Contents

1 Bool

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

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

declare {isabelle, hol, ocaml, coq} 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 not_1 : ¬ (¬ true)
assert not_2 : ¬ false

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

val && [and] : 𝔹 → 𝔹 → 𝔹
let && b_1 b_2 = match (b_1, b_2) 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 and_1 : (¬ (true ∧ false))
assert and_2 : (¬ (false ∧ true))
assert and_3 : (¬ (false ∧ false))
assert and_4 : (true ∧ true)


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

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

val || [or] : 𝔹 → 𝔹 → 𝔹
let || b1 b2 = match (b1, b2) with
  | (false, false) → false
  | _ → 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 '&or;'
declare tex target_rep function or = infix '$\vee$'

assert or1 : (true ∨ false)
assert or2 : (false ∨ true)
assert or3 : (true ∨ true)
assert or4 : (¬ (false ∨ false))


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

val − − > [imp] : 𝔹 → 𝔹 → 𝔹
let − − > b1 b2 = match (b1, b2) with
  | (true, false) → false
  | _ → 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 '&rarr;'
declare tex target_rep function imp = infix '$\longrightarrow$'

let inline {ocaml, coq} imp x y = ((¬ x) ∨ y)

assert imp1 : (¬ (true ⟶ false))
assert imp2 : (false ⟶ true)
assert imp3 : (false ⟶ false)
assert imp4 : (true ⟶ true)


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

val < − > [equiv] : 𝔹 → 𝔹 → 𝔹
let < − > b1 b2 = match (b1, b2) with
  | (true, true) → true
  | (false, false) → true
  | _ → 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 $equiv_1$: (¬ (true ⟷ false))
assert $equiv_2$: (¬ (false ⟷ true))
assert $equiv_3$: (false ⟷ false)
assert $equiv_4$: (true ⟷ true)

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

val xor : $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$
let inline $xor\ b_1\ b_2 = \neg\ (b_1 \longleftrightarrow b_2)$

assert xor_1 : (xor true false)
assert xor_2 : (xor false true)
assert xor_3 : (¬ (xor true true))
assert xor_4 : (¬ (xor false false))

2 Basic_classes

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

open import *Bool*

declare {*isabelle*, *ocaml*, *hol*, *coq*} 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_match1 : (match LT with GT → false ∧ false | _ → true end)
assert ordering_match2 : (match EQ with GT → false | _ → true end)
assert ordering_match3 : (match GT with GT → true ∧ true | _ → false end)
assert ordering_match4 : ((fun r → (match r with GT → false | _ → true end)) LT)
assert ordering_match5 : ((fun r → (match r with GT → false | _ → true end)) EQ)
assert ordering_match6 : ((fun r → (match r with GT → true ∧ true | _ → false end)) GT)


val orderingEqual : ORDERING → ORDERING → 𝔹
let inline ∼{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 = ¬ (orderingEqual x y)
end

class ( Ord α )
  val compare : α → α → ORDERING
  val < [isLess] : α → α → 𝔹
  val <= [isLessEqual] : α → α → 𝔹
  val > [isGreater] : α → α → 𝔹
  val >= [isGreaterEqual] : α → α → 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 : ∀ α. α → α → ORDERING
val defaultLess : ∀ α. α → α → 𝔹
val defaultLessEq : ∀ α. α → α → 𝔹
val defaultGreater : ∀ α. α → α → 𝔹
val defaultGreaterEq : ∀ α. α → α → 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* $(less : \alpha \to \alpha \to \mathbb{B})$ $(equal : \alpha \to \alpha \to \mathbb{B})$ $(x : \alpha)$ $(y : \alpha)$ =
  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* : $\forall\, \alpha.\ Eq\ \alpha,\ Ord\ \alpha \Rightarrow \alpha \to \alpha \to$ ORDERING
let *ordCompare x y* =
  if $(x < y)$ then LT else
  if $(x = y)$ then EQ else GT

class (*OrdMaxMin* α)
  val *max* : $\alpha \to \alpha \to \alpha$
  val *min* : $\alpha \to \alpha \to \alpha$
end

val *minByLessEqual* : $\forall\, \alpha.\ (\alpha \to \alpha \to \mathbb{B}) \to \alpha \to \alpha \to \alpha$
let $\sim$\{*isabelle*\} *minByLessEqual le x y* = if (*le x y*) then *x* else *y*
let inline \{*isabelle*\} *minByLessEqual le x y* = if (*le x y*) then *x* else *y*

val *maxByLessEqual* : $\forall\, \alpha.\ (\alpha \to \alpha \to \mathbb{B}) \to \alpha \to \alpha \to \alpha$
let $\sim$\{*isabelle*\} *maxByLessEqual le x y* = if (*le y x*) then *x* else *y*
let inline \{*isabelle*\} *maxByLessEqual le x y* = if (*le y x*) then *x* else *y*

val *defaultMax* : $\forall\, \alpha.\ Ord\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$
let inline *defaultMax* = maxByLessEqual $(\leq)$

declare *ocaml* target_rep function defaultMax = 'max'

val *defaultMin* : $\forall\ \alpha.\ Ord\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha$
let inline *defaultMin* = minByLessEqual $(\leq)$
declare *ocaml* target_rep function defaultMin = 'min'

default_instance $\forall\ \alpha.\ Ord\ \alpha\ \Rightarrow\ (\ OrdMaxMin\ \alpha)$
  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* : $\alpha\ \rightarrow\ \alpha\ \rightarrow$ ORDERING
end

default_instance $\forall\ \alpha.\ (\ SetType\ \alpha\ )$
  let *setElemCompare* = defaultCompare
end

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

instance $(Eq\ \mathbb{B})$
  let $=\ =\ (\longleftrightarrow)$
  let $<>\ x\ y\ =\ \neg\ ((\longleftrightarrow)\ x\ y)$
end

let *boolCompare* $b_1\ b_2$ = match $(b_1,\ b_2)$ with
  | (true, true) $\rightarrow$ EQ
  | (true, false) $\rightarrow$ GT
  | (false, true) $\rightarrow$ LT
  | (false, false) $\rightarrow$ EQ
end

instance $(SetType\ \mathbb{B})$
  let *setElemCompare* = boolCompare
end

```
(* pairs *)
```

val *pairEqual* : $\forall\ \alpha\ \beta.\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ (\alpha\ *\ \beta)\ \rightarrow\ (\alpha\ *\ \beta)\ \rightarrow\ \mathbb{B}$
let *pairEqual* $(a_1,\ b_1)\ (a_2,\ b_2)\ =\ (a_1 = a_2) \wedge (b_1 = b_2)$

```
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 = ¬ (pairEqual x y)
end

val pairCompare : ∀ α β. (α → α → ORDERING) → (β → β → ORDERING) → (α ∗ β) →
(α ∗ β) → ORDERING
let pairCompare cmpa cmpb (a1, b1) (a2, b2) =
  match cmpa a1 a2 with
    | LT → LT
    | GT → GT
    | EQ → cmpb b1 b2
  end

let pairLess (x1, x2) (y1, y2) = (x1 < y1) ∨ ((x1 ≤ y1) ∧ (x2 < y2))
let pairLessEq (x1, x2) (y1, y2) = (x1 < y1) ∨ ((x1 ≤ y1) ∧ (x2 ≤ y2))

let pairGreater x12 y12 = pairLess y12 x12
let pairGreaterEq x12 y12 = pairLessEq y12 x12

instance ∀ α β. Ord α, Ord β ⇒ (Ord (α ∗ β))
  let compare = pairCompare compare compare
  let < = pairLess
  let <= = pairLessEq
  let > = pairGreater
  let >= = pairGreaterEq
end

instance ∀ α β. SetType α, SetType β ⇒ (SetType (α ∗ β))
  let setElemCompare = pairCompare setElemCompare setElemCompare
end


(* triples *)

val tripleEqual : ∀ α β γ. Eq α, Eq β, Eq γ ⇒ (α ∗ β ∗ γ) → (α ∗ β ∗ γ) → 𝔹
let tripleEqual (x1, x2, x3) (y1, y2, y3) = ((x1, (x2, x3)) = (y1, (y2, y3)))
let inline {hol, isabelle} tripleEqual = unsafe_structural_equality

instance ∀ α β γ. Eq α, Eq β, Eq γ ⇒ (Eq (α ∗ β ∗ γ))
  let = = tripleEqual
  let <> x y = ¬ (tripleEqual x y)
end

val tripleCompare : ∀ α β γ. (α → α → ORDERING) → (β → β → ORDERING) → (γ → γ →
ORDERING) → (α ∗ β ∗ γ) → (α ∗ β ∗ γ) → ORDERING
let tripleCompare cmpa cmpb cmpc (a1, b1, c1) (a2, b2, c2) =
  pairCompare cmpa (pairCompare cmpb cmpc) (a1, (b1, c1)) (a2, (b2, c2))

let tripleLess (x1, x2, x3) (y1, y2, y3) = (x1, (x2, x3)) < (y1, (y2, y3))
let tripleLessEq (x1, x2, x3) (y1, y2, y3) = (x1, (x2, x3)) ≤ (y1, (y2, y3))
```

let $tripleGreater\ x_{123}\ y_{123}\ =\ \mathrm{tripleLess}\ y_{123}\ x_{123}$
let $tripleGreaterEq\ x_{123}\ y_{123}\ =\ \mathrm{tripleLessEq}\ y_{123}\ x_{123}$

instance $\forall\ \alpha\ \beta\ \gamma.\ Ord\ \alpha,\ Ord\ \beta,\ Ord\ \gamma\ \Rightarrow\ (Ord\ (\alpha\ *\ \beta\ *\ \gamma))$
  let $compare\ =\ \mathrm{tripleCompare\ compare\ compare\ compare}$
  let $<\ =\ \mathrm{tripleLess}$
  let $<=\ =\ \mathrm{tripleLessEq}$
  let $>\ =\ \mathrm{tripleGreater}$
  let $>=\ =\ \mathrm{tripleGreaterEq}$
end

