :: Integral a => [b] -> a -> bSource
genericReplicate :: Integral i => i -> a -> [a]Source


*)


(* ----------------------- *)
(* skipped from Lem Lib    *)
(* -----------------------


val for_all2 : forall 'a 'b. ('a -> 'b -> bool) -> list 'a -> list 'b -> bool
val exists2 : forall 'a 'b. ('a -> 'b -> bool) -> list 'a -> list 'b -> bool
val map2 : forall 'a 'b 'c. ('a -> 'b -> 'c) -> list 'a -> list 'b -> list 'c
val rev_map2 : forall 'a 'b 'c. ('a -> 'b -> 'c) -> list 'a -> list 'b -> list 'c
val fold_left2 : forall 'a 'b 'c. ('a -> 'b -> 'c -> 'a) -> 'a -> list 'b -> list 'c ->
'a
val fold_right2 : forall 'a 'b 'c. ('a -> 'b -> 'c -> 'c) -> list 'a -> list 'b -> 'c ->
'c


(* now maybe result and called lookup *)
val assoc : forall 'a 'b. 'a -> list ('a * 'b) -> 'b
let inline {ocaml} assoc = Ocaml.List.assoc


val mem_assoc : forall 'a 'b. 'a -> list ('a * 'b) -> bool
val remove_assoc : forall 'a 'b. 'a -> list ('a * 'b) -> list ('a * 'b)



val stable_sort : forall 'a. ('a -> 'a -> num) -> list 'a -> list 'a
val fast_sort : forall 'a. ('a -> 'a -> num) -> list 'a -> list 'a

val merge : forall 'a. ('a -> 'a -> num) -> list 'a -> list 'a -> list 'a
val intersect : forall 'a. list 'a -> list 'a -> list 'a


*)
```

9 List_extra

```
(******************************************************************************)
(* A library for lists - the non-pure part                                    *)
(*                                                                            *)
(* It mainly follows the Haskell List-library                                 *)
(******************************************************************************)
```

```
(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

```
(* rename module to clash with existing list modules of targets
   problem: renaming from inside the module itself! *)
```

declare $\{isabelle;\ hol;\ ocaml\}$ rename module $=$ lem_list_extra

open import *Bool Maybe Basic_classes Tuple Num List*

```
(* ------------------------- *)
(* head of non-empty list    *)
(* ------------------------- *)
```

val *head* : $\forall\ \alpha.\ \textsc{list}\ \alpha\ \rightarrow\ \alpha$
let *head* l $=$ match l with $|\ x :: xs\ \rightarrow\ x$ end

declare compile_message head $=$ "$headisonlydefinedonnon{-}emptylistandshouldthereforebeavoided.Usemachinginsteadand$

declare *hol* target_rep function head $=$ `'HD'`
declare *ocaml* target_rep function head $=$ `'List.hd'`
declare *isabelle* target_rep function head $=$ `'List.hd'`

assert *head_simple*$_1$: $(\text{head } [3; 1] = (3 : \textsc{nat}))$
assert *head_simple*$_2$: $(\text{head } [5; 4] = (5 : \textsc{nat}))$

```
(* ------------------------- *)
(* tail of non-empty list    *)
(* ------------------------- *)
```

val *tail* : $\forall\ \alpha.\ \textsc{list}\ \alpha\ \rightarrow\ \textsc{list}\ \alpha$
let *tail* l $=$ match l with $|\ x :: xs\ \rightarrow\ xs$ end

declare compile_message tail $=$ "$tailisonlydefinedonnon{-}emptylistandshouldthereforebeavoided.Usemachinginsteadandhan$

declare *hol* target_rep function tail $=$ `'TL'`
declare *ocaml* target_rep function tail $=$ `'List.tl'`
declare *isabelle* target_rep function tail $=$ `'List.tl'`

assert *tail_simple*$_1$: $(\text{tail } [(3 : \textsc{nat}); 1] = [1])$
assert *tail_simple*$_2$: $(\text{tail } [(5 : \textsc{nat})] = [])$
assert *tail_simple*$_3$: $(\text{tail } [(5 : \textsc{nat}); 4; 3; 2] = [4; 3; 2])$

lemma *head_tail_cons* : $(\forall\ l.\ \text{length } l > 0 \longrightarrow (l = (\text{head } l)::(\text{tail } l)))$

```
(* ------------------------- *)
(* last                      *)
(* ------------------------- *)
```

val $last$: $\forall\ \alpha.$ LIST $\alpha\ \rightarrow\ \alpha$
let rec $last\ l\ =$ match l with $|\ [x]\ \rightarrow\ x\ |\ x_1 :: x_2 :: xs\ \rightarrow$ last $(x_2 :: xs)$ end
declare compile_message last $=$ $''lastisonlydefinedonnon{-}emptylistandshouldthereforebeavoided.Usemachinginsteadandha$

declare hol target_rep function last $=$ `'LAST'`
declare $isabelle$ target_rep function last $=$ `'List.last'`

assert $last\_simple_1$: $(\text{last }[(3 : \text{NAT}); 1] = 1)$
assert $last\_simple_2$: $(\text{last }[(5 : \text{NAT}); 4] = 4)$

```
(* ------------------------- *)
(* init                      *)
(* ------------------------- *)

(* All elements of a non-empty list except the last one. *)
```

val $init$: $\forall\ \alpha.$ LIST $\alpha\ \rightarrow$ LIST α
let rec $init\ l\ =$ match l with $|\ [x]\ \rightarrow\ []\ |\ x_1 :: x_2 :: xs\ \rightarrow\ x_1{::}(\text{init }(x_2{::}xs))$ end

declare compile_message init $=$ $''initisonlydefinedonnon{-}emptylistandshouldthereforebeavoided.Usemachinginsteadandha$

declare hol target_rep function init $=$ `'FRONT'`
declare $isabelle$ target_rep function init $=$ `'List.butlast'`

assert $init\_simple_1$: $(\text{init }[(3 : \text{NAT}); 1] = [3])$
assert $init\_simple_2$: $(\text{init }[(5 : \text{NAT})] = [])$
assert $init\_simple_3$: $(\text{init }[(5 : \text{NAT}); 4; 3; 2] = [5; 4; 3])$

lemma $init\_last\_append$: $(\forall\ l.\ \text{length }l > 0 \longrightarrow (l = (\text{init }l) +\!\!+ [\text{last }l]))$

```
(* ------------------------- *)
(* foldl1 / foldr1           *)
(* ------------------------- *)

(* folding functions for non-empty lists,
   which don‘t take the base case *)
```

val $foldl_1$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha)\ \rightarrow$ LIST $\alpha\ \rightarrow\ \alpha$
let $foldl_1\ f\ (x\ ::\ xs)\ =$ foldl $f\ x\ xs$
declare compile_message foldl_1 $=$ $''foldl1isonlydefinedonnon{-}emptylists.Betterusefoldlorexplicitpatternmatching.''$

val $foldr_1$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha)\ \rightarrow$ LIST $\alpha\ \rightarrow\ \alpha$
let $foldr_1\ f\ (x\ ::\ xs)\ =$ foldr $f\ x\ xs$
declare compile_message foldr_1 $=$ $''foldr1isonlydefinedonnon{-}emptylists.Betterusefoldrorexplicitpatternmatching.''$

```
(* ------------------------- *)
(* nth element               *)
(* ------------------------- *)
```

```
(* get the nth element of a list *)
```
val nth : $\forall\ \alpha.$ LIST $\alpha \rightarrow$ NAT $\rightarrow \alpha$
let $nth\ l\ n$ = match index $l\ n$ with Just $e \rightarrow e$ end
declare compile_message foldl_1 = $"nth is undefined for too large indices, use carefully"$

declare hol target_rep function nth $l\ n$ = `'EL'` $n\ l$
declare $ocaml$ target_rep function nth = `'List.nth'`
declare $isabelle$ target_rep function nth = `'List.nth'`
declare coq target_rep function nth $l\ n$ = `'List.nth'` $n\ l$

assert nth_0 : (nth [0; 1; 2; 3; 4; 5] 0 = (0 : NAT))
assert nth_1 : (nth [0; 1; 2; 3; 4; 5] 1 = (1 : NAT))
assert nth_2 : (nth [0; 1; 2; 3; 4; 5] 2 = (2 : NAT))
assert nth_3 : (nth [0; 1; 2; 3; 4; 5] 3 = (3 : NAT))
assert nth_4 : (nth [0; 1; 2; 3; 4; 5] 4 = (4 : NAT))
assert nth_5 : (nth [0; 1; 2; 3; 4; 5] 5 = (5 : NAT))

lemma nth_index : $(\forall\ l\ n\ e.\ n < \text{length}\ l \longrightarrow \text{index}\ l\ n = \text{Just}\ (\text{nth}\ l\ n))$

```
(* ------------------------- *)
(* Find_non_pure             *)
(* ------------------------- *)
```
val $find\_non\_pure$: $\forall\ \alpha.\ (\alpha \rightarrow \mathbb{B}) \rightarrow$ LIST $\alpha \rightarrow \alpha$
let $find\_non\_pure\ P\ l$ = match (find $P\ l$) with
| Just $e \rightarrow e$
end

declare compile_message find_non_pure = $"find\_non\_pure is undefined, if no element with the property is in the list. Better use find$

```
(* ------------------------- *)
(* zip same length           *)
(* ------------------------- *)
```

val zip_same_length : $\forall\ \alpha\ \beta.$ LIST $\alpha \rightarrow$ LIST $\beta \rightarrow$ LIST $(\alpha * \beta)$
let inline zip_same_length = List.zip

declare compile_message zip_same_length = $"zip\_same\_length is undefined, if the two lists have different lengths"$

declare hol target_rep function zip_same_length $l_1\ l_2$ = `'ZIP'` $(l_1,\ l_2)$
declare $ocaml$ target_rep function zip_same_length = `'List.combine'`

assert $zip\_same\_length_1$: (zip_same_length [(1 : NAT); 2; 3; 4; 5] [(2 : NAT); 3; 4; 5; 6] = [(1, 2); (2, 3); (3, 4); (4, 5); (5, 6)])

10 Set_helpers

```
(******************************************************************************)
(* Helper functions for sets                                                  *)
(******************************************************************************)

(* Usually there is a something.lem file containing the main definitions and a
   something_extra.lem one containing functions that might cause problems for
   some backends or are just seldomly used.

   For sets the situation is different. folding is not well defined, since it
   is only sensibly defined for finite sets and it the traversel
   order is underspecified. *)

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

open import *Bool Basic_classes Maybe Function Num*
declare {*isabelle*; *hol*; *ocaml*} rename module = lem_set_helpers

open import {*coq*} *Coq.Lists.TheoryList*

```
(* ------------------------ *)
(* fold                     *)
(* ------------------------ *)

(* fold is suspicious, because if given a function, for which
   the order, in which the arguments are given, matters, it's
   results are undefined. On the other hand, it is very handy to
   define other - non suspicious functions.

   Moreover, fold is central for OCaml, size it is used to
   compile set comprehensions *)
```

val *fold* : $\forall\ \alpha\ \beta.\ (\alpha\ \rightarrow\ \beta\ \rightarrow\ \beta)\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \beta\ \rightarrow\ \beta$
declare compile_message fold = ”$foldisnon-deterministicbecausetheorderoftheiterationisunclear.It'sresultmaydifferbetwe$
$levelrepresentationofsetsandbedifferentfortworepresentationsofthesameset.$”

declare *hol* target_rep function fold = `'ITSET'`
declare *isabelle* target_rep function fold $f\ A\ q$ = `'Finite_Set.fold'` $f\ q\ A$
declare *ocaml* target_rep function fold = `'Pset.fold'`
declare *coq* target_rep function fold = `'set_fold'`

11 Set

```
(******************************************************************************)
(* A library for sets                                                         *)
(*                                                                            *)
(* It mainly follows the Haskell Set-library                                  *)
(******************************************************************************)

(* Sets in Lem are a bit tricky. On the one hand, we want efficiently executable sets.
   OCaml and Haskell both represent sets by some kind of balancing trees. This means
   that sets are finite and an order on the elemet type is required.
   Such sets are constructed by simple, executable operations like inserting or
   deleting elements, union, intersection, filtering etc.

   On the other hand, we want to use sets for specifications. This leads often
   infinite sets, which are specificied in complicated, perhaps even undecidable
   ways.

   The set library in this file, chooses the first approach. It describes
   *finite* sets with an underlying order. Infinite sets should in the medium
   run be represented by a separate type. Since this would require some significant
   changes to Lem, for the moment also infinite sets are represented using this
   class. However, a run-time exception might occour when using these sets.
   This problem needs adressing in the future. *)


(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

open import *Bool Basic_classes Maybe Function Num List Set_helpers*

declare {*isabelle*; *hol*; *ocaml*} rename module = lem_set

```
(* DPM: sets currently implemented as lists due to mismatch between Coq type
 * class hierarchy and the hierarchy implemented in Lem.
 *)
```

open import {*coq*} *Coq.Lists.TheoryList*
open import {*hol*} *lemTheory*
open import {*isabelle*} *$LIB_DIR/Lem*

```
(* Type of sets and set comprehensions are hard-coded *)
```

declare *ocaml* target_rep type SET = `'Pset.set'`

```
(* ----------------------- *)
(* Equality check          *)
(* ----------------------- *)
```

val *setEqualBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \text{ORDERING})\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
declare *coq* target_rep function setEqualBy = `'set_equal_by'`

val *setEqual* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let inline {*hol*; *isabelle*} *setEqual* = unsafe_structural_equality
let inline {*coq*} *setEqual* = setEqualBy setElemCompare
declare *ocaml* target_rep function setEqual = `'Pset.equal'`

instance $\forall\ \alpha.\ \mathit{SetType}\ \alpha\ \Rightarrow\ (\mathit{Eq}\ (\text{SET}\ \alpha))$

```
  let = = setEqual
  let <> s1 s2 = ¬ (setEqual s1 s2)
end



(* ----------------------- *)
(* compare                 *)
(* ----------------------- *)

val setCompareBy : ∀ α. (α → α → ORDERING) → SET α → SET α → ORDERING
declare coq target_rep function setCompareBy = 'set_compare_by'

val setCompare : ∀ α. SetType α ⇒ SET α → SET α → ORDERING
let inline {coq} setCompare = setCompareBy setElemCompare
declare ocaml target_rep function setCompare = 'Pset.compare'

instance ∀ α. SetType α ⇒ (SetType (SET α))
  let setElemCompare = setCompare
end



(* ----------------------- *)
(* Empty set               *)
(* ----------------------- *)

val empty : ∀ α. SetType α ⇒ SET α
val emptyBy : ∀ α. (α → α → ORDERING) → SET α

declare ocaml target_rep function emptyBy = 'Pset.empty'
let inline {ocaml} empty = emptyBy setElemCompare

declare coq target_rep function empty = 'set_empty'
declare hol target_rep function empty = 'EMPTY'
declare isabelle target_rep function empty = '{}'
declare html target_rep function empty = '&empty;'
declare tex target_rep function empty = '$\emptyset$'

assert empty0 : (∅ : SET 𝔹) = {}
assert empty1 : (∅ : SET NAT) = {}
assert empty2 : (∅ : SET (LIST NAT)) = {}
assert empty3 : (∅ : SET (SET NAT)) = {}

(* ----------------------- *)
(* any / all               *)
(* ----------------------- *)

val any : ∀ α. SetType α ⇒ (α → 𝔹) → SET α → 𝔹
let inline any P s = (∃ e ∈ s. P e)

declare coq target_rep function any = 'set_any'
declare hol target_rep function any P s = 'EXISTS' P ('SET_TO_LIST' s)
declare isabelle target_rep function any P s = 'Set.Bex' s P
declare ocaml target_rep function any = 'Pset.exists'

assert any0 : any (fun (x : NAT) → x > 5) {3; 4; 6}
assert any1 : ¬ (any (fun (x : NAT) → x > 10) {3; 4; 6})
```

val all : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \mathbb{B}$
let inline $all\ P\ s\ =\ (\forall\ e \in s.\ P\ e)$

declare coq target_rep function all = 'set_for_all'
declare hol target_rep function all $P\ s$ = 'EVERY' P ('SET_TO_LIST' s)
declare $isabelle$ target_rep function all $P\ s$ = 'Set.Ball' $s\ P$
declare $ocaml$ target_rep function all = 'Pset.for_all'