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

3 Function

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

open import *Bool Basic_classes*

declare {*isabelle*, *hol*, *ocaml*, *coq*} rename module = lem_function

open import {*coq*} *Program.Basics*

```
(* ---------------------- *)
(* 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 $x \to f\ (g\ x)$)

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 $x \to f\ x$)

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*, *coq*} 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* : $\forall\ \alpha.\ Eq\ \alpha\ \Rightarrow$ MAYBE $\alpha\ \rightarrow$ MAYBE $\alpha\ \rightarrow\ \mathbb{B}$
val *maybeEqualBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow$ MAYBE $\alpha\ \rightarrow$ MAYBE $\alpha\ \rightarrow\ \mathbb{B}$

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

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

instance $\forall\ \alpha.\ Eq\ \alpha\ \Rightarrow\ (Eq$ (MAYBE α))
 let = = maybeEqual
 let <> $x\ y$ = $\neg$ (maybeEqual $x\ y$)
end

assert $maybe\_eq_1$: ((Nothing : MAYBE $\mathbb{B}$) = Nothing)
assert $maybe\_eq_2$: ((Just true) $\neq$ Nothing)
assert $maybe\_eq_3$: ((Just false) $\neq$ (Just true))

```
assert maybe_eq4 : ((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 maybe1 :  (maybe true (fun b → ¬ b) Nothing = true)
assert maybe2 :  (maybe false (fun b → ¬ b) Nothing = false)
assert maybe3 :  (maybe true (fun b → ¬ b) (Just true) = false)
assert maybe4 :  (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 isJust1 :  (isJust (Just true))
assert isJust2 :  (¬ (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 $isNothing_1$: $(\neg$ (isNothing (Just true)))
assert $isNothing_2$: (isNothing (Nothing : MAYBE $\mathbb{B}$))

lemma *isJustNothing* : (
 $(\forall\ x.$ isNothing $x = \neg$ (isJust $x)) \wedge$
 $(\forall\ v.$ isJust (Just $v)) \wedge$
 (isNothing Nothing))

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

val *fromMaybe* : $\forall\ \alpha.\ \alpha \ \rightarrow$ MAYBE $\alpha \ \rightarrow \ \alpha$
let *fromMaybe* d mb $=$ match mb with
 | Just $v \ \rightarrow \ v$
 | Nothing $\rightarrow \ d$
end

declare *ocaml* target_rep function fromMaybe $=$ `'Lem.option_default'`
let inline {*isabelle*, *hol*} *fromMaybe* d $=$ maybe d id

lemma *fromMaybe* : (
 $(\forall\ d\ v.$ fromMaybe d (Just $v) = v) \wedge$
 $(\forall\ d.$ 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* : $\forall\ \alpha\ \beta.\ (\alpha \ \rightarrow \ \beta) \ \rightarrow$ MAYBE $\alpha \ \rightarrow$ MAYBE β
let *map* f $=$ maybe Nothing (fun $v \ \rightarrow$ 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* : (
 $(\forall\ f.$ map f Nothing = Nothing$) \wedge$
 $(\forall\ f\ v.$ map f (Just $v)$ = Just $(f\ v)))$

assert map_1 : (map (fun $b \rightarrow \neg\ b$) Nothing = Nothing)
assert map_2 : (map (fun $b \rightarrow \neg\ b$) (Just true) = Just false)
assert map_3 : (map (fun $b \rightarrow \neg\ b$) (Just false) = Just true)

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

val $bind$: $\forall\ \alpha\ \beta.$ MAYBE $\alpha\ \rightarrow\ (\alpha\ \rightarrow$ MAYBE $\beta)\ \rightarrow$ MAYBE β
let $bind\ mb\ f\ =\ \mathrm{maybe\ Nothing}\ f\ mb$

declare $isabelle$ target_rep function bind = `'Option.bind'`
declare $ocaml$ target_rep function bind = `'Lem.option_bind'`
declare hol target_rep function bind = `'OPTION_BIND'`

lemma $maybe\_bind$: (
  ($\forall\ f.$ bind Nothing f = Nothing) $\wedge$
  ($\forall\ f\ v.$ bind (Just v) f = ($f\ v$)))

assert $bind_1$: (bind Nothing (fun $b\ \rightarrow$ Just ($\neg\ b$)) = Nothing)
assert $bind_2$: (bind (Just true) (fun $b\ \rightarrow$ Just ($\neg\ b$)) = Just false)
assert $bind_3$: (bind (Just false) (fun $b\ \rightarrow$ Just ($\neg\ b$)) = Just true)
assert $bind_4$: (bind (Just false) (fun $b\ \rightarrow$ (Nothing : MAYBE $\mathbb{B}$)) = 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*, *coq*} rename module = lem_num

open import *Bool Basic_classes*
open import {*isabelle*} *~~/src/HOL/Word/Word*
open import {*hol*} *integerTheory intReduce wordsTheory wordsLib*
open import {*coq*} *Coq.ZArith.BinInt Coq.ZArith.Zpower Coq.ZArith.Zdiv Coq.ZArith.Zmax*

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

(* Numerals like 0, 1, 2, 42, 4543 are built-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 (*NumAbs* α)
 val *abs* : α $\rightarrow$ α
end

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

class (*NumMinus* α)
  val $-$ [numMinus] : $\alpha \rightarrow \alpha \rightarrow \alpha$
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                    *)
(* ----------------------- *)

(* TODO the bounded ints are only partially implemented, use with care. *)

(* 32 bit integers *)
```

type INT_{32}
declare *ocaml* target_rep type INT_{32} = 'Int32.t'
declare *coq* target_rep type INT_{32} = 'Z' (* ???: better type for this in Coq? *)
declare *isabelle* target_rep type INT_{32} = 'word' 32
declare *hol* target_rep type INT_{32} = 'word'$_{32}$

```
(* 64 bit integers *)
```

type INT_{64}
declare *ocaml* target_rep type INT_{64} = 'Int64.t'

declare *coq* target_rep type INT_{64} = 'Z' (* ???: better type for this in Coq? *)
declare *isabelle* target_rep type INT_{64} = 'word' 64
declare *hol* target_rep type INT_{64} = 'word'$_{64}$

```
(* ----------------------- *)
(* 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 FLOAT_{64}
declare *ocaml* target_rep type FLOAT_{64} = 'double'
declare *coq* target_rep type FLOAT_{64} = 'Q' (* ???: better type for this in Coq? *)
declare *isabelle* target_rep type FLOAT_{64} = '???' (* ???: better type for this in Isa? *)
declare *hol* target_rep type FLOAT_{64} = 'XXX' (* ???: better type for this in HOL? *)

type FLOAT_{32}
declare *ocaml* target_rep type FLOAT_{32} = 'float'
declare *coq* target_rep type FLOAT_{32} = 'Q' (* ???: better type for this in Coq? *)
declare *isabelle* target_rep type FLOAT_{32} = '???' (* ???: better type for this in Isa? *)
declare *hol* target_rep type FLOAT_{32} = '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 → $\mathbb{B}$

```
let inline natEq = unsafe_structural_equality
declare coq target_rep function natEq = 'beq_nat'
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 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 gen_pow_aux : ∀ α. (α → α → α) → α → α → nat → α
let rec gen_pow_aux (mul : α → α → α) (a : α) (b : α) (e : nat) =
   match e with
     | 0 → a (* cannot happen, call discipline guarentees e >= 1 *)
     | 1 → mul a b
     | (e' + 2) → let e'' = e / 2 in
                  let a' = (if (e mod 2) = 0 then a else mul a b) in
                  gen_pow_aux mul a' (mul b b) e''
   end
declare termination_argument gen_pow_aux = automatic

let gen_pow (one : α) (mul : α → α → α) (b : α) (e : nat) : α =
  if e < 0 then one else
  if (e = 0) then one else gen_pow_aux mul one b e

val natPow : nat → nat → nat
let {ocaml} natPow = gen_pow 1 natMult

declare hol target_rep function natPow = infix '**'
declare isabelle target_rep function natPow = infix '↑'
declare coq target_rep function natPow = 'nat_power'

instance ( NumPow nat )
  let ** = natPow
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'
declare coq target_rep function naturalEq = 'beq_nat'
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 : ℕ → ℕ → 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 : ℕ → ℕ → 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 ℕ )
  let max = naturalMax
  let min = naturalMin
end


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

val intFromNumeral : NUMERAL → 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 → INT → 𝔹
let inline intEq = unsafe_structural_equality
declare coq target_rep function intEq = 'Z.eqb'
instance (Eq INT)
  let = = intEq
  let <> n1 n2 = ¬ (intEq n1 n2)
end

val intLess : INT → INT → 𝔹
val intLessEqual : INT → INT → 𝔹
val intGreater : INT → INT → 𝔹
val intGreaterEqual : INT → INT → 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 $\sim$ = intNegate
end

val *intAbs* : INT $\rightarrow$ INT
declare *hol* target_rep function intAbs = 'ABS'
declare *ocaml* target_rep function intAbs = 'abs'
declare *isabelle* target_rep function intAbs = 'abs'
declare *coq* target_rep function intAbs *input* = 'Zpred' ('Zpos' ('P_of_succ_nat' ('Zabs_nat' *input*)))
(* TODO: check *)

instance (*NumAbs* INT)
  let *abs* = intAbs
end

val *intAdd* : INT $\rightarrow$ INT $\rightarrow$ 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 $\rightarrow$ INT $\rightarrow$ 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 $\rightarrow$ INT
let inline *intSucc* n = $n + 1$
declare *ocaml* target_rep function intSucc = 'succ'
instance (*NumSucc* INT)
  let *succ* = intSucc
end

val *intPred* : INT $\rightarrow$ INT
let inline *intPred* n = $n - 1$
declare *ocaml* target_rep function intPred = 'pred'
instance (*NumPred* INT)
  let *pred* = intPred
end

val *intMult* : INT $\rightarrow$ INT $\rightarrow$ 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
let {ocaml} intPow = gen_pow 1 intMult
declare hol target_rep function intPow = infix '**'
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

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