assert all_0 : all (fun (x : NAT) $\rightarrow\ x > 2$) $\{3;\ 4;\ 6\}$
assert all_1 : $\neg$ (all (fun (x : NAT) $\rightarrow\ x > 2$) $\{3;\ 4;\ 6;\ 1\}$)

```
(* ----------------------- *)
(* (IN)                    *)
(* ----------------------- *)
```

val IN [member] : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \mathbb{B}$
val $memberBy$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \textsc{ordering})\ \rightarrow\ \alpha\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \mathbb{B}$

declare coq target_rep function memberBy = 'set_member_by'
let inline $\{coq\}$ $member$ = memberBy setElemCompare
declare $ocaml$ target_rep function member = 'Pset.mem'
declare $isabelle$ target_rep function member = infix '\<in>'
declare hol target_rep function member = infix 'IN'
declare $html$ target_rep function member = infix '∈'
declare tex target_rep function member = infix '$\in$'

assert in_1 : $((1 : \textsc{nat}) \in \{(2 : \textsc{nat});\ 3;\ 1\})$
assert in_2 : $(\neg\ ((1 : \textsc{nat}) \in \{2;\ 3;\ 4\}))$
assert in_3 : $(\neg\ ((1 : \textsc{nat}) \in \{\}))$
assert in_4 : $((1 : \textsc{nat}) \in \{1;\ 2;\ 1;\ 3;\ 1;\ 4\})$

```
(* ----------------------- *)
(* not (IN)                *)
(* ----------------------- *)
```

val NIN [notMember] : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \mathbb{B}$
let inline $notMember\ e\ s\ =\ \neg\ (e \in s)$
declare $html$ target_rep function notMember = infix '∉'
declare $isabelle$ target_rep function notMember = infix '\<notin>'
declare tex target_rep function notMember = infix '$\not\in$'

assert nin_1 : $\neg\ ((1 : \textsc{nat}) \notin \{2;\ 3;\ 1\})$
assert nin_2 : $((1 : \textsc{nat}) \notin \{2;\ 3;\ 4\})$
assert nin_3 : $((1 : \textsc{nat}) \notin \{\})$
assert nin_4 : $\neg\ ((1 : \textsc{nat}) \notin \{1;\ 2;\ 1;\ 3;\ 1;\ 4\})$

```
(* ----------------------- *)
(* insert                  *)
(* ----------------------- *)
```

val $insert$: $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \textsc{set}\ \alpha$ (* before add *)

declare $ocaml$ target_rep function insert = 'Pset.add'
declare coq target_rep function insert = 'set_add'
declare hol target_rep function insert = infix 'INSERT'
declare $isabelle$ target_rep function insert = 'Set.insert'

assert $insert_1$: ((insert (2 : NAT) {3; 4}) = {2; 3; 4})
assert $insert_2$: ((insert (3 : NAT) {3; 4}) = {3; 4})
assert $insert_3$: ((insert (3 : NAT) {}) = {3})

```
(* ----------------------- *)
(* Emptyness check         *)
(* ----------------------- *)
```

val *null* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow$ SET $\alpha \rightarrow \mathbb{B}$ `(* before is_empty *)`
let inline *null* s = $(s = \{\})$

declare *ocaml* target_rep function null = `'Pset.is_empty'`
declare *coq* target_rep function null = `'set_is_empty'`

assert $null_1$: (null ({} : SET NAT))
assert $null_2$: ($\neg$ (null {(1 : NAT)}))

```
(* ------------------------ *)
(* singleton                *)
(* ------------------------ *)
```

val *singleton* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \alpha \rightarrow$ SET α
let inline *singleton* x = $\{x\}$

declare *coq* target_rep function singleton = `'set_singleton'`

assert $singleton_1$: singleton (2 : NAT) = {2}
assert $singleton_2$: $\neg$ (null (singleton (2 : NAT)))
assert $singleton_3$: 2 $\in$ (singleton (2 : NAT))
assert $singleton_4$: 3 $\notin$ (singleton (2 : NAT))

```
(* ----------------------- *)
(* size                    *)
(* ----------------------- *)
```

val *size* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow$ SET $\alpha \rightarrow$ NAT

declare *ocaml* target_rep function size = `'Pset.cardinal'`
declare *coq* target_rep function size = `'set_cardinal'`
declare *hol* target_rep function size = `'CARD'`
declare *isabelle* target_rep function size = `'card'`

assert $size_1$: (size ({} : SET NAT) = 0)
assert $size_2$: (size {(2 : NAT)} = 1)
assert $size_3$: (size {(1 : NAT); 1} = 1)
assert $size_4$: (size {(2 : NAT); 1; 3} = 3)
assert $size_5$: (size {(2 : NAT); 1; 3; 9} = 4)

lemma *null_size* : ($\forall\ s.$ (null s) $\longrightarrow$ (size s = 0))
lemma *null_singleton* : ($\forall\ x.$ (size (singleton x) = 1))

```
(* ----------------------------*)
(* setting up pattern matching *)
```

```
(* -------------------------- *)
```

val set_case : $\forall\ \alpha\ \beta.\ SetType\ \alpha\ \Rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \beta\ \rightarrow\ (\alpha\ \rightarrow\ \beta)\ \rightarrow\ \beta\ \rightarrow\ \beta$

```
(* please provide target bindings, since choose is defined only in extra
   and not the right thing to use here anyhow.

let set_case s c_empty c_sing c_else =
  if (null s) then c_empty else
  if (size s = 1) then c_sing (choose s)
  else c_else
*)
```

declare hol target_rep function $\mathrm{set\_case}$ = `'set_CASE'`
declare $isabelle$ target_rep function $\mathrm{set\_case}$ = `'set_case'`
declare coq target_rep function $\mathrm{set\_case}$ = `'set_case'`
declare $ocaml$ target_rep function $\mathrm{set\_case}$ = `'Pset.set_case'`

declare pattern_match inexhaustive $\textsc{set}\ \alpha\ =\ [\ \mathrm{empty};\ \mathrm{singleton}\ ]\ \mathrm{set\_case}$

assert $set\_patterns_0$: (
  match $(\{\}\ :\ \textsc{set nat})$ with
    | $\emptyset$ $\rightarrow$ true
    | _ $\rightarrow$ false
  end
)

assert $set\_patterns_1$: $\neg$ (
  match $\{(2 : \textsc{nat})\}$ with
    | $\emptyset$ $\rightarrow$ true
    | _ $\rightarrow$ false
  end
)

assert $set\_patterns_2$: $\neg$ (
  match $\{(3\ :\ \textsc{nat});\ 4\}$ with
    | $\emptyset$ $\rightarrow$ true
    | _ $\rightarrow$ false
  end
)

assert $set\_patterns_3$: (
  match $(\{2\}\ :\ \textsc{set nat})$ with
    | $\emptyset\ \rightarrow\ 0$
    | $\mathrm{singleton}\ x\ \rightarrow\ x$
    | _ $\rightarrow\ 1$
  end
) $= 2$

assert $set\_patterns_4$: (
  match $(\{\}\ :\ \textsc{set nat})$ with
    | $\emptyset\ \rightarrow\ 0$
    | $\mathrm{singleton}\ x\ \rightarrow\ x$
    | _ $\rightarrow\ 1$
  end
) $= 0$

assert $set\_patterns_5$: (

```
  match ({3; 4; 5} : SET NAT) with
    | ∅ → 0
    | singleton x → x
    | _ → 1
  end
) = 1

assert set_patterns_6 : (
  match ({3; 3; 3} : SET NAT) with
    | ∅ → 0
    | singleton x → x
    | _ → 1
  end
) = 3

assert set_patterns_7 : (
  match ({3; 4; 5} : SET NAT) with
    | ∅ → 0
    | singleton _ → 1
    | s → size s
  end
) = 3

assert set_patterns_8 : (
  match (({3; 4; 5} : SET NAT), false) with
    | (∅, true) → 0
    | (singleton _, _) → 1
    | (s, true) → size s
    | _ → 5
  end
) = 5

assert set_patterns_9 : (
  match ({5} : SET NAT) with
    | ∅ → 0
    | singleton 2 → 0
    | singleton (x + 3) → x
    | _ → 1
  end
) = 2

assert set_patterns_10 : (
  match ({2} : SET NAT) with
    | ∅ → 0
    | singleton 2 → 0
    | singleton (x + 3) → x
    | _ → 1
  end
) = 0


(* ----------------------- *)
(* filter                  *)
(* ----------------------- *)

val filter : ∀ α. SetType α ⇒ (α → 𝔹) → SET α → SET α
let filter P s = {e | ∀ e ∈ s | P e}
```

declare *ocaml* target_rep function filter $=$ `'Pset.filter'`
declare *isabelle* target_rep function filter $=$ `'set_filter'`
declare *hol* target_rep function filter $=$ `'SET_FILTER'`

assert $filter_1$: (filter (fun $n \rightarrow (n > 2)$) $\{(1 : \text{NAT}); 2; 3; 4\} = \{3; 4\}$)
assert $filter_2$: (filter (fun $n \rightarrow n > (2 : \text{NAT})$) $\{\} = \{\}$)
lemma *filter_emp* : ($\forall$ P. (filter P $\{\}$) $= \{\}$)
lemma *filter_insert* : ($\forall$ e s P. (filter P (insert e s)) $=$
(if (P e) then insert e (filter P s) else (filter P s)))

```
(* ----------------------- *)
(* partition               *)
(* ----------------------- *)
```

val *partition* : $\forall\ \alpha.$ *SetType* $\alpha \Rightarrow (\alpha \rightarrow \mathbb{B}) \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha * \text{SET}\ \alpha$
let *partition* P s $=$ (filter P s, filter (fun $e \rightarrow \neg$ (P e)) s)
declare $\{hol\}$ rename function partition $=$ SET_PARTITION

```
(* ----------------------- *)
(* split                   *)
(* ----------------------- *)
```

val *split* : $\forall\ \alpha.$ *SetType* α, *Ord* $\alpha \Rightarrow \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha * \text{SET}\ \alpha$
let *split* p s $=$ (filter ((<) p) s, filter ((>) p) s)
declare $\{hol\}$ rename function split $=$ SET_SPLIT

val *splitMember* : $\forall\ \alpha.$ *SetType* α, *Ord* $\alpha \Rightarrow \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha * \mathbb{B} * \text{SET}\ \alpha$
let *splitMember* p s $=$ (filter ((<) p) s, $p \in s$, filter ((>) p) s)

```
(* ------------------------ *)
(* subset and proper subset *)
(* ------------------------ *)
```

val *isSubsetOfBy* : $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \text{ORDERING}) \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha \rightarrow \mathbb{B}$
val *isProperSubsetOfBy* : $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \text{ORDERING}) \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha \rightarrow \mathbb{B}$

val *isSubsetOf* : $\forall\ \alpha.$ *SetType* $\alpha \Rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha \rightarrow \mathbb{B}$
val *isProperSubsetOf* : $\forall\ \alpha.$ *SetType* $\alpha \Rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha \rightarrow \mathbb{B}$

declare *ocaml* target_rep function isSubsetOf $=$ `'Pset.subset'`
declare *hol* target_rep function isSubsetOf $=$ infix `'SUBSET'`
declare *isabelle* target_rep function isSubsetOf $=$ infix `'\<subseteq>'`
declare *html* target_rep function isSubsetOf $=$ infix `'⊆'`
declare *tex* target_rep function isSubsetOf $=$ infix `'$\subseteq$'`
declare *coq* target_rep function isSubsetOfBy $=$ `'set_subset_by'`
let inline $\{coq\}$ *isSubsetOf* $=$ isSubsetOfBy setElemCompare

declare *ocaml* target_rep function isProperSubsetOf $=$ `'Pset.subset_proper'`
declare *hol* target_rep function isProperSubsetOf $=$ infix `'PSUBSET'`
declare *isabelle* target_rep function isProperSubsetOf $=$ infix `'\<subset>'`
declare *html* target_rep function isProperSubsetOf $=$ infix `'⊂'`
declare *tex* target_rep function isProperSubsetOf $=$ infix `'$\subset$'`
declare *coq* target_rep function isProperSubsetOfBy $=$ `'set_proper_subset_by'`
let inline $\{coq\}$ *isProperSubsetOf* $=$ isProperSubsetOfBy setElemCompare

let inline $subset$ = $(\subseteq)$
declare tex target_rep function subset = infix '$\subseteq$'

assert $isSubsetOf_1$: $((\{\} : \text{SET NAT}) \subseteq \{\})$
assert $isSubsetOf_2$: $(\{(1 : \text{NAT}); 2; 3\} \subseteq \{1; 2; 3\})$
assert $isSubsetOf_3$: $(\{(1 : \text{NAT}); 2\} \subseteq \{3; 2; 1\})$
lemma $isSubsetOf\_refl$: $(\forall\ s.\ s \subseteq s)$
lemma $isSubsetOf\_def$: $(\forall\ s_1\ s_2.\ s_1 \subseteq s_2 = (\forall\ e.\ e \in s_1 \longrightarrow e \in s_2))$
lemma $isSubsetOf\_eq$: $(\forall\ s_1\ s_2.\ (s_1 = s_2) \longleftrightarrow ((s_1 \subseteq s_2) \wedge (s_2 \subseteq s_1)))$

assert $isProperSubsetOf_1$: $(\neg\ ((\{\} : \text{SET NAT}) \subset \{\}))$
assert $isProperSubsetOf_2$: $(\neg\ (\{(1 : \text{NAT}); 2; 3\} \subset \{1; 2; 3\}))$
assert $isProperSubsetOf_3$: $(\{(1 : \text{NAT}); 2\} \subset \{3; 2; 1\})$
lemma $isProperSubsetOf\_irrefl$: $(\forall\ s.\ \neg\ (s \subset s))$
lemma $isProperSubsetOf\_def$: $(\forall\ s_1\ s_2.\ s_1 \subset s_2 \longleftrightarrow ((s_1 \subseteq s_2) \wedge \neg\ (s_2 \subseteq s_1)))$

```
(* ------------------------ *)
(* delete                   *)
(* ------------------------ *)
```

val $delete$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha \Rightarrow \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha$
val $deleteBy$: $\forall\ \alpha.\ SetType\ \alpha \Rightarrow (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha$

let inline $deleteBy\ eq\ e\ s$ = filter (fun $e_2 \rightarrow \neg\ (eq\ e\ e_2))\ s$
let inline $delete\ e\ s$ = deleteBy $(=)\ e\ s$

```
(* ------------------------ *)
(* union                    *)
(* ------------------------ *)
```

val $unionBy$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \text{ORDERING}) \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha$
val $union$: $\forall\ \alpha.\ SetType\ \alpha \Rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha$
declare $ocaml$ target_rep function union = 'Pset.(union)'
declare hol target_rep function union = infix 'UNION'
declare $isabelle$ target_rep function union = infix '\<union>'
declare coq target_rep function unionBy = 'set_union_by'
declare tex target_rep function union = infix '$\cup$'
let inline $\{coq\}$ $union$ = unionBy setElemCompare

assert $union_1$: $(\{(1 : \text{NAT}); 2; 3\} \cup \{3; 2; 4\} = \{1; 2; 3; 4\})$
lemma $union\_in$: $(\forall\ e\ s_1\ s_2.\ e \in (s_1 \cup s_2) \longleftrightarrow (e \in s_1 \vee e \in s_2))$

```
(* ------------------------ *)
(* bigunion                 *)
(* ------------------------ *)
```

val $bigunion$: $\forall\ \alpha.\ SetType\ \alpha \Rightarrow \text{SET}\ (\text{SET}\ \alpha) \rightarrow \text{SET}\ \alpha$
val $bigunionBy$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \text{ORDERING}) \rightarrow \text{SET}\ (\text{SET}\ \alpha) \rightarrow \text{SET}\ \alpha$

let $bigunion\ bs$ = $\{x \mid \forall\ s \in bs\ x \in s \mid \text{true}\}$

declare $ocaml$ target_rep function bigunionBy = 'Pset.bigunion'
let inline $\{ocaml\}$ $bigunion$ = bigunionBy setElemCompare

declare *hol* target_rep function bigunion = `'BIGUNION'`
declare *isabelle* target_rep function bigunion = `'\<Union>'`
declare *tex* target_rep function bigunion = `'$\bigcup$'`