```
(* int32                  *)
(* ---------------------- *)
val int32FromNumeral : NUMERAL → INT32

declare ocaml target_rep function int32FromNumeral = 'Int32.of_int'
declare isabelle target_rep function int32FromNumeral n = (('word_of_int' n) : INT32)
declare hol target_rep function int32FromNumeral n = (('n2w' n) : INT32)
declare coq target_rep function int32FromNumeral n = ('Zpos' ('P_of_succ_nat' n)) (* TODO: check *)

instance (Numeral INT32)
  let fromNumeral n = int32FromNumeral n
end

val int32Eq : INT32 → INT32 → 𝔹
let inline int32Eq = unsafe_structural_equality
declare coq target_rep function int32Eq = 'Z.eqb'

instance (Eq INT32)
  let = = int32Eq
  let <> n1 n2 = ¬ (int32Eq n1 n2)
end

val int32Less : INT32 → INT32 → 𝔹
val int32LessEqual : INT32 → INT32 → 𝔹
val int32Greater : INT32 → INT32 → 𝔹
val int32GreaterEqual : INT32 → INT32 → 𝔹

declare ocaml target_rep function int32Less = infix '<'
declare isabelle target_rep function int32Less = 'word_sless'
declare hol target_rep function int32Less = infix '<'
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Less = 'int_ltb'

declare ocaml target_rep function int32LessEqual = infix '<='
declare isabelle target_rep function int32LessEqual = 'word_sle'
declare hol target_rep function int32LessEqual = infix '<='
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32LessEqual = 'int_lteb'

declare ocaml target_rep function int32Greater = infix '>'
let inline {isabelle} int32Greater x y = int32Less y x
declare hol target_rep function int32Greater = infix '>'
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Greater = 'int_gtb'

declare ocaml target_rep function int32GreaterEqual = infix '>='
let inline {isabelle} int32GreaterEqual x y = int32LessEqual y x
declare hol target_rep function int32GreaterEqual = infix '>='
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32GreaterEqual = 'int_gteb'

val int32Compare : INT32 → INT32 → ORDERING
let inline int32Compare = defaultCompare
let inline {coq, isabelle, hol} int32Compare = genericCompare int32Less int32Eq
declare ocaml target_rep function int32Compare = 'Int32.compare'

instance (Ord INT32)
  let compare = int32Compare
```

```
  let < = int32Less
  let <= = int32LessEqual
  let > = int32Greater
  let >= = int32GreaterEqual
end

instance (SetType INT32)
  let setElemCompare = int32Compare
end

val int32Negate : INT32 → INT32
declare ocaml target_rep function int32Negate = 'Int32.neg'
declare isabelle target_rep function int32Negate i = '-' i
declare hol target_rep function int32Negate i = (('-' i) : INT32)
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Negate i = ('Coq.ZArith.BinInt.Zminus' 'Z'0 i)

instance (NumNegate INT32)
  let ~ = int32Negate
end

val int32Abs : INT32 → INT32
let int32Abs i = (if 0 ≤ i then i else −i)
declare ocaml target_rep function int32Abs = 'Int32.abs'

instance (NumAbs INT32)
  let abs = int32Abs
end


val int32Add : INT32 → INT32 → INT32
declare ocaml target_rep function int32Add = 'Int32.add'
declare isabelle target_rep function int32Add = infix '+'
(*TODO: Implement the following two correctly. *)
declare hol target_rep function int32Add i1 i2 = (('word_add' i1 i2) : INT32)
declare coq target_rep function int32Add = 'Coq.ZArith.BinInt.Zplus'

instance (NumAdd INT32)
  let + = int32Add
end

val int32Minus : INT32 → INT32 → INT32
declare ocaml target_rep function int32Minus = 'Int32.sub'
declare isabelle target_rep function int32Minus = infix '-'
(*TODO: Implement the following two correctly. *)
declare hol target_rep function int32Minus i1 i2 = (('word_sub' i1 i2) : INT32)
declare coq target_rep function int32Minus = 'Coq.ZArith.BinInt.Zminus'

instance (NumMinus INT32)
  let − = int32Minus
end

val int32Succ : INT32 → INT32
let inline int32Succ n = n + 1
declare ocaml target_rep function int32Succ = 'Int32.succ'

instance (NumSucc INT32)
  let succ = int32Succ
```

end

val $int32Pred$: $\text{INT}_{32} \rightarrow \text{INT}_{32}$
let inline $int32Pred\ n = n - 1$
declare $ocaml$ target_rep function int32Pred = `'Int32.pred'`
instance ($NumPred$ INT_{32})
  let $pred$ = int32Pred
end

val $int32Mult$: $\text{INT}_{32} \rightarrow \text{INT}_{32} \rightarrow \text{INT}_{32}$
declare $ocaml$ target_rep function int32Mult = `'Int32.mul'`
declare $isabelle$ target_rep function int32Mult = infix `'*'`
declare hol target_rep function int32Mult i_1 i_2 = ((`'word_mul'` i_1 i_2) : INT_{32})
`(*TODO: Implement the following correctly. *)`
declare coq target_rep function int32Mult = `'Coq.ZArith.BinInt.Zmult'`

instance ($NumMult$ INT_{32})
  let $*$ = int32Mult
end

val $int32Pow$: $\text{INT}_{32} \rightarrow \text{NAT} \rightarrow \text{INT}_{32}$
let $\{ocaml, hol\}$ $int32Pow$ = gen_pow 1 int32Mult
declare $isabelle$ target_rep function int32Pow = infix `'↑'`
`(*TODO: Implement the following two correctly. *)`
declare coq target_rep function int32Pow = `'Coq.ZArith.Zpower.Zpower_nat'`

instance ($NumPow$ INT_{32})
  let $**$ = int32Pow
end

val $int32Div$: $\text{INT}_{32} \rightarrow \text{INT}_{32} \rightarrow \text{INT}_{32}$
declare $ocaml$ target_rep function int32Div = `'Nat_num.int32_div'`
declare $isabelle$ target_rep function int32Div = infix `'div'`
declare hol target_rep function int32Div i_1 i_2 = ((`'word_div'` i_1 i_2) : INT_{32})
`(*TODO: Implement the following correctly. *)`
declare coq target_rep function int32Div = `'Coq.ZArith.Zdiv.Zdiv'`

instance ($NumIntegerDivision$ INT_{32})
  let div = int32Div
end

instance ($NumDivision$ INT_{32})
  let $/$ = int32Div
end

val $int32Mod$: $\text{INT}_{32} \rightarrow \text{INT}_{32} \rightarrow \text{INT}_{32}$
declare $ocaml$ target_rep function int32Mod = `'Nat_num.int32_mod'`
declare $isabelle$ target_rep function int32Mod = infix `'mod'`
declare hol target_rep function int32Mod i_1 i_2 = ((`'word_mod'` i_1 i_2) : INT_{32})
`(*TODO: Implement the following correctly. *)`
declare coq target_rep function int32Mod = `'Coq.ZArith.Zdiv.Zmod'`

instance ($NumRemainder$ INT_{32})
  let mod = int32Mod
end

val $int32Min$: $\text{INT}_{32} \rightarrow \text{INT}_{32} \rightarrow \text{INT}_{32}$

```
let inline int32Min = defaultMin
declare hol target_rep function int32Min = 'word_smin'
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Min = 'Zmin'

val int32Max : INT32 → INT32 → INT32
let inline int32Max = defaultMax
declare hol target_rep function int32Max = 'word_smax'
(*TODO: Implement the following correctly. *)
declare coq target_rep function int32Max = 'Zmax'

instance ( OrdMaxMin INT32 )
  let max = int32Max
  let min = int32Min
end



(* ----------------------- *)
(* int64                   *)
(* ----------------------- *)
val int64FromNumeral : NUMERAL → INT64

declare ocaml target_rep function int64FromNumeral = 'Int64.of_int'
declare isabelle target_rep function int64FromNumeral n = (('word_of_int' n) : INT64)
declare hol target_rep function int64FromNumeral n = (('n2w' n) : INT64)
declare coq target_rep function int64FromNumeral n = ('Zpos' ('P_of_succ_nat' n)) (* TODO: check *)

instance (Numeral INT64)
  let fromNumeral n = int64FromNumeral n
end

val int64Eq : INT64 → INT64 → 𝔹
let inline int64Eq = unsafe_structural_equality
declare coq target_rep function int64Eq = 'Z.eqb'

instance (Eq INT64)
  let = = int64Eq
  let <> n1 n2 = ¬ (int64Eq n1 n2)
end

val int64Less : INT64 → INT64 → 𝔹
val int64LessEqual : INT64 → INT64 → 𝔹
val int64Greater : INT64 → INT64 → 𝔹
val int64GreaterEqual : INT64 → INT64 → 𝔹

declare ocaml target_rep function int64Less = infix '<'
declare isabelle target_rep function int64Less = 'word_sless'
declare hol target_rep function int64Less = infix '<'
(*TODO: Implement the following correctly. *)
declare coq target_rep function int64Less = 'int_ltb'

declare ocaml target_rep function int64LessEqual = infix '<='
declare isabelle target_rep function int64LessEqual = 'word_sle'
declare hol target_rep function int64LessEqual = infix '<='
(*TODO: Implement the following correctly. *)
declare coq target_rep function int64LessEqual = 'int_lteb'
```

```
declare ocaml target_rep function int64Greater = infix '>'
let inline {isabelle} int64Greater x y = int64Less y x
declare hol target_rep function int64Greater = infix '>'
(*TODO: Implement the following correctly. *)
declare coq target_rep function int64Greater = 'int_gtb'

declare ocaml target_rep function int64GreaterEqual = infix '>='
let inline {isabelle} int64GreaterEqual x y = int64LessEqual y x
declare hol target_rep function int64GreaterEqual = infix '>='
(*TODO: Implement the following correctly. *)
declare coq target_rep function int64GreaterEqual = 'int_gteb'

val int64Compare : INT64 → INT64 → ORDERING
let inline int64Compare = defaultCompare
let inline {coq, isabelle, hol} int64Compare = genericCompare int64Less int64Eq
declare ocaml target_rep function int64Compare = 'Int64.compare'

instance (Ord INT64)
  let compare = int64Compare
  let < = int64Less
  let <= = int64LessEqual
  let > = int64Greater
  let >= = int64GreaterEqual
end

instance (SetType INT64)
  let setElemCompare = int64Compare
end

val int64Negate : INT64 → INT64
declare ocaml target_rep function int64Negate = 'Int64.neg'
declare isabelle target_rep function int64Negate i = '-' i
declare hol target_rep function int64Negate i = (('-' i) : INT64)
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Negate i = ('Coq.ZArith.BinInt.Zminus' 'Z'0 i)

instance (NumNegate INT64)
  let ~ = int64Negate
end

val int64Abs : INT64 → INT64
let int64Abs i = (if 0 ≤ i then i else −i)
declare ocaml target_rep function int64Abs = 'Int64.abs'

instance (NumAbs INT64)
  let abs = int64Abs
end


val int64Add : INT64 → INT64 → INT64
declare ocaml target_rep function int64Add = 'Int64.add'
declare isabelle target_rep function int64Add = infix '+'
declare hol target_rep function int64Add i1 i2 = (('word_add' i1 i2) : INT64)
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Add = 'Coq.ZArith.BinInt.Zplus'