assert $bigunion_0$: $(\bigcup \{\{(1 : \text{NAT})\}\} = \{1\})$
assert $bigunion_1$: $(\bigcup \{\{(1 : \text{NAT}); 2; 3\} ;\ \{3; 2; 4\}\} = \{1; 2; 3; 4\})$
assert $bigunion_2$: $(\bigcup \{\{(1 : \text{NAT}); 2; 3\} ;\ \{3; 2; 4\};\ \{\}\} = \{1; 2; 3; 4\})$
assert $bigunion_3$: $(\bigcup \{\{(1 : \text{NAT}); 2; 3\} ;\ \{3; 2; 4\};\ \{5\}\} = \{1; 2; 3; 4; 5\})$
lemma *bigunion_in* : $(\forall\ e\ bs.\ e \in \bigcup bs \longleftrightarrow (\exists\ s.\ s \in bs \wedge e \in s))$

```
(* ------------------------ *)
(* difference               *)
(* ------------------------ *)
```

val *differenceBy* : $\forall\ \alpha.\ (\alpha \to \alpha \to \text{ORDERING}) \to \text{SET}\ \alpha \to \text{SET}\ \alpha \to \text{SET}\ \alpha$
val *difference* : $\forall\ \alpha.\ SetType\ \alpha \Rightarrow \text{SET}\ \alpha \to \text{SET}\ \alpha \to \text{SET}\ \alpha$
declare *ocaml* target_rep function difference = `'Pset.diff'`
declare *hol* target_rep function difference = infix `'DIFF'`
declare *isabelle* target_rep function difference = infix `'-'`
declare *coq* target_rep function differenceBy = `'set_diff_by'`
let inline {*coq*} *difference* = differenceBy setElemCompare

let inline \ = difference

assert $difference_1$: (difference $\{(1 : \text{NAT}); 2; 3\}\ \{3; 2; 4\} = \{1\}$)
lemma *difference_in* : $(\forall\ e\ s_1\ s_2.\ e \in (\text{difference}\ s_1\ s_2) \longleftrightarrow (e \in s_1 \wedge \neg\ (e \in s_2)))$

```
(* ------------------------ *)
(* intersection             *)
(* ------------------------ *)
```

val *intersection* : $\forall\ \alpha.\ SetType\ \alpha \Rightarrow \text{SET}\ \alpha \to \text{SET}\ \alpha \to \text{SET}\ \alpha$
val *intersectionBy* : $\forall\ \alpha.\ (\alpha \to \alpha \to \text{ORDERING}) \to \text{SET}\ \alpha \to \text{SET}\ \alpha \to \text{SET}\ \alpha$

declare *ocaml* target_rep function intersection = `'Pset.inter'`
declare *hol* target_rep function intersection = infix `'INTER'`
declare *isabelle* target_rep function intersection = infix `'\<inter>'`
declare *coq* target_rep function intersectionBy = `'set_inter_by'`
declare *tex* target_rep function intersection = infix `'$\cap$'`
let inline {*coq*} *intersection* = intersectionBy setElemCompare
let inline *inter* = $(\cap)$
declare *tex* target_rep function inter = infix `'$\cap$'`

assert $intersection_1$: $(\{1; 2; 3\} \cap \{(3 : \text{NAT}); 2; 4\} = \{2; 3\})$
lemma *intersection_in* : $(\forall\ e\ s_1\ s_2.\ e \in (s_1 \cap s_2) \longleftrightarrow (e \in s_1 \wedge e \in s_2))$

```
(* ------------------------ *)
(* map                      *)
(* ------------------------ *)
```

val *map* : $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta \Rightarrow (\alpha \to \beta) \to \text{SET}\ \alpha \to \text{SET}\ \beta$ `(* before image *)`
let *map* $f\ s = \{\ f\ e \mid \forall\ e \in s \mid$ true $\}$

val *mapBy* : $\forall\ \alpha\ \beta.\ (\beta \to \beta \to \text{ORDERING}) \to (\alpha \to \beta) \to \text{SET}\ \alpha \to \text{SET}\ \beta$

declare *ocaml* target_rep function mapBy = `'Pset.map'`

let inline {*ocaml*} *map* = mapBy setElemCompare
declare *hol* target_rep function map = `'IMAGE'`
declare *isabelle* target_rep function map = `'Set.image'`

assert map_1 : (map succ $\{(2 : \text{NAT}); 3; 4\} = \{5; 4; 3\}$)
assert map_2 : (map (fun $n \rightarrow n * 3$) $\{(2 : \text{NAT}); 3; 4\} = \{6; 9; 12\}$)

```
(* ------------------------ *)
(* min and max              *)
(* ------------------------ *)
```

val *findMin* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{MAYBE}\ \alpha$
val *findMax* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{MAYBE}\ \alpha$

```
(* Informal, since THE is not supported by all backends
val findMinBy : forall 'a.  ('a -> 'a -> bool) -> ('a -> 'a -> bool) -> set 'a -> maybe
'a
let findMinBy le eq s = THE (fun e -> ((memberBy eq e s) && (forall (e2 IN s). le e e2)))

let inline findMin = findMinBy (<=) (=)
let inline findMax = findMinBy (>=) (=)
*)
```

declare *ocaml* target_rep function findMin = `'Pset.min_elt_opt'`
declare *ocaml* target_rep function findMax = `'Pset.max_elt_opt'`

```
(* ------------------------ *)
(* fromList                 *)
(* ------------------------ *)
```

val *fromList* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{SET}\ \alpha$ `(* before from_list *)`
val *fromListBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \text{ORDERING})\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{SET}\ \alpha$

declare *ocaml* target_rep function fromListBy = `'Pset.from_list'`
let inline {*ocaml*} *fromList* = fromListBy setElemCompare
declare *hol* target_rep function fromList = `'LIST_TO_SET'`
declare *isabelle* target_rep function fromList = `'List.set'`
declare *coq* target_rep function fromListBy = `'set_from_list_by'`
let inline {*coq*} *fromList* = fromListBy setElemCompare

assert $fromList_1$: (fromList $[(2 : \text{NAT}); 4; 3] = \{2; 3; 4\}$)
assert $fromList_2$: (fromList $[(2 : \text{NAT}); 2; 3; 2; 4] = \{2; 3; 4\}$)
assert $fromList_3$: (fromList ($[]$: LIST NAT) $= \{\}$)

```
(* ------------------------ *)
(* Sigma                    *)
(* ------------------------ *)
```

val *sigma* : $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ (\alpha\ \rightarrow\ \text{SET}\ \beta)\ \rightarrow\ \text{SET}\ (\alpha\ *\ \beta)$
val *sigmaBy* : $\forall\ \alpha\ \beta.\ ((\alpha\ *\ \beta)\ \rightarrow\ (\alpha\ *\ \beta)\ \rightarrow\ \text{ORDERING})\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ (\alpha\ \rightarrow\ \text{SET}\ \beta)\ \rightarrow\ \text{SET}\ (\alpha\ *\ \beta)$

```
declare ocaml target_rep function sigmaBy = 'Pset.sigma'

let sigma sa sb = { (a, b) | ∀ a ∈ sa b ∈ sb a | true }
let inline {ocaml} sigma = sigmaBy setElemCompare

declare isabelle target_rep function sigma = 'Sigma'
declare coq target_rep function sigmaBy = 'set_sigma_by'
let inline {coq} sigma = sigmaBy setElemCompare
declare hol target_rep function sigma = 'SET_SIGMA'

assert Sigma_1 : (sigma {(2 : NAT); 3} (fun n → {n*2; n * 3}) = {(2, 4); (2, 6); (3, 6); (3, 9)})
lemma Sigma_2 : (∀ sa sb a b. ((a, b) ∈ sigma sa sb) ⟷ ((a ∈ sa) ∧ (b ∈ sb a)))


(* ------------------------ *)
(* cross product            *)
(* ------------------------ *)

val cross : ∀ α β. SetType α, SetType β ⇒ SET α → SET β → SET (α * β)
val crossBy : ∀ α β. ((α * β) → (α * β) → ORDERING) → SET α → SET β → SET (α * β)

declare ocaml target_rep function crossBy = 'Pset.cross'

let cross s_1 s_2 = { (e_1, e_2) | ∀ e_1 ∈ s_1 e_2 ∈ s_2 | true }

declare isabelle target_rep function cross = infix '\<times>'
declare hol target_rep function cross = infix 'CROSS'
declare tex target_rep function cross = infix '$\times$'
let inline {ocaml} cross = crossBy setElemCompare

lemma cross_by_sigma : ∀ s_1 s_2. s_1 × s_2 = sigma s_1 (const s_2)
assert cross_1 : ({(2 : NAT); 3} × {true; false} = {(2, true); (3, true); (2, false); (3, false)})


(* ------------------------ *)
(* finite                   *)
(* ------------------------ *)

val finite : ∀ α. SetType α ⇒ SET α → 𝔹

let inline {ocaml; coq} finite _s = true
declare hol target_rep function finite = 'FINITE'
declare isabelle target_rep function finite = 'finite'


(* ----------------------------*)
(* fixed point                 *)
(* --------------------------- *)

val leastFixedPoint : ∀ α. SetType α
  ⇒ NAT → (SET α → SET α) → SET α → SET α
let rec leastFixedPoint bound f x =
  match bound with
  | 0 → x
  | bound' + 1 → let fx = f x in
                if fx ⊆ x then x
                else leastFixedPoint bound' f (fx ∪ x)
  end
```

assert $\mathit{lfp\_empty}_0$: leastFixedPoint 0 (map (fun $x \;\rightarrow\; x$)) ({} : SET NAT) = {}
assert $\mathit{lfp\_empty}_1$: leastFixedPoint 1 (map (fun $x \;\rightarrow\; x$)) ({} : SET NAT) = {}
assert $\mathit{lfp\_saturate\_neg}_1$: leastFixedPoint 1 (map (fun $x \;\rightarrow\; -x$)) ({1; 2; 3} : SET INT) = {−3; −2; −1; 1; 2; 3}

assert $\mathit{lfp\_saturate\_neg}_2$: leastFixedPoint 2 (map (fun $x \;\rightarrow\; -x$)) ({1; 2; 3} : SET INT) = {−3; −2; −1; 1; 2; 3}

assert $\mathit{lfp\_saturate\_mod}_3$: leastFixedPoint 3 (map (fun $x \;\rightarrow\; (2*x)$ mod 5)) ({1} : SET NAT) = {1; 2; 3; 4}

assert $\mathit{lfp\_saturate\_mod}_4$: leastFixedPoint 4 (map (fun $x \;\rightarrow\; (2*x)$ mod 5)) ({1} : SET NAT) = {1; 2; 3; 4}

assert $\mathit{lfp\_saturate\_mod}_5$: leastFixedPoint 5 (map (fun $x \;\rightarrow\; (2*x)$ mod 5)) ({1} : SET NAT) = {1; 2; 3; 4}

assert $\mathit{lfp\_termination}$: {1; 3; 5; 7; 9} $\subseteq$ leastFixedPoint 5 (map (fun $x \;\rightarrow\; 2+x$)) {(1 : $\mathbb{N}$)}

12 Map

```
(*********************************************************************************)
(* A library for finite maps                                                     *)
(*********************************************************************************)

(* ============================================================================= *)
(* Header                                                                        *)
(* ============================================================================= *)
```

declare $\{isabelle; ocaml; hol\}$ rename module $=$ lem_map

open import *Bool Basic_classes Function Maybe List Tuple Set Num*
open import $\{hol\}$ *finite_mapTheory finite_mapLib*

type MAP $'k\ 'v$
declare *ocaml* target_rep type MAP $=$ `'Pmap.map'`
declare *isabelle* target_rep type MAP $=$ `'Map.map'`
declare *hol* target_rep type MAP $=$ `'fmap'`
declare *coq* target_rep type MAP $=$ `'fmap'`

```
(* ----------------------------------------------------------------------------- *)
(* Map equality.                                                                 *)
(* ----------------------------------------------------------------------------- *)
```

val *mapEqual* : $\forall\ 'k\ 'v.\ Eq\ 'k,\ Eq\ 'v\ \Rightarrow$ MAP $'k\ 'v \rightarrow$ MAP $'k\ 'v \rightarrow \mathbb{B}$
val *mapEqualBy* : $\forall\ 'k\ 'v.\ ('k \rightarrow\ 'k \rightarrow \mathbb{B}) \rightarrow ('v \rightarrow\ 'v \rightarrow \mathbb{B}) \rightarrow$ MAP $'k\ 'v \rightarrow$ MAP $'k\ 'v \rightarrow \mathbb{B}$

declare *ocaml* target_rep function mapEqualBy $eq\_k\ eq\_v$ $=$ `'Pmap.equal'` eq_v
declare *coq* target_rep function mapEqualBy $=$ `'fmap_equal_by'`
let inline $\sim\{hol; isabelle\}$ *mapEqual* $=$ mapEqualBy (=) (=)
let inline $\{hol; isabelle\}$ *mapEqual* $=$ unsafe_structural_equality

instance $\forall\ 'k\ 'v.\ Eq\ 'k,\ Eq\ 'v\ \Rightarrow\ (Eq$ (MAP $'k\ 'v$))
  let $=$ $=$ mapEqual
  let $<>$ $m_1\ m_2\ =\ \neg$ (mapEqual $m_1\ m_2$)
end

```
(* ----------------------------------------------------------------------------- *)
(* Map type class                                                                *)
(* ----------------------------------------------------------------------------- *)
```

class (*MapKeyType* α)
  val $\{ocaml; coq\}$ *mapKeyCompare* : $\alpha \rightarrow \alpha \rightarrow$ ORDERING
end

default_instance $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow$ (*MapKeyType* α)
  let *mapKeyCompare* $=$ setElemCompare
end

```
(* ----------------------------------------------------------------------------- *)
(* Empty maps                                                                    *)
(* ----------------------------------------------------------------------------- *)
```

val *empty* : $\forall$ 'k 'v. *MapKeyType* 'k $\Rightarrow$ MAP 'k 'v
val *emptyBy* : $\forall$ 'k 'v. ('k $\rightarrow$ 'k $\rightarrow$ ORDERING) $\rightarrow$ MAP 'k 'v

declare *ocaml* target_rep function emptyBy = `'Pmap.empty'`

let inline {*ocaml*} *empty* = emptyBy mapKeyCompare
declare *coq* target_rep function empty = `'fmap_empty'`
declare *hol* target_rep function empty = `'FEMPTY'`
declare *isabelle* target_rep function empty = `'Map.empty'`

```
(* -------------------------------------------------------------------------- *)
(* Insertion                                                                  *)
(* -------------------------------------------------------------------------- *)
```

val *insert* : $\forall$ 'k 'v. *MapKeyType* 'k $\Rightarrow$ 'k $\rightarrow$ 'v $\rightarrow$ MAP 'k 'v $\rightarrow$ MAP 'k 'v

declare *coq* target_rep function insert = `'fmap_add'`
declare *ocaml* target_rep function insert = `'Pmap.add'`
`(* declare hol      target_rep function insert k v m = ‘FUPDATE‘ m (k,v) *)`
declare *hol* target_rep function insert k v m = special "$\%e|+(\%e,\%e)$" m k v

declare *isabelle* target_rep function insert = `'map_update'`

```
(* -------------------------------------------------------------------------- *)
(* Singleton                                                                  *)
(* -------------------------------------------------------------------------- *)
```

val *singleton* : $\forall$ 'k 'v. *MapKeyType* 'k $\Rightarrow$ 'k $\rightarrow$ 'v $\rightarrow$ MAP 'k 'v
let inline *singleton* k v = insert k v empty

assert *insert_equal_singleton* : (mapEqual (insert (42 : NAT) false empty)
(singleton 42 false))
assert *commutative_insert*$_1$: (mapEqual
(insert (8 : NAT) true (insert 5 false empty))
(insert 5 false (insert 8 true empty)))
assert *commutative_insert*$_2$: ($\neg$ (mapEqual
(insert (8 : NAT) true (insert 8 false empty))
(insert 8 false (insert 8 true empty))))

```
(* -------------------------------------------------------------------------- *)
(* Emptyness check                                                            *)
(* -------------------------------------------------------------------------- *)
```

val *null* : $\forall$ 'k 'v. *MapKeyType* 'k, *Eq* 'k, *Eq* 'v $\Rightarrow$ MAP 'k 'v $\rightarrow$ $\mathbb{B}$
let inline *null* m = (m = empty)

declare *coq* target_rep function null = `'fmap_is_empty'`
declare *ocaml* target_rep function null = `'Pmap.is_empty'`