instance (NumAdd INT64)
  let + = int64Add
```

```
end

val int64Minus : INT64 → INT64 → INT64
declare ocaml target_rep function int64Minus = 'Int64.sub'
declare isabelle target_rep function int64Minus = infix '-'
declare hol target_rep function int64Minus i1 i2 = (('word_sub' i1 i2) : INT64)
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Minus = 'Coq.ZArith.BinInt.Zminus'

instance (NumMinus INT64)
  let − = int64Minus
end

val int64Succ : INT64 → INT64
let inline int64Succ n = n + 1
declare ocaml target_rep function int64Succ = 'Int64.succ'

instance (NumSucc INT64)
  let succ = int64Succ
end

val int64Pred : INT64 → INT64
let inline int64Pred n = n − 1
declare ocaml target_rep function int64Pred = 'Int64.pred'
instance (NumPred INT64)
  let pred = int64Pred
end

val int64Mult : INT64 → INT64 → INT64
declare ocaml target_rep function int64Mult = 'Int64.mul'
declare isabelle target_rep function int64Mult = infix '*'
declare hol target_rep function int64Mult i1 i2 = (('word_mul' i1 i2) : INT64)
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Mult = 'Coq.ZArith.BinInt.Zmult'

instance (NumMult INT64)
  let ∗ = int64Mult
end


val int64Pow : INT64 → NAT → INT64
let {ocaml, hol} int64Pow = gen_pow 1 int64Mult
declare isabelle target_rep function int64Pow = infix '↑'
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Pow = 'Coq.ZArith.Zpower.Zpower_nat'

instance ( NumPow INT64 )
  let ∗∗ = int64Pow
end

val int64Div : INT64 → INT64 → INT64
declare ocaml target_rep function int64Div = 'Nat_num.int64_div'
declare isabelle target_rep function int64Div = infix 'div'
(*TODO: Implement the following two correctly. *)
declare hol target_rep function int64Div i1 i2 = (('word_div' i1 i2) : INT64)
declare coq target_rep function int64Div = 'Coq.ZArith.Zdiv.Zdiv'

instance ( NumIntegerDivision INT64 )
```

```
  let div = int64Div
end

instance ( NumDivision int64 )
  let / = int64Div
end

val int64Mod : int64 → int64 → int64
declare ocaml target_rep function int64Mod = 'Nat_num.int64_mod'
declare isabelle target_rep function int64Mod = infix 'mod'
(*TODO: Implement the following two correctly. *)
declare hol target_rep function int64Mod i1 i2 = (('word_mod' i1 i2) : int64)
declare coq target_rep function int64Mod = 'Coq.ZArith.Zdiv.Zmod'

instance ( NumRemainder int64 )
  let mod = int64Mod
end

val int64Min : int64 → int64 → int64
let inline int64Min = defaultMin
declare hol target_rep function int64Min = 'word_smin'
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Min = 'Zmin'

val int64Max : int64 → int64 → int64
let inline int64Max = defaultMax
declare hol target_rep function int64Max = 'word_smax'
(*TODO: Implement the following one correctly. *)
declare coq target_rep function int64Max = 'Zmax'

instance ( OrdMaxMin int64 )
  let max = int64Max
  let min = int64Min
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'
declare coq target_rep function integerEq = 'Z.eqb'
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 integerAbs : ℤ → ℤ
declare hol target_rep function integerAbs = 'ABS'
declare ocaml target_rep function integerAbs = 'Big_int.abs_big_int'
declare isabelle target_rep function integerAbs = 'abs'
declare coq target_rep function integerAbs input = 'Zpred' ('Zpos' ('P_of_succ_nat' ('Zabs_nat' input)))
```

```
(* TODO: check *)

instance (NumAbs ℤ)
  let abs = integerAbs
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_test_7 : $(11 \;/\; (4 : \textsc{nat}) = 2)$
assert nat_test_8 : $(11 \bmod (4 : \textsc{nat}) = 3)$
assert nat_test_9 : $(11 < (12 : \textsc{nat}))$
assert nat_test_{10} : $(11 \leq (12 : \textsc{nat}))$
assert nat_test_{11} : $(12 \leq (12 : \textsc{nat}))$
assert nat_test_{12} : $(\neg\,(12 < (12 : \textsc{nat})))$
assert nat_test_{13} : $(12 > (11 : \textsc{nat}))$
assert nat_test_{14} : $(12 \geq (11 : \textsc{nat}))$
assert nat_test_{15} : $(12 \geq (12 : \textsc{nat}))$
assert nat_test_{16} : $(\neg\,(12 > (12 : \textsc{nat})))$
assert nat_test_{17} : $(\min 12\ (12 : \textsc{nat}) = 12)$
assert nat_test_{18} : $(\min 10\ (12 : \textsc{nat}) = 10)$
assert nat_test_{19} : $(\min 12\ (10 : \textsc{nat}) = 10)$
assert nat_test_{20} : $(\max 12\ (12 : \textsc{nat}) = 12)$
assert nat_test_{21} : $(\max 10\ (12 : \textsc{nat}) = 12)$
assert nat_test_{22} : $(\max 12\ (10 : \textsc{nat}) = 12)$
assert nat_test_{23} : $(\mathrm{succ}\ 12 = (13 : \textsc{nat}))$
assert nat_test_{24} : $(\mathrm{succ}\ 0 = (1 : \textsc{nat}))$
assert nat_test_{25} : $(\mathrm{pred}\ 12 = (11 : \textsc{nat}))$
assert nat_test_{26} : $(\mathrm{pred}\ 0 = (0 : \textsc{nat}))$
assert nat_test_{27} : (match $(27 : \textsc{nat})$ with
  | $0 \rightarrow$ false
  | $x + 2 \rightarrow (x = 25)$
  | $x + 1 \rightarrow (x = 26)$
 end)
assert $nat\_test28a$: (match $(27 : \textsc{nat})$ 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 $(30 : \textsc{nat})$ 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 $natural\_test_6$: $(\mathrm{div}\ 11\ (4 : \mathbb{N}) = 2)$
assert $natural\_test_7$: $(11 \;/\; (4 : \mathbb{N}) = 2)$
assert $natural\_test_8$: $(11 \bmod (4 : \mathbb{N}) = 3)$
assert $natural\_test_9$: $(11 < (12 : \mathbb{N}))$
assert $natural\_test_{10}$: $(11 \leq (12 : \mathbb{N}))$
assert $natural\_test_{11}$: $(12 \leq (12 : \mathbb{N}))$
assert $natural\_test_{12}$: $(\neg\,(12 < (12 : \mathbb{N})))$

assert $natural\_test_{13}$: $(12 > (11 : \mathbb{N}))$
assert $natural\_test_{14}$: $(12 \geq (11 : \mathbb{N}))$
assert $natural\_test_{15}$: $(12 \geq (12 : \mathbb{N}))$
assert $natural\_test_{16}$: $(\neg (12 > (12 : \mathbb{N})))$
assert $natural\_test_{17}$: $(\min 12\ (12 : \mathbb{N}) = 12)$
assert $natural\_test_{18}$: $(\min 10\ (12 : \mathbb{N}) = 10)$
assert $natural\_test_{19}$: $(\min 12\ (10 : \mathbb{N}) = 10)$
assert $natural\_test_{20}$: $(\max 12\ (12 : \mathbb{N}) = 12)$
assert $natural\_test_{21}$: $(\max 10\ (12 : \mathbb{N}) = 12)$
assert $natural\_test_{22}$: $(\max 12\ (10 : \mathbb{N}) = 12)$
assert $natural\_test_{23}$: $(\mathrm{succ}\ 12 = (13 : \mathbb{N}))$
assert $natural\_test_{24}$: $(\mathrm{succ}\ 0 = (1 : \mathbb{N}))$
assert $natural\_test_{25}$: $(\mathrm{pred}\ 12 = (11 : \mathbb{N}))$
assert $natural\_test_{26}$: $(\mathrm{pred}\ 0 = (0 : \mathbb{N}))$
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
  | $n + 50 \rightarrow$ "$50<=x$"
  | $40 \rightarrow$ "$x = 40$"
  | $n + 31 \rightarrow$ "$x<>40\&\&31<=x<50$"
  | $29 \rightarrow$ "$x = 29$"
  | $n + 30 \rightarrow$ "$x = 30$"
  | $4 \rightarrow$ "$x = 4$"
  | _ $\rightarrow$ "$x<>4\&\&x<>29\&\&x<30$"
  end = "$x<>4\&\&x<>29\&\&x<30$")
assert $natural\_test28b$: (match $(30 : \mathbb{N})$ with
  | $n + 50 \rightarrow$ "$50<=x$"
  | $40 \rightarrow$ "$x = 40$"
  | $n + 31 \rightarrow$ "$x<>40\&\&31<=x<50$"
  | $29 \rightarrow$ "$x = 29$"
  | $n + 30 \rightarrow$ "$x = 30$"
  | $4 \rightarrow$ "$x = 4$"
  | _ $\rightarrow$ "$x<>4\&\&x<>29\&\&x<30$"
  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 : $(\mathrm{div}\ 11\ (4 : \text{INT}) = 2)$
assert int_test6a : $(\mathrm{div}\ (-\ 11)\ (4 : \text{INT}) = -3)$
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 int_test_{27} : $(\text{abs } 42 = (42 : \text{INT}))$
assert int_test_{28} : $(\text{abs } (-42) = (42 : \text{INT}))$