assert *empty_null* : (null (empty : MAP NAT $\mathbb{B}$))

```
(* -------------------------------------------------------------------------- *)
(* lookup                                                                     *)
```

```
(* -------------------------------------------------------------------------- *)

val lookupBy : ∀ 'k 'v. ('k → 'k → ORDERING) → 'k → MAP 'k 'v → MAYBE 'v
declare coq target_rep function lookupBy = 'fmap_lookup_by'

val lookup : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → MAP 'k 'v → MAYBE 'v
let inline {coq} lookup = lookupBy mapKeyCompare
declare isabelle target_rep function lookup k m = '' m k
declare hol target_rep function lookup k m = 'FLOOKUP' m k
declare ocaml target_rep function lookup = 'Pmap.lookup'

assert lookup_insert_1 : (lookup 16 (insert (16 : NAT) true empty) = Just true)
assert lookup_insert_2 : (lookup 16 (insert 36 false (insert (16 : NAT) true empty)) = Just true )
assert lookup_insert_3 : (lookup 36 (insert 36 false (insert (16 : NAT) true empty)) = Just false )

assert lookup_empty_0 : (lookup 25 (empty : MAP NAT 𝔹) = Nothing)
assert find_insert_0 : (lookup 16 (insert (16 : NAT) true empty) = Just true)

lemma lookup_empty : (∀ k. lookup k empty = Nothing)
lemma lookup_insert : (∀ k k' v m. lookup k (insert k' v m) = (if (k = k') then Just v else lookup k m))

(* -------------------------------------------------------------------------- *)
(* findWithDefault                                                            *)
(* -------------------------------------------------------------------------- *)

val findWithDefault : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → 'v → MAP 'k 'v → 'v
let inline findWithDefault k v m = fromMaybe v (lookup k m)

(* -------------------------------------------------------------------------- *)
(* from lists                                                                 *)
(* -------------------------------------------------------------------------- *)

val fromList : ∀ 'k 'v. MapKeyType 'k ⇒ LIST ('k * 'v) → MAP 'k 'v
let fromList l = foldl (fun m (k, v) → insert k v m) empty l

declare isabelle target_rep function fromList l = 'Map.map_of' (reverse l)
declare hol target_rep function fromList l = 'FUPDATE_LIST' 'FEMPTY' l

assert fromList_0 : (fromList [((2 : NAT), true); ((3 : NAT), true); ((4 : NAT), false)] =
            fromList [((4 : NAT), false); ((3 : NAT), true); ((2 : NAT), true)])
(* later entries have priority *)
assert fromList_1 : (fromList [((2 : NAT), true); ((2 : NAT), false); ((3 : NAT), true); ((4 : NAT), false)] =
            fromList [((4 : NAT), false); ((3 : NAT), true); ((2 : NAT), false)])


(* -------------------------------------------------------------------------- *)
(* to sets / domain / range                                                   *)
(* -------------------------------------------------------------------------- *)

val toSet : ∀ 'k 'v. MapKeyType 'k, SetType 'k, SetType 'v ⇒ MAP 'k 'v → SET ('k * 'v)
val toSetBy : ∀ 'k 'v. (('k * 'v) → ('k * 'v) → ORDERING) → MAP 'k 'v → SET ('k * 'v)

declare ocaml target_rep function toSetBy = 'Pmap.bindings'
let inline {ocaml} toSet = toSetBy setElemCompare
declare isabelle target_rep function toSet = 'map_to_set'
declare hol target_rep function toSet = 'FMAP_TO_SET'
declare coq target_rep function toSet = 'id'
```

assert $toSet_0$: (toSet (empty : MAP NAT $\mathbb{B}$) = {})
assert $toSet_1$: (toSet (fromList [((2 : NAT), true); (3, true); (4, false)]) =
{(2, true); (3, true); (4, false)})
assert $toSet_2$: (toSet (fromList [((2 : NAT), true); (3, true); (2, false); (4, false)]) =
{(2, false); (3, true); (4, false)})

val *domainBy* : $\forall$ 'k 'v. ('k → 'k → ORDERING) → MAP 'k 'v → SET 'k
val *domain* : $\forall$ 'k 'v. *MapKeyType* 'k, *SetType* 'k ⇒ MAP 'k 'v → SET 'k
declare *ocaml* target_rep function domain = `'Pmap.domain'`
declare *isabelle* target_rep function domain = `'Map.dom'`
declare *hol* target_rep function domain = `'FDOM'`
declare *coq* target_rep function domainBy = `'fmap_domain_by'`
let inline {*coq*} *domain* = domainBy setElemCompare

assert $domain_0$: (domain (empty : MAP NAT $\mathbb{B}$) = {})
assert $domain_1$: (domain (fromList [((2 : NAT), true); (3, true); (4, false)]) =
{2; 3; 4})
assert $domain_2$: (domain (fromList [((2 : NAT), true); (3, true); (2, false); (4, false)]) =
{2; 3; 4})

val *range* : $\forall$ 'k 'v. *MapKeyType* 'k, *SetType* 'v ⇒ MAP 'k 'v → SET 'v
val *rangeBy* : $\forall$ 'k 'v. ('v → 'v → ORDERING) → MAP 'k 'v → SET 'v

declare *ocaml* target_rep function rangeBy = `'Pmap.range'`
declare *hol* target_rep function range = `'FRANGE'`
declare *isabelle* target_rep function range = `'Map.ran'`
declare *coq* target_rep function rangeBy = `'map_range_by'`
let inline {*ocaml*; *coq*} *range* = rangeBy setElemCompare

assert $range_0$: (range (empty : MAP NAT $\mathbb{B}$) = {})
assert $range_1$: (range (fromList [((2 : NAT), true); (3, true); (4, false)]) =
{true; false})
assert $range_2$: (range (fromList [((2 : NAT), true); (3, true); (4, true)]) = {true})

```
(* -------------------------------------------------------------------------- *)
(* member                                                                     *)
(* -------------------------------------------------------------------------- *)
```

val *member* : $\forall$ 'k 'v. *MapKeyType* 'k, *SetType* 'k, *Eq* 'k ⇒ 'k → MAP 'k 'v → $\mathbb{B}$
let inline *member* k m = $k \in$ domain m
declare *ocaml* target_rep function member = `'Pmap.mem'`

val *notMember* : $\forall$ 'k 'v. *MapKeyType* 'k, *SetType* 'k, *Eq* 'k ⇒ 'k → MAP 'k 'v → $\mathbb{B}$
let inline *notMember* k m = $\neg$ (member k m)

assert $member\_insert_1$: (member 16 (insert (16 : NAT) true empty))
assert $member\_insert_2$: ($\neg$ (member 25 (insert (16 : NAT) true empty)))
assert $member\_insert_3$: (member 16 (insert 36 false (insert (16 : NAT) true empty)))

lemma *member_empty* : ($\forall$ k. $\neg$ (member k empty))
lemma *member_insert* : ($\forall$ k k' v m. member k (insert k' v m) = (($k = k'$) $\vee$ member k m))

```
(* -------------------------------------------------------------------------- *)
(* Quantification                                                             *)
```

```
(* -------------------------------------------------------------------------- *)

val any : ∀ 'k 'v. MapKeyType 'k, Eq 'v ⇒ ('k → 'v → 𝔹) → MAP 'k 'v → 𝔹
val all : ∀ 'k 'v. MapKeyType 'k, Eq 'v ⇒ ('k → 'v → 𝔹) → MAP 'k 'v → 𝔹

let all P m = (∀ k v. (P k v ∧ (lookup k m = Just v)))
let inline any P m = ¬ (all (fun k v → ¬ (P k v)) m)

declare ocaml target_rep function any = 'Pmap.exist'
declare ocaml target_rep function all = 'Pmap.for_all'
declare coq target_rep function all = 'fmap_all'
declare isabelle target_rep function any = 'map_any'
declare isabelle target_rep function all = 'map_all'
declare hol target_rep function all P = 'FEVERY' (uncurry P)

assert any_0 : (any (fun _k v → v) (insert 36 false (insert (16 : NAT) true empty)))
assert any_1 : (¬ (any (fun _k v → v) (insert 36 false (insert (16 : NAT) false empty))))
assert any_2 : (any (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) true empty)))
assert any_3 : (¬ (any (fun _k v → ¬ v) (insert 36 true (insert (16 : NAT) true empty))))

assert all_0 : (all (fun _k v → v) (insert 36 true (insert (16 : NAT) true empty)))
assert all_1 : (¬ (all (fun _k v → v) (insert 36 true (insert (16 : NAT) false empty))))
assert all_2 : (all (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) false empty)))
assert all_3 : (¬ (all (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) true empty))))


(* -------------------------------------------------------------------------- *)
(* Set-like operations.                                                       *)
(* -------------------------------------------------------------------------- *)
val deleteBy : ∀ 'k 'v. ('k → 'k → ORDERING) → 'k → MAP 'k 'v → MAP 'k 'v
val delete : ∀ 'k 'v. MapKeyType 'k ⇒ 'k → MAP 'k 'v → MAP 'k 'v
val deleteSwap : ∀ 'k 'v. MapKeyType 'k ⇒ MAP 'k 'v → 'k → MAP 'k 'v

declare coq target_rep function deleteBy = 'fmap_delete_by'
declare ocaml target_rep function delete = 'Pmap.remove'
declare isabelle target_rep function delete = 'map_remove'
declare hol target_rep function deleteSwap = infix '\\'
let inline {hol} delete k m = deleteSwap m k
let inline {coq} delete = deleteBy mapKeyCompare
let inline {coq} deleteSwap m k = delete k m

assert delete_insert_1 : (¬ (member (5 : NAT) (delete 5 (insert 5 true empty))))
assert delete_insert_2 : (member (7 : NAT) (delete 5 (insert 7 true empty)))
assert delete_delete : (null (delete (5 : NAT) (delete (5 : NAT) (insert 5 true empty))))

val union : ∀ 'k 'v. MapKeyType 'k ⇒ MAP 'k 'v → MAP 'k 'v → MAP 'k 'v
declare coq target_rep function union = 'app'
declare ocaml target_rep function union = 'Pmap.union'
declare isabelle target_rep function union = infix '++'
declare hol target_rep function union = 'FUNION'

val unions : ∀ 'k 'v. MapKeyType 'k ⇒ LIST (MAP 'k 'v) → MAP 'k 'v
let inline unions = foldr (union) empty


(* -------------------------------------------------------------------------- *)
(* Maps (in the functor sense).                                               *)
(* -------------------------------------------------------------------------- *)
```

val map : $\forall$ $'k$ $'v$ $'w$. $MapKeyType$ $'k$ $\Rightarrow$ $('v \rightarrow 'w) \rightarrow$ MAP $'k$ $'v$ $\rightarrow$ MAP $'k$ $'w$

declare hol target_rep function map = infix `'o_f'`
declare coq target_rep function map = `'fmap_map'`
declare $ocaml$ target_rep function map = `'Pmap.map'`
declare $isabelle$ target_rep function map = `'map_image'`

assert map_0 : (map (fun b $\rightarrow$ $\neg$ b) (insert (2 : NAT) true (insert (3 : NAT) false empty)) =
insert (2 : NAT) false (insert (3 : NAT) true empty))

```
(* -------------------------------------------------------------------------- *)
(* Cardinality                                                                *)
(* -------------------------------------------------------------------------- *)
```

val $size$: $\forall$ $'k$ $'v$. $MapKeyType$ $'k$, $SetType$ $'k$ $\Rightarrow$ MAP $'k$ $'v$ $\rightarrow$ NAT
let inline $size$ m = Set.size (domain m)

declare $ocaml$ target_rep function size = `'Pmap.cardinal'`
declare hol target_rep function size = `'FCARD'`

assert $empty\_size$: (size (empty : MAP NAT $\mathbb{B}$) = 0)
assert $singleton\_size$: (size (singleton (2 : NAT) (3 : NAT)) = 1)

13 Map_extra

```
(******************************************************************************)
(* A library for finite maps                                                  *)
(******************************************************************************)

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

declare $\{isabelle;\ hol;\ ocaml\}$ rename module $=$ lem_map_extra

open import *Bool Basic_classes Function Maybe List Num Set Map*

```
(* -------------------------------------------------------------------------- *)
(* find                                                                       *)
(* -------------------------------------------------------------------------- *)
```

val *find* : $\forall\ 'k\ 'v.\ MapKeyType\ 'k\ \Rightarrow\ 'k\ \rightarrow\ \textsc{map}\ 'k\ 'v\ \rightarrow\ 'v$
let *find* $k\ m\ =$ match (lookup $k\ m$) with Just $x\ \rightarrow\ x$ end

declare *ocaml* target_rep function find $=$ `'Pmap.find'`
declare *isabelle* target_rep function find $=$ `'map_find'`
declare *hol* target_rep function find $k\ m\ =$ `'FAPPLY'` $m\ k$

declare compile_message find $=$ "$find is only defined if the key is found. Use lookup instead and handle the not-found case explicitly.$"

assert $find\_insert_1$: (find 16 (insert (16 : $\textsc{nat}$) true empty) = true)
assert $find\_insert_2$: (find 36 (insert 36 false (insert (16 : $\textsc{nat}$) true empty)) = false)

```
(* -------------------------------------------------------------------------- *)
(* from sets / domain / range                                                 *)
(* -------------------------------------------------------------------------- *)
```

val *fromSet* : $\forall\ 'k\ 'v.\ MapKeyType\ 'k\ \Rightarrow\ ('k\ \rightarrow\ 'v)\ \rightarrow\ \textsc{set}\ 'k\ \rightarrow\ \textsc{map}\ 'k\ 'v$
let *fromSet* $f\ s\ =$ Set_helpers.fold (fun $k\ m\ \rightarrow$ Map.insert $k\ (f\ k)\ m$) s Map.empty

declare compile_message fromSet $=$ "$fromSet only works for finite sets, use carefully.$"

declare *ocaml* target_rep function fromSet $=$ `'Pmap.from_set'`
declare *hol* target_rep function fromSet $=$ `'FUN_FMAP'`

assert $fromSet_0$: (fromSet succ ($\emptyset$: $\textsc{set nat}$) = Map.empty)
assert $fromSet_1$: (fromSet succ $\{(2 : \textsc{nat});\ 3;\ 4\}$) = Map.fromList [(2, 3); (3, 4); (4, 5)]

14 Maybe_extra

```
(******************************************************************************)
(* extra functions for maybe / option                                         *)
(*                                                                            *)
(******************************************************************************)
```

declare {*isabelle*; *hol*; *ocaml*} rename module = lem_maybe_extra

open import *Basic_classes Maybe*

```
(* ----------------------- *)
(* fromJust                *)
(* ----------------------- *)
```

val *fromJust* : $\forall\ \alpha.$ MAYBE $\alpha \rightarrow \alpha$
let *fromJust* (Just v) = v
declare termination_argument fromJust = automatic
declare compile_message fromJust = "*fromJustisonlydefinedonJust.Betteruse‘fromMaybe‘oruseexplicitmachingtohandlet case.*"

declare *hol* target_rep function fromJust = 'THE'
declare *isabelle* target_rep function fromJust = 'the'

15 Either

```
(******************************************************************************)
(* A library for sum types                                                    *)
(******************************************************************************)

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)

declare {isabelle; hol} rename module = Lem_either
declare {ocaml} rename module = Lem_either

open import Bool Basic_classes List Tuple
open import {hol} sumTheory
open import {ocaml} Either

type EITHER α β
  = LEFT of α
  | RIGHT of β

declare ocaml target_rep type EITHER = 'either'
declare isabelle target_rep type EITHER = 'sum'
declare hol target_rep type EITHER = 'sum'
declare coq target_rep type EITHER = 'sum'

declare isabelle target_rep function Left = 'Inl'
declare isabelle target_rep function Right = 'Inr'
declare ocaml target_rep function Left = 'Left'
declare ocaml target_rep function Right = 'Right'
declare hol target_rep function Left = 'INL'
declare hol target_rep function Right = 'INR'
declare coq target_rep function Left = 'inl'
declare coq target_rep function Right = 'inr'


(* -------------------------------------------------------------------------- *)
(* Equality.                                                                  *)
(* -------------------------------------------------------------------------- *)

val eitherEqual : ∀ α β. Eq α, Eq β ⇒ (EITHER α β) → (EITHER α β) → 𝔹
val eitherEqualBy : ∀ α β. (α → α → 𝔹) → (β → β → 𝔹) → (EITHER α β) → (EITHER α β) → 𝔹


let eitherEqualBy eql eqr (left : EITHER α β) (right : EITHER α β) =
  match (left, right) with
    | (Left l, Left l') → eql l l'
    | (Right r, Right r') → eqr r r'
    | _ → false
  end
let eitherEqual = eitherEqualBy (=) (=)

let inline {hol; isabelle} eitherEqual = unsafe_structural_equality
let inline {ocaml} eitherEqual = eitherEqualBy (=) (=)
declare ocaml target_rep function eitherEqualBy = 'Either.eitherEqualBy'

instance ∀ α β. Eq α, Eq β ⇒ (Eq (EITHER α β))
  let = = eitherEqual
```

```
  let <> x y  =  ¬ (eitherEqual x y)
end

assert either_equal1 :  (((Left false)  :  EITHER 𝔹 𝔹) = Left false)
assert either_equal2 :  (((Left true)  :  EITHER 𝔹 𝔹) ≠ Left false)
assert either_equal3 :  (((Left true)  :  EITHER 𝔹 𝔹) = Left true)
assert either_equal4 :  (((Right false)  :  EITHER 𝔹 𝔹) = Right false)
assert either_equal5 :  (((Right false)  :  EITHER 𝔹 𝔹) ≠ Right true)
assert either_equal6 :  (((Right true)  :  EITHER 𝔹 𝔹) ≠ Left true)
assert either_equal7 :  (((Left true)  :  EITHER 𝔹 𝔹) ≠ Right true)

assert either_pattern1 :  (match (Left true) with Left x  →  x | Right y  →  ¬ y end)
assert either_pattern2 :  (match (Right false) with Left x  →  x | Right y  →  ¬ y end)
assert either_pattern3 :  (¬ (match (Left false) with Left x  →  x | Right y  →  ¬ y end))
assert either_pattern4 :  (¬ (match (Right true) with Left x  →  x | Right y  →  ¬ y end))


(* -------------------------------------------------------------------------- *)
(* Utility functions.                                                         *)
(* -------------------------------------------------------------------------- *)

val isLeft  :  ∀ α β. EITHER α β  →  𝔹
let inline isLeft  =  function
  | Left _  →  true
  | Right _  →  false
end

declare hol target_rep function isLeft  =  'ISL'

assert isLeft1  :  (isLeft ((Left true) :  EITHER 𝔹 𝔹))
assert isLeft2  :  (¬ (isLeft ((Right true) :  EITHER 𝔹 𝔹)))

val isRight  :  ∀ α β. EITHER α β  →  𝔹
let inline isRight  =  function
  | Right _  →  true
  | Left _  →  false
end

declare hol target_rep function isRight  =  'ISR'

assert isRight1  :  (isRight ((Right true) :  EITHER 𝔹 𝔹))
assert isRight2  :  (¬ (isRight ((Left true) :  EITHER 𝔹 𝔹)))


val either  :  ∀ α β γ. (α  →  γ)  →  (β  →  γ)  →  EITHER α β  →  γ
let either fa fb x  =  match x with
  | Left a  →  fa a
  | Right b  →  fb b
end

declare ocaml target_rep function either  =  'Either.either_case'
declare isabelle target_rep function either  =  'sum_case'
declare hol target_rep function either fa fb x  =  'sum_CASE' x fa fb

assert either1 :  (either ((fun b → ¬ b)) (fun b  →  b) (Left true) = false)
assert either2 :  (either ((fun b → ¬ b)) (fun b  →  b) (Left false) = true)
assert either3 :  (either ((fun b → ¬ b)) (fun b  →  b) (Right true) = true)
assert either4 :  (either ((fun b → ¬ b)) (fun b  →  b) (Right false) = false)
```

```
val partitionEither : ∀ α β. list (either α β) → (list α * list β)
let rec partitionEither l = match l with
  | [] → ([], [])
  | x :: xs → begin
      let (ll, rl) = partitionEither xs in
      match x with
        | Left l → (l::ll, rl)
        | Right r → (ll, r::rl)
      end
    end
end
declare termination_argument partitionEither = automatic
declare {hol} rename function partitionEither = SUM_PARTITION

declare isabelle target_rep function partitionEither = 'sum_partition'
declare ocaml target_rep function partitionEither = 'Either.either_partition'

assert partitionEither_1 : (partitionEither [Left true; Right false; Right false; Left false; Right true] = ([true; false], [false; false; tru


val lefts : ∀ α β. list (either α β) → list α
let inline lefts l = fst (partitionEither l)

assert lefts_1 : ((lefts [Left true; Right false; Right false; Left false; Right true]) = [true; false])


val rights : ∀ α β. list (either α β) → list β
let inline rights l = snd (partitionEither l)


assert rights_1 : (rights [Left true; Right false; Right false; Left false; Right true] = [false; false; true])
```

16 Relation

```
(**************************************************************************)
(* A library for binary relations                                         *)
(**************************************************************************)

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

declare $\{isabelle;\ ocaml;\ hol\}$ rename module $=$ lem_relation

open import *Bool Basic_classes Tuple Set Num*
open import $\{hol\}$ *set_relationTheory*

```
(* ========================================================================== *)
(* The type of relations                                                      *)
(* ========================================================================== *)
```

type $\text{REL\_PRED}\ \alpha\ \beta\ =\ \alpha\ \to\ \beta\ \to\ \mathbb{B}$
type $\text{REL\_SET}\ \alpha\ \beta\ =\ \text{SET}\ (\alpha\ *\ \beta)$

```
(* Binary relations are usually represented as either
   sets of pairs (rel_set) or as curried functions (rel_pred).

   The choice depends on taste and the backend. Lem should not take a
   decision, but supports both representations. There is an abstract type
   pred, which can be converted to both representations. The representation
   of pred itself then depends on the backend. However, for the time beeing,
   let's implement relations as sets to get them working more quickly. *)
```

type $\text{REL}\ \alpha\ \beta\ =\ \text{REL\_SET}\ \alpha\ \beta$

val *relToSet* $:\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \to\ \text{REL\_SET}\ \alpha\ \beta$
val *relFromSet* $:\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \text{REL\_SET}\ \alpha\ \beta\ \to\ \text{REL}\ \alpha\ \beta$

let inline *relToSet* $s\ =\ s$
let inline *relFromSet* $r\ =\ r$

val *relEq* $:\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \to\ \text{REL}\ \alpha\ \beta\ \to\ \mathbb{B}$
let *relEq* $r_1\ r_2\ =\ (\text{relToSet}\ r_1 = \text{relToSet}\ r_2)$

```
(*
instance forall 'a 'b. SetType 'a, SetType 'b => (Eq (rel 'a 'b))
  let (=) = relEq
end
*)
```

lemma *relToSet_inv* : $(\forall\ r.\ \text{relFromSet}\ (\text{relToSet}\ r) = r)$

val *relToPred* $:\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \to\ \text{REL\_PRED}\ \alpha\ \beta$
val *relFromPred* $:\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \text{SET}\ \alpha\ \to\ \text{SET}\ \beta\ \to\ \text{REL\_PRED}\ \alpha\ \beta\ \to$ $\text{REL}\ \alpha\ \beta$

let *relToPred* $r\ =\ (\text{fun}\ x\ y\ \to\ (x,\ y) \in \text{relToSet}\ r)$
let *relFromPred* $xs\ ys\ p\ =\ \text{Set.filter}\ (\text{fun}\ (x,\ y)\ \to\ p\ x\ y)\ (xs \times ys)$

let inline $\{hol\}$ *relToPred* $r\ x\ y\ =\ (x,\ y) \in \text{relToSet}\ r$

declare $\{hol\}$ rename function relToPred $=$ rel_to_pred

assert rel_basic_0 : relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4)\} =$ relFromPred $\{2;\ 3\}\ \{1; 2; 3; 4; 5; 6\}$ (fun $x\ y\ \rightarrow$ $y = x + 1$)
assert rel_basic_1 : relToSet (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4)\}) = \{(2, 3);\ (3, 4)\}$
assert rel_basic_2 : relToPred (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4)\}$) 2 3

```
(* ========================================================================== *)
(* Basic Operations                                                           *)
(* ========================================================================== *)

(* ----------------------- *)
(* membership test         *)
(* ----------------------- *)
```

val $inRel$: $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \alpha\ \rightarrow\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \mathbb{B}$
let inline $inRel\ a\ b\ rel\ =\ (a,\ b) \in$ relToSet rel

lemma $inRel\_set$: $(\forall\ s\ a\ b.$ inRel $a\ b$ (relFromSet $s) = ((a,\ b) \in s))$
lemma $inRel\_pred$: $(\forall\ p\ a\ b\ sa\ sb.$ inRel $a\ b$ (relFromPred $sa\ sb\ p) = p\ a\ b \wedge a \in sa \wedge b \in sb)$

assert in_rel_0 : (inRel 2 3 (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (4, 5)\}$))
assert in_rel_1 : (inRel 4 5 (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (4, 5)\}$))
assert in_rel_2 : $\neg$ (inRel 3 2 (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (4, 5)\}$))
assert in_rel_3 : $\neg$ (inRel 7 4 (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (4, 5)\}$))

```
(* ----------------------- *)
(* empty relation          *)
(* ----------------------- *)
```

val $relEmpty$: $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta$
let inline $relEmpty\ =$ relFromSet $\{\}$

assert $relEmpty_0$: relToSet relEmpty $= (\{\}\ :\ \text{SET}\ (\text{NAT}\ *\ \text{NAT}))$
assert $relEmpty_1$: $\neg$ (inRel true $(2 : \text{NAT})$ relEmpty)

```
(* ----------------------- *)
(* Insertion               *)
(* ----------------------- *)
```

val $relAdd$: $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \alpha\ \rightarrow\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta$
let inline $relAdd\ a\ b\ r\ =$ relFromSet (insert $(a,\ b)$ (relToSet r))

assert $relAdd_0$: inRel $(2\ :\ \text{NAT})\ (3\ :\ \text{NAT})$ (relAdd 2 3 relEmpty)
assert $relAdd_1$: inRel $(4\ :\ \text{NAT})\ (5\ :\ \text{NAT})$ (relAdd 2 3 (relAdd 4 5 relEmpty))
assert $relAdd_2$: $\neg$ (inRel $(2\ :\ \text{NAT})\ (5\ :\ \text{NAT})$ (relAdd 2 3 (relAdd 4 5 relEmpty)))
assert $relAdd_3$: $\neg$ (inRel $(4\ :\ \text{NAT})\ (9\ :\ \text{NAT})$ (relAdd 2 3 (relAdd 4 5 relEmpty)))

lemma in_relAdd : $(\forall\ a\ b\ a'\ b'\ r.$ inRel $a\ b$ (relAdd $a'\ b'\ r) =$
$((a = a') \wedge (b = b')) \vee$ inRel $a\ b\ r)$

```
(* ----------------------- *)
(* Identity relation       *)
(* ----------------------- *)
```

val *relIdOn* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$
let *relIdOn* s = relFromPred $s\ s\ (=)$

val *relId* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha$
let $\sim\{coq;\ ocaml\}$ *relId* = $\{(x,\ x) \mid \forall\ x \mid \mathsf{true}\}$

lemma *relId_spec* : $(\forall\ x\ y\ s.\ (\text{inRel}\ x\ y\ (\text{relIdOn}\ s) \longleftrightarrow (x \in s \wedge (x = y))))$

assert rel_id_0 : $\text{inRel}\ (0 : \text{NAT})\ 0\ (\text{relIdOn}\ \{0;\ 1;\ 2;\ 3\})$
assert rel_id_1 : $\text{inRel}\ (2 : \text{NAT})\ 2\ (\text{relIdOn}\ \{0;\ 1;\ 2;\ 3\})$
assert rel_id_2 : $\neg\ (\text{inRel}\ (5 : \text{NAT})\ 5\ (\text{relIdOn}\ \{0;\ 1;\ 2;\ 3\}))$
assert rel_id_3 : $\neg\ (\text{inRel}\ (0 : \text{NAT})\ 2\ (\text{relIdOn}\ \{0;\ 1;\ 2;\ 3\}))$

```
(* ----------------------- *)
(* relation union          *)
(* ----------------------- *)
```

val *relUnion* : $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta$
let inline *relUnion* $r_1\ r_2$ = $\text{relFromSet}\ ((\text{relToSet}\ r_1) \cup (\text{relToSet}\ r_2))$

lemma *in_rel_union* : $(\forall\ a\ b\ r_1\ r_2.\ \text{inRel}\ a\ b\ (\text{relUnion}\ r_1\ r_2) = \text{inRel}\ a\ b\ r_1 \vee \text{inRel}\ a\ b\ r_2)$
assert rel_union_0 : $\text{relUnion}\ (\text{relAdd}\ (2 : \text{NAT})\ \mathsf{true}\ \text{relEmpty})\ (\text{relAdd}\ 5\ \mathsf{false}\ \text{relEmpty}) =$
$\text{relFromSet}\ \{(5,\ \mathsf{false});\ (2,\ \mathsf{true})\}$

```
(* ----------------------- *)
(* relation intersection   *)
(* ----------------------- *)
```

val *relIntersection* : $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta$
let inline *relIntersection* $r_1\ r_2$ = $\text{relFromSet}\ ((\text{relToSet}\ r_1) \cap (\text{relToSet}\ r_2))$

lemma *in_rel_inter* : $(\forall\ a\ b\ r_1\ r_2.\ \text{inRel}\ a\ b\ (\text{relIntersection}\ r_1\ r_2) = \text{inRel}\ a\ b\ r_1 \wedge \text{inRel}\ a\ b\ r_2)$
assert rel_inter_0 : $\text{relIntersection}\ (\text{relAdd}\ (2 : \text{NAT})\ \mathsf{true}\ (\text{relAdd}\ 7\ \mathsf{false}\ \text{relEmpty}))$
$(\text{relAdd}\ 7\ \mathsf{false}\ (\text{relAdd}\ 2\ \mathsf{false}\ \text{relEmpty})) =$
$\text{relFromSet}\ \{(7,\ \mathsf{false})\}$

```
(* ----------------------- *)
(* Relation Composition    *)
(* ----------------------- *)
```

val *relComp* : $\forall\ \alpha\ \beta\ \gamma.\ SetType\ \alpha,\ SetType\ \beta,\ SetType\ \gamma,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \beta\ \gamma\ \rightarrow\ \text{REL}\ \alpha\ \gamma$

let *relComp* $r_1\ r_2$ = $\text{relFromSet}\ \{(e_1,\ e_3) \mid \forall\ (e_1,\ e_2) \in (\text{relToSet}\ r_1)\ (e_2',\ e_3) \in (\text{relToSet}\ r_2) \mid e_2 = e_2'\}$

declare *hol* target_rep function relComp = `'rcomp'`

lemma rel_comp_1 : $(\forall\ r_1\ r_2\ e_1\ e_2\ e_3.\ (\text{inRel}\ e_1\ e_2\ r_1 \wedge \text{inRel}\ e_2\ e_3\ r_2) \longrightarrow \text{inRel}\ e_1\ e_3\ (\text{relComp}\ r_1\ r_2))$
lemma $\sim\{coq;\ ocaml\}$ rel_comp_2 : $(\forall\ r.\ (\text{relComp}\ r\ \text{relId} = r) \wedge (\text{relComp}\ \text{relId}\ r = r))$
lemma rel_comp_3 : $(\forall\ r.\ (\text{relComp}\ r\ \text{relEmpty} = \text{relEmpty}) \wedge (\text{relComp}\ \text{relEmpty}\ r = \text{relEmpty}))$

assert rel_comp_0 : $(\text{relComp}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (4 : \text{NAT}));\ (2,\ 8)\})\ (\text{relFromSet}\ \{(4,\ (3 : \text{NAT}));\ (2,\ 8)\}) =$
$\text{relFromSet}\ \{(2,\ 3)\})$

```
(* ----------------------- *)
(* restrict                *)
(* ----------------------- *)
```

val *relRestrict* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$

let *relRestrict* r s = relFromSet ({ (a, b) | $\forall$ $a \in s$ $b \in s$ | inRel a b r })

declare *hol* target_rep function relRestrict = `'rrestrict'`

assert *rel_restrict*$_0$: (relRestrict (relFromSet {((2 : NAT), (4 : NAT)); (2, 2); (2, 8)}) {2; 8} =
relFromSet {(2, 8); (2, 2)})

lemma *rel_restrict_empty* : ($\forall$ r. relRestrict r {} = relEmpty)
lemma *rel_restrict_rel_empty* : ($\forall$ s. relRestrict relEmpty s = relEmpty)
lemma *rel_restrict_rel_add* : ($\forall$ r x y s. relRestrict (relAdd x y r) s =
if (($x \in s$) $\wedge$ ($y \in s$)) then relAdd x y (relRestrict r s) else relRestrict r s)

```
(* ----------------------- *)
(* Converse                *)
(* ----------------------- *)
```