assert $int32\_test_{1}$: $(2 + (5 : \text{INT}_{32}) = 7)$
assert $int32\_test_{2}$: $(8 - (7 : \text{INT}_{32}) = 1)$
assert $int32\_test_{3}$: $(7 - (8 : \text{INT}_{32}) = -1)$
assert $int32\_test_{4}$: $(7 * (8 : \text{INT}_{32}) = 56)$
assert $int32\_test_{5}$: $((7 : \text{INT}_{32})^2 = 49)$
assert $int32\_test_{6}$: $(\text{div } 11\ (4 : \text{INT}_{32}) = 2)$
assert $int32\_test_{7}$: $(11 \mathbin{/} (4 : \text{INT}_{32}) = 2)$
assert $int32\_test_{8}$: $(11 \bmod (4 : \text{INT}_{32}) = 3)$
assert $int32\_test_{9}$: $(11 < (12 : \text{INT}_{32}))$
assert $int32\_test_{10}$: $(11 \leq (12 : \text{INT}_{32}))$
assert $int32\_test_{11}$: $(12 \leq (12 : \text{INT}_{32}))$
assert $int32\_test_{12}$: $(\neg\ (12 < (12 : \text{INT}_{32})))$
assert $int32\_test_{13}$: $(12 > (11 : \text{INT}_{32}))$
assert $int32\_test13a$: $(12 > (-(11 : \text{INT}_{32})))$
assert $int32\_test_{14}$: $(12 \geq (11 : \text{INT}_{32}))$
assert $int32\_test_{15}$: $(12 \geq (12 : \text{INT}_{32}))$
assert $int32\_test_{16}$: $(\neg\ (12 > (12 : \text{INT}_{32})))$
assert $int32\_test_{17}$: $(\min 12\ (12 : \text{INT}_{32}) = 12)$
assert $int32\_test_{18}$: $(\min 10\ (12 : \text{INT}_{32}) = 10)$
assert $int32\_test_{19}$: $(\min 12\ (10 : \text{INT}_{32}) = 10)$
assert $int32\_test_{20}$: $(\max 12\ (12 : \text{INT}_{32}) = 12)$
assert $int32\_test_{21}$: $(\max (-10)\ (12 : \text{INT}_{32}) = 12)$
assert $int32\_test_{22}$: $(\max 12\ (10 : \text{INT}_{32}) = 12)$
assert $int32\_test_{23}$: $(\text{succ } 12 = (13 : \text{INT}_{32}))$
assert $int32\_test_{24}$: $(\text{succ } 0 = (1 : \text{INT}_{32}))$
assert $int32\_test_{25}$: $(\text{pred } 12 = (11 : \text{INT}_{32}))$
assert $int32\_test_{26}$: $(\text{pred } 0 = -(1 : \text{INT}_{32}))$
assert $int32\_test_{27}$: $(\text{abs } 42 = (42 : \text{INT}_{32}))$
assert $int32\_test_{28}$: $(\text{abs } (-42) = (42 : \text{INT}_{32}))$

assert $int64\_test_{1}$: $(2 + (5 : \text{INT}_{64}) = 7)$
assert $int64\_test_{2}$: $(8 - (7 : \text{INT}_{64}) = 1)$
assert $int64\_test_{3}$: $(7 - (8 : \text{INT}_{64}) = -1)$
assert $int64\_test_{4}$: $(7 * (8 : \text{INT}_{64}) = 56)$
assert $int64\_test_{5}$: $((7 : \text{INT}_{64})^2 = 49)$
assert $int64\_test_{6}$: $(\text{div } 11\ (4 : \text{INT}_{64}) = 2)$
assert $int64\_test_{7}$: $(11 \mathbin{/} (4 : \text{INT}_{64}) = 2)$
assert $int64\_test_{8}$: $(11 \bmod (4 : \text{INT}_{64}) = 3)$
assert $int64\_test_{9}$: $(11 < (12 : \text{INT}_{64}))$
assert $int64\_test_{10}$: $(11 \leq (12 : \text{INT}_{64}))$
assert $int64\_test_{11}$: $(12 \leq (12 : \text{INT}_{64}))$
assert $int64\_test_{12}$: $(\neg\ (12 < (12 : \text{INT}_{64})))$
assert $int64\_test_{13}$: $(12 > (11 : \text{INT}_{64}))$
assert $int64\_test13a$: $(12 > (-(11 : \text{INT}_{64})))$
assert $int64\_test_{14}$: $(12 \geq (11 : \text{INT}_{64}))$

assert $int64\_test_{15}$: $(12 \geq (12 : \text{INT}_{64}))$
assert $int64\_test_{16}$: $(\neg (12 > (12 : \text{INT}_{64})))$
assert $int64\_test_{17}$: $(\min 12\ (12 : \text{INT}_{64}) = 12)$
assert $int64\_test_{18}$: $(\min 10\ (12 : \text{INT}_{64}) = 10)$
assert $int64\_test_{19}$: $(\min 12\ (10 : \text{INT}_{64}) = 10)$
assert $int64\_test_{20}$: $(\max 12\ (12 : \text{INT}_{64}) = 12)$
assert $int64\_test_{21}$: $(\max (-10)\ (12 : \text{INT}_{64}) = 12)$
assert $int64\_test_{22}$: $(\max 12\ (10 : \text{INT}_{64}) = 12)$
assert $int64\_test_{23}$: $(\text{succ } 12 = (13 : \text{INT}_{64}))$
assert $int64\_test_{24}$: $(\text{succ } 0 = (1 : \text{INT}_{64}))$
assert $int64\_test_{25}$: $(\text{pred } 12 = (11 : \text{INT}_{64}))$
assert $int64\_test_{26}$: $(\text{pred } 0 = -(1 : \text{INT}_{64}))$
assert $int64\_test_{27}$: $(\text{abs } 42 = (42 : \text{INT}_{64}))$
assert $int64\_test_{28}$: $(\text{abs } (-42) = (42 : \text{INT}_{64}))$

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 $integer\_test_{23}$: $(\text{succ } 12 = (13 : \mathbb{Z}))$
assert $integer\_test_{24}$: $(\text{succ } 0 = (1 : \mathbb{Z}))$
assert $integer\_test_{25}$: $(\text{pred } 12 = (11 : \mathbb{Z}))$
assert $integer\_test_{26}$: $(\text{pred } 0 = -(1 : \mathbb{Z}))$
assert $integer\_test_{27}$: $(\text{abs } 42 = (42 : \mathbb{Z}))$
assert $integer\_test_{28}$: $(\text{abs } (-42) = (42 : \mathbb{Z}))$

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

(*****************)
(* integerFrom... *)
(*****************)
```

val $integerFromInt$: $\text{INT} \rightarrow \mathbb{Z}$

declare *hol* target_rep function integerFromInt = `''` (* remove natFromNumeral, as it is the identify function *)
declare *ocaml* target_rep function integerFromInt = `'Big_int.big_int_of_int'`
declare *isabelle* target_rep function integerFromInt = `''`
declare *coq* target_rep function integerFromInt = `'id'`

assert *integer_from_int*$_0$: integerFromInt $0 = 0$
assert *integer_from_int*$_1$: integerFromInt $1 = 1$
assert *integer_from_int*$_2$: integerFromInt $(-2) = (-2)$

val *integerFromNat* : NAT $\rightarrow$ $\mathbb{Z}$
declare *hol* target_rep function integerFromNat = `'int_of_num'`
declare *ocaml* target_rep function integerFromNat = `'Big_int.big_int_of_int'`
declare *isabelle* target_rep function integerFromNat = `'int'`
declare *coq* target_rep function integerFromNat n = (`'Zpos'` (`'P_of_succ_nat'` n)) (* TODO: check *)

assert *integer_from_nat*$_0$: integerFromNat $0 = 0$
assert *integer_from_nat*$_1$: integerFromNat $1 = 1$
assert *integer_from_nat*$_2$: integerFromNat $12 = 12$

val *integerFromNatural* : $\mathbb{N}$ $\rightarrow$ $\mathbb{Z}$
declare *hol* target_rep function integerFromNatural = `'int_of_num'`
declare *ocaml* target_rep function integerFromNatural n = `''`n
declare *isabelle* target_rep function integerFromNatural = `'int'`
declare *coq* target_rep function integerFromNatural n = (`'Zpos'` (`'P_of_succ_nat'` n)) (* TODO: check *)

assert *integerFromNatural*$_0$: integerFromNatural $0 = 0$
assert *integerFromNatural*$_1$: integerFromNatural $822 = 822$
assert *integerFromNatural*$_2$: integerFromNatural $12 = 12$

val *integerFromInt*$_{32}$: INT$_{32}$ $\rightarrow$ $\mathbb{Z}$
declare *ocaml* target_rep function integerFromInt$_{32}$ = `'Big_int.big_int_of_int'`$_{32}$
declare *isabelle* target_rep function integerFromInt$_{32}$ = `'sint'`
declare *hol* target_rep function integerFromInt$_{32}$ = `'w2int'`
declare *coq* target_rep function integerFromInt$_{32}$ = `'TODO'`

assert *integer_from_int*$_{32}$_0 : integerFromInt$_{32}$ $0 = 0$
assert *integer_from_int*$_{32}$_1 : integerFromInt$_{32}$ $1 = 1$
assert *integer_from_int*$_{32}$_2 : integerFromInt$_{32}$ $123 = 123$
assert *integer_from_int*$_{32}$_3 : integerFromInt$_{32}$ $(-0) = -0$
assert *integer_from_int*$_{32}$_4 : integerFromInt$_{32}$ $(-1) = -1$
assert *integer_from_int*$_{32}$_5 : integerFromInt$_{32}$ $(-123) = -123$

val *integerFromInt*$_{64}$: INT$_{64}$ $\rightarrow$ $\mathbb{Z}$
declare *ocaml* target_rep function integerFromInt$_{64}$ = `'Big_int.big_int_of_int'`$_{64}$
declare *isabelle* target_rep function integerFromInt$_{64}$ = `'sint'`
declare *hol* target_rep function integerFromInt$_{64}$ = `'w2int'`
declare *coq* target_rep function integerFromInt$_{64}$ = `'TODO'`

assert *integer_from_int*$_{64}$_0 : integerFromInt$_{64}$ $0 = 0$
assert *integer_from_int*$_{64}$_1 : integerFromInt$_{64}$ $1 = 1$
assert *integer_from_int*$_{64}$_2 : integerFromInt$_{64}$ $123 = 123$
assert *integer_from_int*$_{64}$_3 : integerFromInt$_{64}$ $(-0) = -0$
assert *integer_from_int*$_{64}$_4 : integerFromInt$_{64}$ $(-1) = -1$

assert $integer\_from\_int_{64}\_5$: $\mathrm{integerFromInt}_{64}\ (-123) = -123$

```
(******************)
(* naturalFrom... *)
(******************)
```

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 $naturalFromInteger$: $\mathbb{Z}$ $\rightarrow$ $\mathbb{N}$
declare compile_message naturalFromInteger = "$naturalFromIntegerisundefinedfornegativeintegers$"

declare hol target_rep function naturalFromInteger i = `'Num'` (`'ABS'` i)
declare $ocaml$ target_rep function naturalFromInteger = `'Big_int.abs_big_int'`
declare coq target_rep function naturalFromInteger = `'Zabs_nat'`
declare $isabelle$ target_rep function naturalFromInteger i = `'nat'` (`'abs'` i)

assert $natural\_from\_integer_0$: naturalFromInteger $0 = 0$
assert $natural\_from\_integer_1$: naturalFromInteger $1 = 1$
assert $natural\_from\_integer_2$: naturalFromInteger $(-\ 2) = 2$