val *relConverse* : $\forall$ α β. *SetType* α, *SetType* β $\Rightarrow$ REL α β $\rightarrow$ REL β α
let *relConverse* r = relFromSet (Set.map swap (relToSet r))

declare {*hol*} rename function relConverse = lem_converse

assert *rel_converse*$_0$: relConverse (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (4, 5)}) =
relFromSet {(3, 2); (4, 3); (5, 4)}
lemma *rel_converse_empty* : relConverse relEmpty = relEmpty
lemma *rel_converse_add* : $\forall$ x y r. relConverse (relAdd x y r) = relAdd y x (relConverse r)
lemma *rel_converse_converse* : $\forall$ r. relConverse (relConverse r) = r

```
(* ----------------------- *)
(* domain                  *)
(* ----------------------- *)
```

val *relDomain* : $\forall$ α β. *SetType* α, *SetType* β $\Rightarrow$ REL α β $\rightarrow$ SET α
let *relDomain* r = Set.map (fun x $\rightarrow$ fst x) (relToSet r)

declare *hol* target_rep function relDomain = `'domain'`

assert *rel_domain*$_0$: relDomain (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (4, 5)}) = {2; 3; 4}
assert *rel_domain*$_1$: relDomain (relFromSet {((5 : NAT), (3 : NAT)); (3, 4); (4, 5)}) = {3; 4; 5}
assert *rel_domain*$_2$: relDomain (relFromSet {((3 : NAT), (3 : NAT)); (3, 4); (4, 5)}) = {3; 4}

```
(* ----------------------- *)
(* range                   *)
(* ----------------------- *)
```

val *relRange* : $\forall$ α β. *SetType* α, *SetType* β $\Rightarrow$ REL α β $\rightarrow$ SET β
let *relRange* r = Set.map (fun x $\rightarrow$ snd x) (relToSet r)

declare *hol* target_rep function relRange = `'range'`

assert *rel_range*$_0$: relRange (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (4, 5)}) = {3; 4; 5}
assert *rel_range*$_1$: relRange (relFromSet {((5 : NAT), (6 : NAT)); (3, 4); (4, 5)}) = {4; 5; 6}
assert *rel_range*$_2$: relRange (relFromSet {((3 : NAT), (5 : NAT)); (3, 4); (4, 5)}) = {4; 5}

```
(* ----------------------- *)
```

```
(* field / definedOn       *)
(*                         *)
(* avoid the keyword field *)
(* ----------------------- *)
```

val *relDefinedOn* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha$
let inline *relDefinedOn* $r\ =\ ((\text{relDomain}\ r) \cup (\text{relRange}\ r))$

declare $\{hol\}$ rename function relDefinedOn = rdefined_on

assert rel_field_0 : relDefinedOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4);\ (4, 5)\}) = \{2; 3; 4; 5\}$
assert rel_field_1 : relDefinedOn (relFromSet $\{((5 : \text{NAT}),\ (6 : \text{NAT}));\ (3,\ 4);\ (4, 5)\}) = \{3; 4; 5; 6\}$
assert rel_field_2 : relDefinedOn (relFromSet $\{((3 : \text{NAT}),\ (5 : \text{NAT}));\ (3,\ 4);\ (4, 5)\}) = \{3; 4; 5\}$

```
(* ----------------------- *)
(* relOver                 *)
(*                         *)
(* avoid the keyword field *)
(* ----------------------- *)
```

val *relOver* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let *relOver* $r\ s\ =\ ((\text{relDefinedOn}\ r) \subseteq s)$

declare $\{hol\}$ rename function relOver = rel_over

assert rel_over_0 : relOver (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4);\ (4, 5)\})\ \{2; 3; 4; 5\}$
assert rel_over_1 : $\neg$ (relOver (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4);\ (4, 5)\})\ \{3; 4; 5\})$

lemma *rel_over_empty* : $\forall\ s.$ relOver relEmpty s
lemma *rel_over_add* : $\forall\ x\ y\ s\ r.$ relOver (relAdd $x\ y\ r)\ s = (x \in s \wedge y \in s \wedge \text{relOver}\ r\ s)$

```
(* ----------------------- *)
(* apply a relation        *)
(* ----------------------- *)

(* Given a relation r and a set s, relApply r s applies s to r, i.e.
   it returns the set of all value reachable via r from a value in s.
   This operation can be seen as a generalisation of function application. *)
```

val *relApply* : $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \beta$
let *relApply* $r\ s\ =\ \{\ y \mid \forall\ (x,\ y) \in (\text{relToSet}\ r) \mid x \in s\ \}$
declare $\{hol\}$ rename function relApply = rapply

assert rel_apply_0 : relApply (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4);\ (4, 5)\})\ \{2; 3\} = \{3; 4\}$
assert rel_apply_1 : relApply (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 7);\ (3, 5)\})\ \{2; 3\} = \{3; 5; 7\}$

lemma *rel_apply_empty_set* : $\forall\ r.$ relApply $r\ \{\} = \{\}$
lemma *rel_apply_empty* : $\forall\ s.$ relApply relEmpty $s = \{\}$
lemma *rel_apply_add* : $\forall\ x\ y\ s\ r.$ relApply (relAdd $x\ y\ r)\ s =$ (if $(x \in s)$ then (insert y (relApply $r\ s$)) else relApply $r\ s$)

```
(* ========================================================================== *)
(* Properties                                                                 *)
(* ========================================================================== *)
```

```
(* ----------------------- *)
(* subrel                  *)
(* ----------------------- *)
```

val $\mathit{isSubrel}$: $\forall\ \alpha\ \beta.\ \mathit{SetType}\ \alpha,\ \mathit{SetType}\ \beta,\ \mathit{Eq}\ \alpha,\ \mathit{Eq}\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \mathbb{B}$
let inline $\mathit{isSubrel}\ r_1\ r_2\ =\ (\text{relToSet}\ r_1) \subseteq (\text{relToSet}\ r_2)$

lemma $\mathit{is\_subrel\_empty}$: $\forall\ r.\ \text{isSubrel relEmpty}\ r$
lemma $\mathit{is\_subrel\_empty}_2$: $\forall\ r.\ \text{isSubrel}\ r\ \text{relEmpty} = (r = \text{relEmpty})$
lemma $\mathit{is\_subrel\_add}$: $\forall\ x\ y\ r_1\ r_2.\ \text{isSubrel}\ (\text{relAdd}\ x\ y\ r_1)\ r_2 = (\text{inRel}\ x\ y\ r_2 \wedge \text{isSubrel}\ r_1\ r_2)$

assert $\mathit{is\_subrel}_0$: $\text{isSubrel relEmpty}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4);\ (4, 5)\})$
assert $\mathit{is\_subrel}_1$: $\text{isSubrel}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4);\ (4, 5)\})\ (\text{relFromSet}\ \{(2, 3);\ (3,\ 4);\ (4, 5)\})$

assert $\mathit{is\_subrel}_2$: $\text{isSubrel}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (4, 5)\})\ (\text{relFromSet}\ \{(2, 3);\ (3,\ 4);\ (4, 5)\})$

assert $\mathit{is\_subrel}_3$: $\neg\ (\text{isSubrel}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3, 4);\ (4, 5)\})\ (\text{relFromSet}\ \{(2, 3);\ (4, 5)\}))$

```
(* ----------------------- *)
(* reflexivity             *)
(* ----------------------- *)
```

val $\mathit{isReflexiveOn}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isReflexiveOn}\ r\ s\ =\ (\forall\ e \in s.\ \text{inRel}\ e\ e\ r)$

declare $\{\mathit{hol}\}$ rename function isReflexiveOn = lem_is_reflexive_on

val $\mathit{isReflexive}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{\mathit{ocaml};\ \mathit{coq}\}\ \mathit{isReflexive}\ r\ =\ (\forall\ e.\ \text{inRel}\ e\ e\ r)$

declare $\{\mathit{hol}\}$ rename function isReflexive = lem_is_reflexive

assert $\mathit{is\_reflexive\_on}_0$: $\text{isReflexiveOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\})\ \{2; 3\}$

assert $\mathit{is\_reflexive\_on}_1$: $\neg\ (\text{isReflexiveOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\})\ \{2; 4; 3\})$

assert $\mathit{is\_reflexive\_on}_2$: $\neg\ (\text{isReflexiveOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\})\ \{5; 2\})$

```
(* ----------------------- *)
(* irreflexivity           *)
(* ----------------------- *)
```

val $\mathit{isIrreflexiveOn}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isIrreflexiveOn}\ r\ s\ =\ (\forall\ e \in s.\ \neg\ (\text{inRel}\ e\ e\ r))$

declare hol target_rep function isIrreflexiveOn = `'irreflexive'`

val $\mathit{isIrreflexive}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isIrreflexive}\ r\ =\ (\forall\ (e_1,\ e_2) \in (\text{relToSet}\ r).\ \neg\ (e_1 = e_2))$

declare $\{\mathit{hol}\}$ rename function isIrreflexive = lem_is_irreflexive

assert $\mathit{is\_irreflexive\_on}_0$: $\text{isIrreflexiveOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\})\ \{4\}$

assert $\mathit{is\_irreflexive\_on}_1$: $\neg\ (\text{isIrreflexiveOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\})\ \{2; 4\})$

assert $is\_irreflexive\_on_2$: $\neg$ (isIrreflexiveOn (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\}$) $\{5; 2\}$)

assert $is\_irreflexive\_on_3$: isIrreflexiveOn (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\}$) $\{5; 4\}$

assert $is\_irreflexive_0$: $\neg$ (isIrreflexive (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\}$))
assert $is\_irreflexive_1$: isIrreflexive (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4);\ (4, 5)\}$)

```
(* ----------------------- *)
(* symmetry                *)
(* ----------------------- *)
```

val $isSymmetricOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isSymmetricOn\ r\ s\ =\ (\forall\ e_1 \in s\ e_2 \in s.\ (\text{inRel}\ e_1\ e_2\ r) \longrightarrow (\text{inRel}\ e_2\ e_1\ r))$

declare $\{hol\}$ rename function isSymmetricOn = lem_is_symmetric_on

val $isSymmetric$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $isSymmetric\ r\ =\ (\forall\ (e_1,\ e_2) \in \text{relToSet}\ r.\ \text{inRel}\ e_2\ e_1\ r)$

declare $\{hol\}$ rename function isSymmetric = lem_is_symmetric

assert $is\_symmetric\_on_0$: isSymmetricOn (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5);\ (5,\ 4)\}$) $\{4\}$

assert $is\_symmetric\_on_1$: isSymmetricOn (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5);\ (5,\ 4)\}$) $\{3\}$

assert $is\_symmetric\_on_2$: $\neg$ (isSymmetricOn (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5);\ (5,\ 4)\}$) $\{3;\ 4\}$)

assert $is\_symmetric_0$: $\neg$ (isSymmetric (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5)\}$))
assert $is\_symmetric_1$: isSymmetric (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 2);\ (4, 5);\ (5,\ 4)\}$)

lemma $is\_symmetric\_empty$: $\forall\ r.$ isSymmetricOn r $\{\}$
lemma $is\_symmetric\_sing$: $\forall\ r\ x.$ isSymmetricOn r $\{x\}$

```
(* ----------------------- *)
(* antisymmetry            *)
(* ----------------------- *)
```

val $isAntisymmetricOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isAntisymmetricOn\ r\ s\ =\ (\forall\ e_1 \in s\ e_2 \in s.\ (\text{inRel}\ e_1\ e_2\ r) \longrightarrow (\text{inRel}\ e_2\ e_1\ r) \longrightarrow (e_1 = e_2))$

declare $\{hol\}$ rename function isAntisymmetricOn = lem_is_antisymmetric_on

val $isAntisymmetric$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $isAntisymmetric\ r\ =\ (\forall\ (e_1,\ e_2) \in \text{relToSet}\ r.\ (\text{inRel}\ e_2\ e_1\ r) \longrightarrow (e_1 = e_2))$

declare hol target_rep function isAntisymmetric = `'antisym'`

assert $is\_antisymmetric\_on_0$: isAntisymmetricOn (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5);\ (5,\ 4)\}$) $\{3; 4$

assert $is\_antisymmetric\_on_1$: $\neg$ (isAntisymmetricOn (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT}));\ (3,\ 3);\ (3,\ 4);\ (4, 5);\ (5,\ 4)\}$) $\{4;$

assert $is\_antisymmetric_0$: isAntisymmetric (relFromSet {((2 : NAT), (2 : NAT)); (3, 3); (3, 4); (4, 5)})
assert $is\_antisymmetric_1$: ¬ (isAntisymmetric (relFromSet {((2 : NAT), (3 : NAT)); (3, 2); (4, 5); (2, 4)}))

lemma *is_antisymmetric_empty* : ∀ r. isAntisymmetricOn r {}
lemma *is_antisymmetric_sing* : ∀ r x. isAntisymmetricOn r {x}

```
(* ----------------------- *)
(* transitivity            *)
(* ----------------------- *)
```

val *isTransitiveOn* : ∀ α. *SetType* α, *Eq* α ⇒ REL α α → SET α → 𝔹
let *isTransitiveOn* r s = (∀ $e_1 \in s$ $e_2 \in s$ $e_3 \in s$. (inRel e_1 e_2 r) ⟶ (inRel e_2 e_3 r) ⟶ (inRel e_1 e_3 r))

declare {*hol*} rename function isTransitiveOn = lem_transitive_on

val *isTransitive* : ∀ α. *SetType* α, *Eq* α ⇒ REL α α → 𝔹
let *isTransitive* r = (∀ (e_1, e_2) ∈ relToSet r e_3 ∈ relApply r {e_2}. inRel e_1 e_3 r)

declare *hol* target_rep function isTransitive = `'transitive'`

assert $is\_transitive\_on_0$: isTransitiveOn (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (2, 4); (4, 5); (5, 4)}) {2; 3; 4}

assert $is\_transitive\_on_1$: ¬ (isTransitiveOn (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (2, 4); (4, 5); (5, 4)}) {2; 3; 4; 5})

assert $is\_transitive_0$: ¬ (isTransitive (relFromSet {((2 : NAT), (2 : NAT)); (3, 3); (3, 4); (4, 5)}))
assert $is\_transitive_1$: isTransitive (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (2, 4) })

```
(* ----------------------- *)
(* total                   *)
(* ----------------------- *)
```

val *isTotalOn* : ∀ α. *SetType* α, *Eq* α ⇒ REL α α → SET α → 𝔹
let *isTotalOn* r s = (∀ $e_1 \in s$ $e_2 \in s$. (inRel e_1 e_2 r) ∨ (inRel e_2 e_1 r))

declare {*hol*} rename function isTotalOn = lem_is_total_on

val *isTotal* : ∀ α. *SetType* α, *Eq* α ⇒ REL α α → 𝔹
let ∼{*ocaml*; *coq*} *isTotal* r = (∀ e_1 e_2. (inRel e_1 e_2 r) ∨ (inRel e_2 e_1 r))
declare {*hol*} rename function isTotal = lem_is_total

val *isTrichotomousOn* : ∀ α. *SetType* α, *Eq* α ⇒ REL α α → SET α → 𝔹
let *isTrichotomousOn* r s = (∀ $e_1 \in s$ $e_2 \in s$. (inRel e_1 e_2 r) ∨ ($e_1 = e_2$) ∨ (inRel e_2 e_1 r))

declare {*hol*} rename function isTrichotomousOn = lem_is_trichotomous_on

val *isTrichotomous* : ∀ α. *SetType* α, *Eq* α ⇒ REL α α → 𝔹
let ∼{*ocaml*; *coq*} *isTrichotomous* r = (∀ e_1 e_2. (inRel e_1 e_2 r) ∨ ($e_1 = e_2$) ∨ (inRel e_2 e_1 r))

declare {*hol*} rename function isTrichotomous = lem_is_trichotomous

assert $is\_total\_on_0$: isTotalOn (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (3, 3); (4, 4)}) {3; 4}
assert $is\_total\_on_1$: ¬ (isTotalOn (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (3, 3); (4, 4)}) {2; 4})

assert $is\_trichotomous\_on_0$: isTrichotomousOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4)\}$) $\{3; 4\}$
assert $is\_trichotomous\_on_1$: $\neg$ (isTrichotomousOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4)\}$) $\{2; 3; 4\}$)

```
(* ---------------------- *)
(* is_single_valued       *)
(* ---------------------- *)
```

val $isSingleValued$: $\forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \mathbb{B}$
let $isSingleValued\ r\ =\ (\forall\ (e_1,\ e2a) \in \text{relToSet}\ r\ e2b \in \text{relApply}\ r\ \{e_1\}.\ e2a = e2b)$

declare $\{hol\}$ rename function isSingleValued = lem_is_single_valued

assert $is\_single\_valued_0$: isSingleValued (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 4)\}$)
assert $is\_single\_valued_1$: $\neg$ (isSingleValued (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (2, 4)\ ;\ (3,\ 4)\}$))

```
(* ---------------------- *)
(* equivalence relation   *)
(* ---------------------- *)
```

val $isEquivalenceOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isEquivalenceOn\ r\ s$ = isReflexiveOn $r\ s$ $\wedge$ isSymmetricOn $r\ s$ $\wedge$ isTransitiveOn $r\ s$

declare $\{hol\}$ rename function isEquivalenceOn = lem_is_equivalence_on

val $isEquivalence$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml;\ coq\}$ $isEquivalence\ r$ = isReflexive r $\wedge$ isSymmetric r $\wedge$ isTransitive r

declare $\{hol\}$ rename function isEquivalence = lem_is_equivalence

assert $is\_equivalence_0$: isEquivalenceOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 2);\ (2, 2);\ (3, 3);\ (4, 4)\}$) $\{2; 3; 4\}$

assert $is\_equivalence_1$: $\neg$ (isEquivalenceOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 2);\ (2,\ 4);\ (2, 2);\ (3, 3);\ (4, 4)\}$) $\{2; 3; 4$

assert $is\_equivalence_2$: $\neg$ (isEquivalenceOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3,\ 2);\ (2, 2);\ (3, 3);\ \}$) $\{2; 3; 4\}$)

```
(* ---------------------- *)
(* well founded           *)
(* ---------------------- *)
```

val $isWellFounded$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml;\ coq\}$ $isWellFounded\ r\ =\ (\forall\ P.\ (\forall\ x.\ (\forall\ y.\ \text{inRel}\ y\ x\ r \longrightarrow P\ x) \longrightarrow P\ x) \longrightarrow (\forall\ x.\ P\ x))$

declare hol target_rep function isWellFounded r = `'WF'` (`'reln_to_rel'` r)

```
(* ========================================================================== *)
(* Orders                                                                     *)
(* ========================================================================== *)
```

```
(* ----------------------- *)
(* pre- or quasiorders     *)
(* ----------------------- *)
```

val $\mathit{isPreorderOn}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \textsc{rel}\ \alpha\ \alpha\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isPreorderOn}\ r\ s\ =\ \mathrm{isReflexiveOn}\ r\ s \wedge \mathrm{isTransitiveOn}\ r\ s$

declare $\{\mathit{hol}\}$ rename function isPreorderOn = lem_is_preorder_on

val $\mathit{isPreorder}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \textsc{rel}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{\mathit{ocaml};\ \mathit{coq}\}\ \mathit{isPreorder}\ r\ =\ \mathrm{isReflexive}\ r \wedge \mathrm{isTransitive}\ r$

declare $\{\mathit{hol}\}$ rename function isPreorder = lem_is_preorder

assert $\mathit{is\_preorder}_0$: $\mathrm{isPreorderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (3,2);\ (2,2);\ (3,3);\ (4,4)\})\ \{2;3;4\}$

assert $\mathit{is\_preorder}_1$: $\neg\,(\mathrm{isPreorderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (2,2);\ (3,3)\})\ \{2;3;4\})$
assert $\mathit{is\_preorder}_2$: $\neg\,(\mathrm{isPreorderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (3,4);\ (2,2);\ (3,3);\ (4,4)\})\ \{2;3;4\})$

```
(* ----------------------- *)
(* partial orders          *)
(* ----------------------- *)
```

val $\mathit{isPartialOrderOn}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \textsc{rel}\ \alpha\ \alpha\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isPartialOrderOn}\ r\ s\ =\ \mathrm{isReflexiveOn}\ r\ s \wedge \mathrm{isTransitiveOn}\ r\ s \wedge \mathrm{isAntisymmetricOn}\ r\ s$

declare $\{\mathit{hol}\}$ rename function isPartialOrderOn = lem_is_partial_order_on

assert $\mathit{is\_partialorder}_0$: $\mathrm{isPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (2,2);\ (3,3);\ (4,4)\})\ \{2;3;4\}$

assert $\mathit{is\_partialorder}_1$: $\neg\,(\mathrm{isPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (3,2);\ (2,2);\ (3,3);\ (4,4)\})\ \{2;3;4\})$

assert $\mathit{is\_partialorder}_2$: $\neg\,(\mathrm{isPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (2,2);\ (3,3)\})\ \{2;3;4\})$
assert $\mathit{is\_partialorder}_3$: $\neg\,(\mathrm{isPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (3,4);\ (2,2);\ (3,3);\ (4,4)\})\ \{2;3;4\})$

val $\mathit{isStrictPartialOrderOn}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \textsc{rel}\ \alpha\ \alpha\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isStrictPartialOrderOn}\ r\ s\ =\ \mathrm{isIrreflexiveOn}\ r\ s \wedge \mathrm{isTransitiveOn}\ r\ s$

declare $\{\mathit{hol}\}$ rename function isStrictPartialOrderOn = lem_is_strict_partial_order_on

lemma $\mathit{isStrictPartialOrderOn\_antisym}$: $(\forall\ r\ s.\ \mathrm{isStrictPartialOrderOn}\ r\ s \longrightarrow \mathrm{isAntisymmetricOn}\ r\ s)$

assert $\mathit{is\_strict\_partialorder\_on}_0$: $\mathrm{isStrictPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}))\})\ \{2;3;4\}$
assert $\mathit{is\_strict\_partialorder\_on}_1$: $\mathrm{isStrictPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (3,4);\ (2,4)\})\ \{2;3;4\}$

assert $\mathit{is\_strict\_partialorder\_on}_2$: $\neg\,(\mathrm{isStrictPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (3,4)\})\ \{2;3;4\})$

assert $\mathit{is\_strict\_partialorder\_on}_3$: $\neg\,(\mathrm{isStrictPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (3,2)\})\ \{2;3;4\})$

assert $\mathit{is\_strict\_partialorder\_on}_4$: $\neg\,(\mathrm{isStrictPartialOrderOn}\ (\mathrm{relFromSet}\ \{((2:\textsc{nat}),\ (3:\textsc{nat}));\ (2,2)\})\ \{2;3;4\})$

val $\mathit{isStrictPartialOrder}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \textsc{rel}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$

let $\mathit{isStrictPartialOrder}\ r\ =\ \text{isIrreflexive}\ r \wedge \text{isTransitive}\ r$

declare $\{\mathit{hol}\}$ rename function $\text{isStrictPartialOrder}\ =\ \text{lem\_is\_strict\_partial\_order}$

assert $\mathit{is\_strict\_partialorder}_0\ :\ \text{isStrictPartialOrder}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}))\})$
assert $\mathit{is\_strict\_partialorder}_1\ :\ \text{isStrictPartialOrder}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3, 4);\ (2, 4)\})$
assert $\mathit{is\_strict\_partialorder}_2\ :\ \neg\ (\text{isStrictPartialOrder}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3, 4)\}))$
assert $\mathit{is\_strict\_partialorder}_3\ :\ \neg\ (\text{isStrictPartialOrder}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (3, 2)\}))$
assert $\mathit{is\_strict\_partialorder}_4\ :\ \neg\ (\text{isStrictPartialOrder}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (2, 2)\}))$

val $\mathit{isPartialOrder}\ :\ \forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{\mathit{ocaml};\ \mathit{coq}\}\ \mathit{isPartialOrder}\ r\ =\ \text{isReflexive}\ r \wedge \text{isTransitive}\ r \wedge \text{isAntisymmetric}\ r$

declare $\{\mathit{hol}\}$ rename function $\text{isPartialOrder}\ =\ \text{lem\_is\_partial\_order}$

```
(* ----------------------- *)
(* total / linear orders   *)
(* ----------------------- *)
```

val $\mathit{isTotalOrderOn}\ :\ \forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isTotalOrderOn}\ r\ s\ =\ \text{isPartialOrderOn}\ r\ s \wedge \text{isTotalOn}\ r\ s$

declare $\{\mathit{hol}\}$ rename function $\text{isTotalOrderOn}\ =\ \text{lem\_is\_total\_order\_on}$

val $\mathit{isStrictTotalOrderOn}\ :\ \forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isStrictTotalOrderOn}\ r\ s\ =\ \text{isStrictPartialOrderOn}\ r\ s \wedge \text{isTrichotomousOn}\ r\ s$

declare $\{\mathit{hol}\}$ rename function $\text{isStrictTotalOrderOn}\ =\ \text{lem\_is\_strict\_total\_order\_on}$

val $\mathit{isTotalOrder}\ :\ \forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{\mathit{ocaml};\ \mathit{coq}\}\ \mathit{isTotalOrder}\ r\ =\ \text{isPartialOrder}\ r \wedge \text{isTotal}\ r$

declare $\{\mathit{hol}\}$ rename function $\text{isTotalOrder}\ =\ \text{lem\_is\_total\_order}$

val $\mathit{isStrictTotalOrder}\ :\ \forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{\mathit{ocaml};\ \mathit{coq}\}\ \mathit{isStrictTotalOrder}\ r\ =\ \text{isStrictPartialOrder}\ r \wedge \text{isTrichotomous}\ r$

declare $\{\mathit{hol}\}$ rename function $\text{isStrictTotalOrder}\ =\ \text{lem\_is\_strict\_total\_order}$

assert $\mathit{is\_totalorder\_on}_0\ :\ \text{isTotalOrderOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (2, 2);\ (3, 3);\ (4, 4)\})\ \{2;\ 3\}$
assert $\mathit{is\_totalorder\_on}_1\ :\ \neg\ (\text{isTotalOrderOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}));\ (2, 2);\ (3, 3);\ (4, 4)\})\ \{2;\ 3;\ 4\})$

assert $\mathit{is\_totalorder\_on}_2\ :\ \neg\ (\text{isTotalOrderOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}))\})\ \{2;\ 3\})$

assert $\mathit{is\_strict\_totalorder\_on}_0\ :\ \text{isStrictTotalOrderOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}))\})\ \{2;\ 3\}$
assert $\mathit{is\_strict\_totalorder\_on}_1\ :\ \neg\ (\text{isStrictTotalOrderOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT}))\})\ \{2;\ 3;\ 4\})$

```
(* ========================================================================== *)
(* closures                                                                   *)
(* ========================================================================== *)

(* ----------------------- *)
(* transitive closure      *)
(* ----------------------- *)
```

val *transitiveClosure* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow$ REL $\alpha\ \alpha \rightarrow$ REL $\alpha\ \alpha$
val *transitiveClosureByEq* : $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow$ REL $\alpha\ \alpha \rightarrow$ REL $\alpha\ \alpha$
val *transitiveClosureByCmp* : $\forall\ \alpha.\ (\alpha * \alpha \rightarrow \alpha * \alpha \rightarrow$ ORDERING$) \rightarrow$ REL $\alpha\ \alpha \rightarrow$ REL $\alpha\ \alpha$

declare *ocaml* target_rep function transitiveClosureByCmp = `'Pset.tc'`
declare *hol* target_rep function transitiveClosure = `'tc'`
declare *isabelle* target_rep function transitiveClosure = `'trancl'`
declare *coq* target_rep function transitiveClosureByEq = `'set_tc'`

let inline {*coq*} *transitiveClosure* = transitiveClosureByEq (=)
let inline {*ocaml*} *transitiveClosure* = transitiveClosureByCmp setElemCompare

lemma *transitiveClosure_spec*$_1$: $(\forall\ r.$ isSubrel r (transitiveClosure r))
lemma *transitiveClosure_spec*$_2$: $(\forall\ r.$ isTransitive (transitiveClosure r))
lemma *transitiveClosure_spec*$_3$: $(\forall\ r_1\ r_2.$ ((isTransitive r_2) $\wedge$ (isSubrel $r_1\ r_2$)) $\longrightarrow$ isSubrel (transitiveClosure r_1) r_2)

lemma *transitiveClosure_spec*$_4$: $(\forall\ r.$ isTransitive $r \longrightarrow$ (transitiveClosure $r = r$))

assert *transitive_closure*$_0$: (transitiveClosure (relFromSet {((2 : NAT), (3 : NAT)); (3, 4)}) =
relFromSet {(2, 3); (2, 4); (3, 4)})
assert *transitive_closure*$_1$: (transitiveClosure (relFromSet {((2 : NAT), (3 : NAT)); (3, 4); (4, 5); (7, 9)}) =
relFromSet {(2, 3); (2, 4); (2, 5); (3, 4); (3, 5); (4, 5); (7, 9)})

```
(* ----------------------- *)
(* transitive closure step *)
(* ----------------------- *)
```

val *transitiveClosureAdd* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \alpha \rightarrow \alpha \rightarrow$ REL $\alpha\ \alpha \rightarrow$ REL $\alpha\ \alpha$

let *transitiveClosureAdd* $x\ y\ r$ =
(relUnion (relAdd $x\ y\ r$) (relUnion (relFromSet $\{(x,\ z) \mid \forall\ z \in$ relRange $r \mid$ inRel $y\ z\ r\}$)
(relFromSet $\{(z,\ y) \mid \forall\ z \in$ relDomain $r \mid$ inRel $z\ x\ r\}$)))

declare {*hol*} rename function transitiveClosureAdd = tc_insert

lemma *transitive_closure_add_thm* : $\forall\ x\ y\ r.$ isTransitive $r \longrightarrow$ (transitiveClosureAdd $x\ y\ r$ = transitiveClosure (relAdd $x\ y\ r$))

assert *transitive_closure_add*$_0$: transitiveClosureAdd (2 : NAT) (3 : NAT) {} = relFromSet {(2, 3)}
assert *transitive_closure_add*$_1$: transitiveClosureAdd (3 : NAT) (4 : NAT) {(2, 3)} = relFromSet {(2, 3); (3, 4); (2, 4)}

assert *transitive_closure_add*$_2$: transitiveClosureAdd (4 : NAT) (5 : NAT) {(2, 3); (3, 4); (2, 4)} =
relFromSet {(2, 3); (3, 4); (2, 4); (4, 5); (2, 5); (3, 5)}

```
(* ===================================================================== *)
(* reflexiv closures                                                     *)
(* ===================================================================== *)
```

val *reflexivTransitiveClosureOn* : $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow$ REL $\alpha\ \alpha \rightarrow$ SET $\alpha \rightarrow$ REL $\alpha\ \alpha$
let *reflexivTransitiveClosureOn* $r\ s$ = transitiveClosure (relUnion r (relIdOn s))
declare {*hol*} rename function reflexivTransitiveClosureOn = reflexiv_transitive_closure_on

assert *reflexiv_transitive_closure*$_0$: (reflexivTransitiveClosureOn (relFromSet {((2 : NAT), (3 : NAT)); (3, 4)}) {2; 3; 4} =
relFromSet {(2, 3); (2, 4); (3, 4); (2, 2); (3, 3); (4, 4)})

val $reflexivTransitiveClosure\ :\ \forall\ \alpha.\ SetType\ \alpha,\ \ Eq\ \alpha\ \Rightarrow\ \textsc{rel}\ \alpha\ \alpha\ \rightarrow\ \textsc{rel}\ \alpha\ \alpha$
let $\sim\{ocaml;\ coq\}\ reflexivTransitiveClosure\ r\ =\ \mathrm{transitiveClosure}\ (\mathrm{relUnion}\ r\ \mathrm{relId})$