```
(******************)
(* intFrom ...    *)
(******************)
```

val $intFromInteger$: $\mathbb{Z}$ $\rightarrow$ INT
declare compile_message naturalFromInteger = "$naturalFromIntegerisundefinedfornegativeintegersandmightfailfornumb$

declare hol target_rep function intFromInteger = `''` `(* remove natFromNumeral, as it is the identify function *)`
declare $ocaml$ target_rep function intFromInteger = `'Big_int.int_of_big_int'`
declare $isabelle$ target_rep function intFromInteger = `''`
declare coq target_rep function intFromInteger = `'id'`

assert $int\_from\_integer_0$: intFromInteger $0 = 0$
assert $int\_from\_integer_1$: intFromInteger $1 = 1$
assert $int\_from\_integer_2$: intFromInteger $(-2) = (-2)$

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 $2 = 2$

```
(*****************)
(* natFrom ...    *)
(*****************)
```

val *natFromNatural* : $\mathbb{N} \rightarrow$ NAT
declare compile_message naturalFromInteger = "$natFromNaturalmightfailfortoobigvalues.Thevaluesallowedaresystem-dependend.However, atleast30bitshouldbeavailable, i.e.allnumbersupto2{\uparrow}30 = 1073741824shouldbeOK.$"
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 *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 $nat\_from\_int_0$: natFromInt $0 = 0$
assert $nat\_from\_int_1$: natFromInt $1 = 1$
assert $nat\_from\_int_2$: natFromInt $(-\ 2) = 2$

```
(*****************)
(* int32From ...  *)
(*****************)
```

val *int32FromNat* : NAT $\rightarrow$ INT_{32}
declare *hol* target_rep function int32FromNat n = ((`'n2w'` n) : INT_{32})
declare *ocaml* target_rep function int32FromNat = `'Int32.of_int'`
declare *coq* target_rep function int32FromNat n = (`'Zpos'` (`'P_of_succ_nat'` n)) `(* TODO check *)`
declare *isabelle* target_rep function int32FromNat n = ((`'word_of_int'` (`'int'` n)) : INT_{32})

assert $int32\_from\_nat_0$: int32FromNat $0 = 0$
assert $int32\_from\_nat_1$: int32FromNat $1 = 1$
assert $int32\_from\_nat_2$: int32FromNat $123 = 123$

val *int32FromNatural* : $\mathbb{N} \rightarrow$ INT_{32}
declare *hol* target_rep function int32FromNatural n = ((`'n2w'` n) : INT_{32})
declare *ocaml* target_rep function int32FromNatural = `'Big_int.int32_of_big_int'`
declare *coq* target_rep function int32FromNatural n = (`'Zpos'` (`'P_of_succ_nat'` n)) `(* TODO check *)`
declare *isabelle* target_rep function int32FromNatural n = ((`'word_of_int'` (`'int'` n)) : INT_{32})

assert $int32\_from\_natural_0$: int32FromNatural $0 = 0$
assert $int32\_from\_natural_1$: int32FromNatural $1 = 1$
assert $int32\_from\_natural_2$: int32FromNatural $123 = 123$

val *int32FromInteger* : $\mathbb{Z} \rightarrow$ INT_{32}
let *int32FromInteger* i = (
  let abs_int_{32} = int32FromNatural (naturalFromInteger i) in
  if $(i < 0)$ then $(-\ abs\_int_{32})$ else abs_int_{32}
)

declare *ocaml* target_rep function int32FromInteger $=$ `'Big_int.int32_of_big_int'`
declare *isabelle* target_rep function int32FromInteger i $=$ $(($`'word_of_int'` $i)$: $\text{INT}_{32})$

assert $int32\_from\_integer_0$: int32FromInteger $0 = 0$
assert $int32\_from\_integer_1$: int32FromInteger $1 = 1$
assert $int32\_from\_integer_2$: int32FromInteger $123 = 123$
assert $int32\_from\_integer_3$: int32FromInteger $(-0) = -0$
assert $int32\_from\_integer_4$: int32FromInteger $(-1) = -1$
assert $int32\_from\_integer_5$: int32FromInteger $(-123) = -123$

val *int32FromInt* : $\text{INT} \rightarrow \text{INT}_{32}$
let *int32FromInt* i $=$ int32FromInteger (integerFromInt i)
declare *ocaml* target_rep function int32FromInt $=$ `'Int32.of_int'`
declare *isabelle* target_rep function int32FromInt i $=$ $(($`'word_of_int'` $i)$: $\text{INT}_{32})$

assert $int32\_from\_int_0$: int32FromInt $0 = 0$
assert $int32\_from\_int_1$: int32FromInt $1 = 1$
assert $int32\_from\_int_2$: int32FromInt $123 = 123$
assert $int32\_from\_int_3$: int32FromInt $(-0) = -0$
assert $int32\_from\_int_4$: int32FromInt $(-1) = -1$
assert $int32\_from\_int_5$: int32FromInt $(-123) = -123$

val $int32FromInt_{64}$: $\text{INT}_{64} \rightarrow \text{INT}_{32}$
let $int32FromInt_{64}$ i $=$ int32FromInteger (integerFromInt$_{64}$ i)
declare *ocaml* target_rep function int32FromInt$_{64}$ $=$ `'Int64.to_int'`$_{32}$
declare *hol* target_rep function int32FromInt$_{64}$ i $=$ $(($`'sw2sw'` $i)$: $\text{INT}_{32})$
declare *isabelle* target_rep function int32FromInt$_{64}$ i $=$ $(($`'scast'` $i)$: $\text{INT}_{32})$

assert $int32\_from\_int_{64}\_0$: int32FromInt$_{64}$ $0 = 0$
assert $int32\_from\_int_{64}\_1$: int32FromInt$_{64}$ $1 = 1$
assert $int32\_from\_int_{64}\_2$: int32FromInt$_{64}$ $123 = 123$
assert $int32\_from\_int_{64}\_3$: int32FromInt$_{64}$ $(-0) = -0$
assert $int32\_from\_int_{64}\_4$: int32FromInt$_{64}$ $(-1) = -1$
assert $int32\_from\_int_{64}\_5$: int32FromInt$_{64}$ $(-123) = -123$

```
(*****************)
(* int64From ...  *)
(*****************)
```

val *int64FromNat* : $\text{NAT} \rightarrow \text{INT}_{64}$
declare *hol* target_rep function int64FromNat n $=$ $(($`'n2w'` $n)$: $\text{INT}_{64})$
declare *ocaml* target_rep function int64FromNat $=$ `'Int64.of_int'`
declare *coq* target_rep function int64FromNat n $=$ (`'Zpos'` (`'P_of_succ_nat'` n)) `(* TODO check *)`
declare *isabelle* target_rep function int64FromNat n $=$ $(($`'word_of_int'` (`'int'` n)) : $\text{INT}_{64})$

assert $int64\_from\_nat_0$: int64FromNat $0 = 0$
assert $int64\_from\_nat_1$: int64FromNat $1 = 1$
assert $int64\_from\_nat_2$: int64FromNat $123 = 123$

val *int64FromNatural* : $\mathbb{N} \rightarrow \text{INT}_{64}$
declare *hol* target_rep function int64FromNatural n $=$ $(($`'n2w'` $n)$: $\text{INT}_{64})$
declare *ocaml* target_rep function int64FromNatural $=$ `'Big_int.int64_of_big_int'`
declare *coq* target_rep function int64FromNatural n $=$ (`'Zpos'` (`'P_of_succ_nat'` n)) `(* TODO check *)`

declare *isabelle* target_rep function int64FromNatural n = (('word_of_int' ('int' n)) : INT_{64})

assert $\mathit{int64\_from\_natural}_0$: int64FromNatural 0 = 0
assert $\mathit{int64\_from\_natural}_1$: int64FromNatural 1 = 1
assert $\mathit{int64\_from\_natural}_2$: int64FromNatural 123 = 123

val *int64FromInteger* : $\mathbb{Z} \rightarrow \text{INT}_{64}$
let *int64FromInteger* i = (
  let $\mathit{abs\_int}_{64}$ = int64FromNatural (naturalFromInteger i) in
  if $(i < 0)$ then $(-\ \mathit{abs\_int}_{64})$ else $\mathit{abs\_int}_{64}$
)

declare *ocaml* target_rep function int64FromInteger = 'Big_int.int64_of_big_int'
declare *isabelle* target_rep function int64FromInteger i = (('word_of_int' i) : INT_{64})

assert $\mathit{int64\_from\_integer}_0$: int64FromInteger 0 = 0
assert $\mathit{int64\_from\_integer}_1$: int64FromInteger 1 = 1
assert $\mathit{int64\_from\_integer}_2$: int64FromInteger 123 = 123
assert $\mathit{int64\_from\_integer}_3$: int64FromInteger $(-0) = -0$
assert $\mathit{int64\_from\_integer}_4$: int64FromInteger $(-1) = -1$
assert $\mathit{int64\_from\_integer}_5$: int64FromInteger $(-123) = -123$

val *int64FromInt* : $\text{INT} \rightarrow \text{INT}_{64}$
let *int64FromInt* i = int64FromInteger (integerFromInt i)
declare *ocaml* target_rep function int64FromInt = 'Int64.of_int'
declare *isabelle* target_rep function int64FromInt i = (('word_of_int' i) : INT_{64})

assert $\mathit{int64\_from\_int}_0$: int64FromInt 0 = 0
assert $\mathit{int64\_from\_int}_1$: int64FromInt 1 = 1
assert $\mathit{int64\_from\_int}_2$: int64FromInt 123 = 123
assert $\mathit{int64\_from\_int}_3$: int64FromInt $(-0) = -0$
assert $\mathit{int64\_from\_int}_4$: int64FromInt $(-1) = -1$
assert $\mathit{int64\_from\_int}_5$: int64FromInt $(-123) = -123$

val $\mathit{int64FromInt}_{32}$: $\text{INT}_{32} \rightarrow \text{INT}_{64}$
let $\mathit{int64FromInt}_{32}$ i = int64FromInteger ($\text{integerFromInt}_{32}$ i)
declare *ocaml* target_rep function int64FromInt_{32} = $\texttt{'Int64.of\_int'}_{32}$
declare *hol* target_rep function int64FromInt_{32} i = (('sw2sw' i) : INT_{64})
declare *isabelle* target_rep function int64FromInt_{32} i = (('scast' i) : INT_{64})

assert $\mathit{int64\_from\_int}_{33}\_0$: int64FromInt_{32} 0 = 0
assert $\mathit{int64\_from\_int}_{32}\_1$: int64FromInt_{32} 1 = 1
assert $\mathit{int64\_from\_int}_{32}\_2$: int64FromInt_{32} 123 = 123
assert $\mathit{int64\_from\_int}_{32}\_3$: int64FromInt_{32} $(-0) = -0$
assert $\mathit{int64\_from\_int}_{32}\_4$: int64FromInt_{32} $(-1) = -1$
assert $\mathit{int64\_from\_int}_{32}\_5$: int64FromInt_{32} $(-123) = -123$

```
(*****************)
(* what's missing *)
(*****************)
```

val *naturalFromInt* : $\text{INT} \rightarrow \mathbb{N}$
val $\mathit{naturalFromInt}_{32}$: $\text{INT}_{32} \rightarrow \mathbb{N}$
val $\mathit{naturalFromInt}_{64}$: $\text{INT}_{64} \rightarrow \mathbb{N}$

let inline *naturalFromInt* i = naturalFromNat (natFromInt i)

let inline $\mathit{naturalFromInt}_{32}\ i\ =\ \mathrm{naturalFromInteger}\ (\mathrm{integerFromInt}_{32}\ i)$
let inline $\mathit{naturalFromInt}_{64}\ i\ =\ \mathrm{naturalFromInteger}\ (\mathrm{integerFromInt}_{64}\ i)$

assert $\mathit{natural\_from\_int}_0$: $\mathrm{naturalFromInt}\ 0 = 0$
assert $\mathit{natural\_from\_int}_1$: $\mathrm{naturalFromInt}\ 1 = 1$
assert $\mathit{natural\_from\_int}_2$: $\mathrm{naturalFromInt}\ (-\ 2) = 2$
assert $\mathit{natural\_from\_int}_{32}\_0$: $\mathrm{naturalFromInt}_{32}\ 0 = 0$
assert $\mathit{natural\_from\_int}_{32}\_1$: $\mathrm{naturalFromInt}_{32}\ 1 = 1$
assert $\mathit{natural\_from\_int}_{32}\_2$: $\mathrm{naturalFromInt}_{32}\ (-\ 2) = 2$
assert $\mathit{natural\_from\_int}_{64}\_0$: $\mathrm{naturalFromInt}_{64}\ 0 = 0$
assert $\mathit{natural\_from\_int}_{64}\_1$: $\mathrm{naturalFromInt}_{64}\ 1 = 1$
assert $\mathit{natural\_from\_int}_{64}\_2$: $\mathrm{naturalFromInt}_{64}\ (-\ 2) = 2$

val $\mathit{intFromNatural}\ :\ \mathbb{N}\ \rightarrow\ \textsc{int}$
val $\mathit{intFromInt}_{32}\ :\ \textsc{int}_{32}\ \rightarrow\ \textsc{int}$
val $\mathit{intFromInt}_{64}\ :\ \textsc{int}_{64}\ \rightarrow\ \textsc{int}$

let inline $\mathit{intFromNatural}\ n\ =\ \mathrm{intFromNat}\ (\mathrm{natFromNatural}\ n)$
let inline $\mathit{intFromInt}_{32}\ i\ =\ \mathrm{intFromInteger}\ (\mathrm{integerFromInt}_{32}\ i)$
let inline $\mathit{intFromInt}_{64}\ i\ =\ \mathrm{intFromInteger}\ (\mathrm{integerFromInt}_{64}\ i)$

assert $\mathit{int\_from\_natural}_0$: $\mathrm{intFromNatural}\ 0 = 0$
assert $\mathit{int\_from\_natural}_1$: $\mathrm{intFromNatural}\ 1 = 1$
assert $\mathit{int\_from\_natural}_2$: $\mathrm{intFromNatural}\ 122 = 122$
assert $\mathit{int\_from\_int}_{32}\_0$: $\mathrm{intFromInt}_{32}\ 0 = 0$
assert $\mathit{int\_from\_int}_{32}\_1$: $\mathrm{intFromInt}_{32}\ 1 = 1$
assert $\mathit{int\_from\_int}_{32}\_2$: $\mathrm{intFromInt}_{32}\ (-\ 2) = (-2)$
assert $\mathit{int\_from\_int}_{64}\_0$: $\mathrm{intFromInt}_{64}\ 0 = 0$
assert $\mathit{int\_from\_int}_{64}\_1$: $\mathrm{intFromInt}_{64}\ 1 = 1$
assert $\mathit{int\_from\_int}_{64}\_2$: $\mathrm{intFromInt}_{64}\ (-\ 2) = (-2)$

val $\mathit{natFromInteger}\ :\ \mathbb{Z}\ \rightarrow\ \textsc{nat}$
val $\mathit{natFromInt}_{32}\ :\ \textsc{int}_{32}\ \rightarrow\ \textsc{nat}$
val $\mathit{natFromInt}_{64}\ :\ \textsc{int}_{64}\ \rightarrow\ \textsc{nat}$

let inline $\mathit{natFromInteger}\ n\ =\ \mathrm{natFromInt}\ (\mathrm{intFromInteger}\ n)$
let inline $\mathit{natFromInt}_{32}\ i\ =\ \mathrm{natFromInteger}\ (\mathrm{integerFromInt}_{32}\ i)$
let inline $\mathit{natFromInt}_{64}\ i\ =\ \mathrm{natFromInteger}\ (\mathrm{integerFromInt}_{64}\ i)$

assert $\mathit{nat\_from\_integer}_0$: $\mathrm{natFromInteger}\ 0 = 0$
assert $\mathit{nat\_from\_integer}_1$: $\mathrm{natFromInteger}\ 1 = 1$
assert $\mathit{nat\_from\_integer}_2$: $\mathrm{natFromInteger}\ 122 = 122$
assert $\mathit{nat\_from\_int}_{32}\_0$: $\mathrm{natFromInt}_{32}\ 0 = 0$
assert $\mathit{nat\_from\_int}_{32}\_1$: $\mathrm{natFromInt}_{32}\ 1 = 1$
assert $\mathit{nat\_from\_int}_{32}\_2$: $\mathrm{natFromInt}_{32}\ (-\ 2) = 2$
assert $\mathit{nat\_from\_int}_{64}\_0$: $\mathrm{natFromInt}_{64}\ 0 = 0$
assert $\mathit{nat\_from\_int}_{64}\_1$: $\mathrm{natFromInt}_{64}\ 1 = 1$
assert $\mathit{nat\_from\_int}_{64}\_2$: $\mathrm{natFromInt}_{64}\ (-\ 2) = 2$

val $\mathit{string\_of\_natural}\ :\ \mathbb{N}\ \rightarrow\ \textsc{string}$
declare *ocaml* target_rep function string_of_natural = `'Big_int.string_of_big_int'`
let inline $\{\mathit{coq}\}\ \mathit{string\_of\_natural}\ n\ =\ "\mathit{TODO}"$

6 Function_extra

declare $\{isabelle, hol, ocaml, coq\}$ rename module = lem_function_extra

open import *Maybe Bool Basic_classes Num Function*

open import $\{hol\}$ *lemTheory*
open import $\{isabelle\}$ *\$LIB_DIR/Lem*

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

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

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

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

assert $comb_0$: $(\mathrm{comb}$ (fun $(x : \text{NAT}) \rightarrow 3 * x)\ \mathrm{succ}\ 2 = 9)$
assert $comb_1$: $(\mathrm{comb}\ \mathrm{succ}$ (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 $(x : \text{NAT})\ y \rightarrow x - y)\ 5\ 3 = 0$

```
(* ------------------------------------ *)
(* failing with a proper error message  *)
(* ------------------------------------ *)
```

val *failwith* : $\forall\ \alpha.\ \text{STRING} \rightarrow \alpha$
declare *ocaml* target_rep function failwith = `'failwith'`
declare *hol* target_rep function failwith = `'failwith'`
declare *isabelle* target_rep function failwith = `'failwith'`
declare *coq* target_rep function failwith s = `'DAEMON'`

```
(* ----------------------- *)
(* 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 $\sim\{coq\}$ *THE_spec* : $(\forall\ p\ x.\ (\mathrm{THE}\ p = \mathrm{Just}\ x) \longleftrightarrow ((p\ x) \wedge (\forall\ y.\ p\ y \longrightarrow (x = y))))$

7 Tuple

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

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

declare {*isabelle*, *hol*, *ocaml*, *coq*} 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 $\mathrm{uncurry}$ = `'UNCURRY'`
declare $isabelle$ target_rep function $\mathrm{uncurry}$ = `'split'`
declare $ocaml$ target_rep function $\mathrm{uncurry}$ = `'Lem.uncurry'`
declare coq target_rep function $\mathrm{uncurry}$ = `'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 $uncurry_1$: $(\mathrm{uncurry}$ (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 $swap_1$: $(\mathrm{swap}$ (false, true) = (true, false))

8 List

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



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

declare {*isabelle*, *ocaml*, *hol*, *coq*} rename module = lem_list

open import *Bool Maybe Basic_classes Tuple Num*

open import {*coq*} *Coq.Lists.List*
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 : ∀ α. LIST α → NAT
let rec length l =
  match l with
    | [] → 0
    | x :: xs → 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 isabelle target_rep function length = 'List.length'
declare coq target_rep function length = 'List.length'

assert length_0 : (length ([] : LIST NAT) = 0)
assert length_1 : (length ([2] : LIST NAT) = 1)
assert length_2 : (length ([2; 3] : LIST NAT) = 2)

lemma length_spec : ((length [] = 0) ∧ (∀ x xs. length (x :: xs) = length xs + 1))