17 Sorting

```
(**************************************************************************)
(* A library for sorting lists                                            *)
(*                                                                        *)
(* It mainly follows the Haskell List-library                             *)
(**************************************************************************)
```

```
(* ====================================================================== *)
(* Header                                                                 *)
(* ====================================================================== *)
```

declare $\{isabelle;\ hol;\ ocaml\}$ rename module $=$ lem_sorting

open import *Bool Basic_classes Maybe List Num*

open import $\{isabelle\}$ $\sim\sim/src/HOL/Library/Permutation$
open import $\{coq\}$ *Coq.Lists.TheoryList*
open import $\{hol\}$ *sortingTheory permLib*
open import $\{isabelle\}$ *\$LIB_DIR/Lem*

```
(* ------------------------- *)
(* permutations              *)
(* ------------------------- *)
```

val *isPermutation* : $\forall\ \alpha.\ Eq\ \alpha\ \Rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \mathbb{B}$
val *isPermutationBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \mathbb{B}$

let rec *isPermutationBy* $eq\ l_1\ l_2$ $=$ match l_1 with
  $|$ [] $\rightarrow$ null l_2
  $|$ $(x\ ::\ xs)$ $\rightarrow$ begin
      match deleteFirst $(eq\ x)$ l_2 with
        $|$ Nothing $\rightarrow$ false
        $|$ Just ys $\rightarrow$ isPermutationBy $eq\ xs\ ys$
      end
    end
end
declare termination_argument isPermutationBy $=$ automatic
declare $\{hol\}$ rename function isPermutationBy $=$ PERM_BY

let inline *isPermutation* $=$ isPermutationBy (=)

declare *isabelle* target_rep function isPermutation $=$ infix '<~~>'
declare *hol* target_rep function isPermutation $=$ 'PERM'

assert $perm_1$: (isPermutation ([] : LIST NAT) [])
assert $perm_2$: ($\neg$ (isPermutation [(2 : NAT)] []))
assert $perm_3$: (isPermutation [(2 : NAT); 1; 3; 5; 4] [1; 2; 3; 4; 5])
assert $perm_4$: ($\neg$ (isPermutation [(2 : NAT); 3; 3; 5; 4] [1; 2; 3; 4; 5]))
assert $perm_5$: ($\neg$ (isPermutation [(2 : NAT); 1; 3; 5; 4; 3] [1; 2; 3; 4; 5]))
assert $perm_6$: (isPermutation [(2 : NAT); 1; 3; 5; 4; 3] [1; 2; 3; 3; 4; 5])

lemma $isPermutation_1$: $(\forall\ l.\ \text{isPermutation}\ l\ l)$
lemma $isPermutation_2$: $(\forall\ l_1\ l_2.\ \text{isPermutation}\ l_1\ l_2 \longleftrightarrow \text{isPermutation}\ l_2\ l_1)$
lemma $isPermutation_3$: $(\forall\ l_1\ l_2\ l_3.\ \text{isPermutation}\ l_1\ l_2 \longrightarrow \text{isPermutation}\ l_2\ l_3 \longrightarrow \text{isPermutation}\ l_1\ l_3)$
lemma $isPermutation_4$: $(\forall\ l_1\ l_2.\ \text{isPermutation}\ l_1\ l_2 \longrightarrow (\text{length}\ l_1 = \text{length}\ l_2))$

lemma $isPermutation_5$: $(\forall\ l_1\ l_2.\ \text{isPermutation}\ l_1\ l_2 \longrightarrow (\forall\ x.\ \text{elem}\ x\ l_1 = \text{elem}\ x\ l_2))$

```
(* ------------------------- *)
(* isSorted                  *)
(* ------------------------- *)

(* isSortedBy R l
   checks, whether the list l is sorted by ordering R.
   R should represent an order, i.e. it should be transitive.
   Different backends defined "isSorted" slightly differently. However,
   the definitions coincide for transitive R. Therefore there is the
   following restriction:

   WARNING: Use isSorted and isSortedBy only with transitive relations!
*)
```

val $isSorted$: $\forall\ \alpha.\ Ord\ \alpha \Rightarrow \text{LIST}\ \alpha \rightarrow \mathbb{B}$
val $isSortedBy$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \text{LIST}\ \alpha \rightarrow \mathbb{B}$

```
(* DPM: rejigged the definition with a nested match to get past Coq's termination checker.
*)
```

let rec $isSortedBy\ cmp\ l$ = match l with
  | [] → true
  | $x_1 :: xs$ →
    match xs with
      | [] → true
      | $x_2 :: \_$ → $(cmp\ x_1\ x_2 \wedge \text{isSortedBy}\ cmp\ xs)$
    end
end
declare termination_argument isSortedBy = automatic

let inline $isSorted$ = isSortedBy $(\leq)$

declare $isabelle$ target_rep function isSortedBy = `'sorted_by'`
declare hol target_rep function isSortedBy = `'SORTED'`

assert $isSorted_1$: (isSorted ([] : LIST NAT))
assert $isSorted_2$: (isSorted [(2 : NAT)])
assert $isSorted_3$: (isSorted [(2 : NAT); 4; 5])
assert $isSorted_4$: (isSorted [(1 : NAT); 2; 2; 4; 4; 8])
assert $isSorted_5$: (¬ (isSorted [(3 : NAT); 2]))
assert $isSorted_6$: (¬ (isSorted [(1 : NAT); 2; 3; 2; 3; 4; 5]))

```
(* ----------------------- *)
(* insertion sort          *)
(* ----------------------- *)
```

val $insert$: $\forall\ \alpha.\ Ord\ \alpha \Rightarrow \alpha \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$
val $insertBy$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \alpha \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$

val $insertSort$: $\forall\ \alpha.\ Ord\ \alpha \Rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$
val $insertSortBy$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$

let rec $insertBy\ cmp\ e\ l$ = match l with
  | [] → $[e]$

| x :: xs $\rightarrow$ if $cmp\ x\ e$ then x :: (insertBy $cmp\ e\ xs$) else (e :: x :: xs)
end
declare termination_argument insertBy = automatic

let inline $insert$ = insertBy ($\leq$)

let $insertSortBy\ cmp\ l$ = List.foldl (fun $l\ e$ $\rightarrow$ insertBy $cmp\ e\ l$) [] l
let inline $insertSort$ = insertSortBy ($\leq$)

declare $isabelle$ target_rep function insertBy = `'insert_sort_insert_by'`
declare $isabelle$ target_rep function insertSortBy = `'insert_sort_by'`

declare $\{hol\}$ rename function insertBy = INSERT_SORT_INSERT
declare $\{hol\}$ rename function insertSortBy = INSERT_SORT

lemma $insertBy_1$: $(\forall\ l\ e\ cmp.\ ((\forall\ x\ y\ z.\ cmp\ x\ y \land cmp\ y\ z \longrightarrow cmp\ x\ z) \land \text{isSortedBy}\ cmp\ l) \longrightarrow \text{isSortedBy}\ cmp\ (\text{insertBy}\ cm$

lemma $insertBy_2$: $(\forall\ l\ e\ cmp.\ \text{length}\ (\text{insertBy}\ cmp\ e\ l) = \text{length}\ l + 1)$
lemma $insertBy_3$: $(\forall\ l\ e_1\ e_2\ cmp.\ \text{elem}\ e_1\ (\text{insertBy}\ cmp\ e_2\ l) = ((e_1 = e_2) \lor \text{elem}\ e_1\ l))$

lemma $insertSort_1$: $(\forall\ l\ cmp.\ \text{isPermutation}\ (\text{insertSort}\ l)\ l)$
lemma $insertSort_2$: $(\forall\ l\ cmp.\ \text{isSorted}\ (\text{insertSort}\ l))$

```
(* ----------------------- *)
(* general sorting         *)
(* ----------------------- *)
```

val $sort$: $\forall\ \alpha.\ Ord\ \alpha \Rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$
val $sortBy$: $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$

let inline $sortBy$ = insertSortBy
let inline $sort$ = sortBy ($\leq$)

declare $isabelle$ target_rep function sortBy = `'sort_by'`
declare hol target_rep function sortBy = `'QSORT'`

assert $sort_1$: $(\text{sort}\ ([]\ :\ \text{LIST NAT}) = [])$
assert $sort_2$: $(\text{sort}\ ([6;4;3;8;1;2]\ :\ \text{LIST NAT}) = [1;2;3;4;6;8])$
assert $sort_3$: $(\text{sort}\ ([5;4;5;2;4]\ :\ \text{LIST NAT}) = [2;4;4;5;5])$

lemma $sort_4$: $(\forall\ l\ cmp.\ \text{isPermutation}\ (\text{sort}\ l)\ l)$
lemma $sort_5$: $(\forall\ l\ cmp.\ \text{isSorted}\ (\text{sort}\ l))$

18 Pervasives

declare $\{isabelle; ocaml; hol\}$ rename module $=$ Lem_pervasives

include import *Basic_classes Bool Tuple Maybe Either Function Num Map Set List*

import *Sorting Relation*

19 Set_extra

```
(******************************************************************************)
(* A library for sets                                                         *)
(*                                                                            *)
(* It mainly follows the Haskell Set-library                                  *)
(******************************************************************************)

(* ========================================================================== *)
(* Header                                                                     *)
(* ========================================================================== *)
```

open import *Bool Basic_classes Maybe Function Num List Sorting Set*

declare $\{hol;\ isabelle;\ ocaml\}$ rename module $=$ lem_set_extra

```
(* ----------------------------*)
(* set choose (be careful !)   *)
(* --------------------------- *)
```

val *choose* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \alpha$
declare compile_message choose $=$ "$chooseisnon-deterministicandonlydefinedfornon-emptysets.It'sresultmaydifferbetwe$
$levelrepresentationofsetsandbedifferentfortworepresentationsofthesameset.$"

declare *hol* target_rep function choose $=$ `'CHOICE'`
declare *isabelle* target_rep function choose $=$ `'set_choose'`
declare *ocaml* target_rep function choose $=$ `'Pset.choose'`

lemma $\sim\{coq\}$ *choose_sing* : $(\forall\ x.\ \text{choose}\ \{x\} = x)$
lemma $\sim\{coq\}$ *choose_in* : $(\forall\ s.\ \neg\ (\text{null}\ s) \longrightarrow ((\text{choose}\ s) \in s))$

assert $\sim\{coq\}$ $choose_0$: choose $\{(2 : \textsc{nat})\} = 2$
assert $\sim\{coq\}$ $choose_1$: choose $\{(5 : \textsc{nat})\} = 5$
assert $\sim\{coq\}$ $choose_2$: choose $\{(6 : \textsc{nat})\} = 6$
assert $\sim\{coq\}$ $choose_3$: choose $\{(6 : \textsc{nat});\ 1;\ 2\} \in \{6;\ 1;\ 2\}$

```
(* ----------------------------*)
(* universal set               *)
(* --------------------------- *)
```

val *universal* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \textsc{set}\ \alpha$
declare compile_message universal $=$ "$universalsetsareusuallyinfiniteandonlyavailableinHOLandIsabelle$"

let $\{hol;\ isabelle\}$ *universal* $=$ $\{\ x \mid \forall\ x \mid \text{true}\ \}$

declare *hol* target_rep function universal $=$ `'UNIV'`

assert $\{hol\}$ in_univ_0 : true $\in$ universal
assert $\{hol\}$ in_univ_1 : $(1 : \textsc{nat}) \in$ universal
lemma $\{hol\}$ *in_univ_thm* : $\forall\ x.\ x \in$ universal

```
(* ----------------------------*)
(* toList                      *)
(* --------------------------- *)
```

val *toList* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \textsc{list}\ \alpha$
declare compile_message toList = "$toListisonlydefinedonfinitesetsandtheorderoftheresultinglistisunspecifiedandtherefor$

declare *ocaml* target_rep function toList = 'Pset.elements'
declare *isabelle* target_rep function toList = 'list_of_set'
declare *hol* target_rep function toList = 'SET_TO_LIST'
declare *coq* target_rep function toList = 'set_to_list'

assert $toList_0$: toList $(\{\}\ :\ \textsc{set nat}) = []$
assert $toList_1$: toList $\{(6 : \textsc{nat});\ 1;\ 2\} \in \{[1;2;6];\ [1;6;2];\ [2;1;6];\ [2;6;1];\ [6;1;2];\ [6;2;1]\}$
assert $toList_2$: toList $(\{(2 : \textsc{nat})\}\ :\ \textsc{set nat}) = [2]$

```
(* ----------------------------*)
(* toOrderedList               *)
(* --------------------------- *)

(* "toOrderedList" returns a sorted list. Therefore the result is (given a suitable order)
deterministic.
   Therefore, it is much preferred to "toList". However, it still is only defined for finite
sets. So, please
   use carefully and consider using set-operations instead of translating sets to lists, performing
list manipulations
   and then transforming back to sets. *)
```

val *toOrderedListBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \textsc{list}\ \alpha$
declare *isabelle* target_rep function toOrderedListBy = 'ordered_list_of_set'

val *toOrderedList* : $\forall\ \alpha.\ SetType\ \alpha,\ Ord\ \alpha\ \Rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \textsc{list}\ \alpha$
let inline $\sim\{isabelle;\ ocaml\}$ *toOrderedList* l = sort (toList l)
let inline $\{isabelle\}$ *toOrderedList* = toOrderedListBy $(\leq)$
declare *ocaml* target_rep function toOrderedList = 'Pset.elements'

declare compile_message toOrderedList = "$toListisonlydefinedonfinitesets.Evenworse,itreturnstheelementsinanunspecifi$
$levelrepresentation.Thesamesetmayhaveseverallow-levelrepresentationsthatmightleadtodifferentresultsfortoList.$"

assert $toOrderedList_0$: toOrderedList $(\{\}\ :\ \textsc{set nat}) = []$
assert $toOrderedList_1$: toOrderedList $\{(6 : \textsc{nat});\ 1;\ 2\} = [1;2;6]$
assert $toOrderedList_2$: toOrderedList $(\{(2 : \textsc{nat})\}\ :\ \textsc{set nat}) = [2]$

```
(* ----------------------------*)
(* unbounded fixed point       *)
(* --------------------------- *)

(* Is NOT supported by the coq backend! *)
```

val *leastFixedPointUnbounded* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ (\textsc{set}\ \alpha\ \rightarrow\ \textsc{set}\ \alpha)\ \rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \textsc{set}\ \alpha$
let rec *leastFixedPointUnbounded* $f\ x$ =
  let $fx = f\ x$ in
  if $fx \subseteq x$ then x
  else leastFixedPointUnbounded $f\ (fx \cup x)$

declare compile_message toOrderedList = "$leastFixedPointUnboundedisdeprecatedasitisnotsupportedbyallbackends(e.g.coq).$

assert *lfp_empty* : leastFixedPointUnbounded (map (fun $x \rightarrow x$)) $(\{\}\ :\ \textsc{set nat}) = \{\}$

assert $\mathit{lfp\_saturate\_neg}$: $\mathrm{leastFixedPointUnbounded}\ (\mathrm{map}\ (\mathsf{fun}\ x\ \rightarrow\ -x))\ (\{1;\ 2;\ 3\}\ :\ \textsc{set int}) = \{-3;\ -2;\ -1;\ 1;\ 2;\ 3\}$

assert $\mathit{lfp\_saturate\_mod}$: $\mathrm{leastFixedPointUnbounded}\ (\mathrm{map}\ (\mathsf{fun}\ x\ \rightarrow\ (2*x)\ \mathrm{mod}\ 5))\ (\{1\}\ :\ \textsc{set nat}) = \{1;\ 2;\ 3;\ 4\}$

20 String_extra

```
(**************************************************************************)
(* String functions                                                       *)
(**************************************************************************)
```

open import *Basic_classes*
open import $\{hol\}$ *stringLib*

declare $\{isabelle;\ ocaml;\ hol\}$ rename module $=$ lem_string_extra

val *stringCompare* : STRING $\rightarrow$ STRING $\rightarrow$ ORDERING

`(* TODO: *)`
let inline *stringCompare* $x\ y\ =\ \mathrm{EQ}$
let inline $\{ocaml\}$ *stringCompare* $=$ defaultCompare

declare compile_message stringCompare $=$ "$Itishighlyunclear, whatstringcomparisonshoulddo.Dowehaveabc<ABC<bbcorabc$

let *stringLess* $x\ y\ =$ orderingIsLess (stringCompare $x\ y$)
let *stringLessEq* $x\ y\ =$ orderingIsLessEqual (stringCompare $x\ y$)
let *stringGreater* $x\ y\ =$ stringLess $y\ x$
let *stringGreaterEq* $x\ y\ =$ stringLessEq $y\ x$

instance (*Ord* STRING)
  let *compare* $=$ stringCompare
  let $<\ =$ stringLess
  let $<=\ =$ stringLessEq
  let $>\ =$ stringGreater
  let $>=\ =$ stringGreaterEq
end

assert $\{ocaml\}$ $string\_compare_1$: "abc" $<$ "bbc"
assert $\{ocaml\}$ $string\_compare_2$: "abc" $\leq$ "abc"
assert $\{ocaml\}$ $string\_compare_3$: "abc" $>$ "ab"

21 Pervasives_extra

declare {*isabelle*; *ocaml*; *hol*} rename module = Lem_pervasives_extra

include import *Pervasives*
include import *Function_extra Maybe_extra Map_extra Set_extra Set_helpers List_extra String_extra*