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

val listEqual : ∀ α. Eq α ⇒ LIST α → LIST α → 𝔹
val listEqualBy : ∀ α. (α → α → 𝔹) → LIST α → LIST α → 𝔹

let rec listEqualBy eq l1 l2 = match (l1, l2) with
  | ([], []) → true
  | ([], (_ :: _)) → false
  | ((_ :: _), []) → false
  | (x :: xs, y :: ys) → (eq x y ∧ listEqualBy eq xs ys)
end
declare termination_argument listEqualBy = automatic

let inline listEqual = listEqualBy (=)
declare hol target_rep function listEqual = infix '='
declare isabelle target_rep function listEqual = infix '='
declare coq target_rep function listEqualBy = 'list_equal_by'

instance ∀ α. Eq α ⇒ (Eq (LIST α))
  let = = listEqual
  let <> l1 l2 = ¬ (listEqual l1 l2)
end


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

val lexicographicCompare : ∀ α. Ord α ⇒ LIST α → LIST α → 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_ord4 : ([2] ≤ [(2 : NAT)])
assert list_ord5 : ([2; 3] > [(2 : 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 $++$: $\forall\, \alpha.$ LIST α $\to$ LIST α $\to$ LIST α `(* originally append *)`
let rec $++$ xs ys = match xs with
  | [] $\to$ ys
  | x :: xs' $\to$ $x :: (\text{append } xs'\ ys)$
  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 *tex* target_rep function append = infix `'$+\!+$'`
declare *coq* target_rep function append = (`'@'` `'List.app'` `'_'`)

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 $snoc$: $\forall\, \alpha.\ \alpha \to$ LIST α $\to$ LIST α
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 : $\forall\, \alpha\, \beta.\ (\alpha \to \beta) \to$ LIST α $\to$ LIST β
let rec $map\ f\ l$ = match l with
  | [] $\to$ []
  | x :: xs $\to$ $(f\ x) :: \text{map } f\ xs$
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 : (map (fun x → x + (1 : NAT)) [] = [])
assert map_1 : (map (fun x → x + (1 : NAT)) [0] = [1])
assert map_4 : (map (fun x → x + (1 : 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 : ∀ α. LIST α → LIST α → LIST α (* originally named rev_append *)
let rec reverseAppend l_1 l_2 = match l_1 with
                       | [] → l_2
                       | x :: xs → 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 : (reverseAppend [(0 : NAT); 1; 2; 3] [4; 5] = [3; 2; 1; 0; 4; 5])

(* Reversing a list *)
val reverse : ∀ α. LIST α → LIST α (* 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 : (reverse ([] : LIST NAT) = [])
assert reverse_1 : (reverse [(1 : NAT)] = [1])
assert reverse_2 : (reverse [(1 : NAT); 2] = [2; 1])
assert reverse_5 : (reverse [(1 : NAT); 2; 3; 4; 5] = [5; 4; 3; 2; 1])

lemma reverseAppend : (∀ l_1 l_2. reverseAppend l_1 l_2 = (++) (reverse l_1) l_2)
let inline {isabelle} reverseAppend l_1 l_2 = ((reverse l_1) ++ l_2)


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

val reverseMap : ∀ α β. (α → β) → LIST α → LIST β
let inline reverseMap f l = reverse (map f l)

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

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

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

val *foldl* : $\forall\ \alpha\ \beta.\ (\alpha\ \rightarrow\ \beta\ \rightarrow\ \alpha)\ \rightarrow\ \alpha\ \rightarrow\ \text{LIST}\ \beta\ \rightarrow\ \alpha$ `(* originally foldl *)`

let rec *foldl* $f\ b\ l\ =$ match l with
  | [] $\rightarrow\ b$
  | $x\ ::\ xs\ \rightarrow$ 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 (+) $(0 : \text{NAT})$ [] = 0)
assert $foldl_1$: (foldl (+) $(0 : \text{NAT})$ [4] = 4)
assert $foldl_4$: (foldl (fun $l\ e\ \rightarrow\ e{::}l$) [] $[(1 : \text{NAT}); 2; 3; 4] = [4; 3; 2; 1]$)

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

val *foldr* : $\forall\ \alpha\ \beta.\ (\alpha\ \rightarrow\ \beta\ \rightarrow\ \beta)\ \rightarrow\ \beta\ \rightarrow\ \text{LIST}\ \alpha\ \rightarrow\ \beta$ `(* originally foldr with different argument order *)`
let rec *foldr* $f\ b\ l\ =$ match l with
  | [] $\rightarrow\ b$
  | $x\ ::\ xs\ \rightarrow\ f\ x$ (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 (+) $(0 : \text{NAT})$ [] = 0)
assert $foldr_1$: (foldr (+) 1 $[(4 : \text{NAT})] = 5$)
assert $foldr_4$: (foldr (fun $e\ l\ \rightarrow\ e{::}l$) [] $[(1 : \text{NAT}); 2; 3; 4] = [1; 2; 3; 4]$)

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

val *concat* : $\forall\ \alpha.\ \text{LIST}\ (\text{LIST}\ \alpha)\ \rightarrow\ \text{LIST}\ \alpha$ `(* 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'`

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

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

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

val *concatMap* : $\forall\ \alpha\ \beta.\ (\alpha\ \rightarrow$ LIST $\beta)\ \rightarrow$ LIST $\alpha\ \rightarrow$ LIST β
let inline *concatMap* f l = concat (map f l)

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

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

val *all* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow$ LIST $\alpha\ \rightarrow\ \mathbb{B}$ `(* originally for_all *)`
let *all* P l = foldl (fun r e $\rightarrow$ 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 $\rightarrow$ $x >$ (2 : NAT)) [])
assert all_4 : (all (fun x $\rightarrow$ $x >$ (2 : NAT)) [4; 5; 6; 7])
assert *all_4_neg* : ($\neg$ (all (fun x $\rightarrow$ $x >$ (2 : 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 $e \wedge$ all P l))

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

val *any* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow$ LIST $\alpha\ \rightarrow\ \mathbb{B}$ `(* originally exist *)`
let *any* P l = foldl (fun r e $\rightarrow$ 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 = ($\exists\ x \in$ (`'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 : $(\forall\ P\ e\ l.\ \text{any}\ P\ (e{::}l) = (P\ e \lor \text{any}\ P\ l))$

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

(* get the initial part and the last element of the list in a safe way *)
```

val $dest\_init$: $\forall\ \alpha.\ \text{LIST}\ \alpha\ \rightarrow\ \text{MAYBE}\ (\text{LIST}\ \alpha\ *\ \alpha)$

let rec $dest\_init\_aux\ rev\_init\ last\_elem\_seen\ to\_process$ =
  match $to\_process$ with
    | $[]\ \rightarrow\ (\text{reverse}\ rev\_init,\ \ last\_elem\_seen)$
    | $x :: xs\ \rightarrow\ \text{dest\_init\_aux}\ (last\_elem\_seen{::}rev\_init)\ x\ xs$
  end
declare termination_argument dest_init_aux = automatic

let $dest\_init\ l$ = match l with
  | $[]\ \rightarrow\ \text{Nothing}$
  | $x :: xs\ \rightarrow\ \text{Just}\ (\text{dest\_init\_aux}\ []\ x\ xs)$
end

assert $dest\_init_0$: $(\text{dest\_init}\ ([] : \text{LIST NAT}) = \text{Nothing})$
assert $dest\_init_1$: $(\text{dest\_init}\ [(1 : \text{NAT})] = \text{Just}\ ([],\ \ 1))$
assert $dest\_init_2$: $(\text{dest\_init}\ [(1 : \text{NAT}); 2; 3; 4; 5] = \text{Just}\ ([1; 2; 3; 4],\ \ 5))$

lemma $dest\_init\_nil$: $(\text{dest\_init}\ [] = \text{Nothing})$
lemma $dest\_init\_snoc$: $(\forall\ x\ xs.\ \text{dest\_init}\ (xs ++ [x]) = \text{Just}\ (xs,\ \ x))$

```
(* ========================================================================== *)
(* 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
  | $[]\ \rightarrow\ \text{Nothing}$
  | $x\ ::\ xs\ \rightarrow$ if $n = 0$ then $\text{Just}\ x$ else $\text{index}\ xs\ (n{-}1)$
end

declare termination_argument index = automatic

declare *isabelle* target_rep function index = `'index'`
declare {*ocaml*, *hol*} rename function index = list_index

assert $index_0$: $(\text{index}\ [(0 : \text{NAT}); 1; 2; 3; 4; 5]\ 0 = \text{Just}\ 0)$

assert $index_1$: (index [(0 : NAT); 1; 2; 3; 4; 5] 1 = Just 1)
assert $index_2$: (index [(0 : NAT); 1; 2; 3; 4; 5] 2 = Just 2)
assert $index_3$: (index [(0 : NAT); 1; 2; 3; 4; 5] 3 = Just 3)
assert $index_4$: (index [(0 : NAT); 1; 2; 3; 4; 5] 4 = Just 4)
assert $index_5$: (index [(0 : NAT); 1; 2; 3; 4; 5] 5 = Just 5)
assert $index_6$: (index [(0 : 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* : $\forall\ \alpha.\ (\alpha \to \mathbb{B}) \to$ LIST $\alpha \to$ LIST NAT

let rec *findIndices_aux* $(i : \text{NAT})\ P\ l$ =
  match l with
    | [] → []
    | x :: xs → 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 $(n : \text{NAT}) \to n > 3$) [] = [])
assert $findIndices_2$: (findIndices (fun $(n : \text{NAT}) \to n > 3$) [4] = [0])
assert $findIndices_3$: (findIndices (fun $(n : \text{NAT}) \to n > 3$) [1; 5; 3; 1; 2; 6] = [1; 5])

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

(* findIndex returns the first index of a list that satisfies a given predicate. *)
```
val *findIndex* : $\forall\ \alpha.\ (\alpha \to \mathbb{B}) \to$ LIST $\alpha \to$ 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 $(n : \text{NAT}) \to n > 3$) [1; 2] = Nothing)
assert $find\_index_1$: (findIndex (fun $(n : \text{NAT}) \to n > 3$) [1; 2; 4] = Just 2)
assert $find\_index_2$: (findIndex (fun $(n : \text{NAT}) \to n > 3$) [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 : ∀ α. (NAT → α) → NAT → LIST α


let rec genlist f n =
  match n with
    | 0 → []
    | n′ + 1 → snoc (f n′) (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 : (∀ f n. (length (genlist f n) = n))
lemma genlist_index : (∀ f n i. i < n ⟶ index (genlist f n) i = Just (f i))


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


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

val replicate : ∀ α. NAT → α → LIST α
let rec replicate n x =
  match n with
```

```
    | 0 → []
    | n' + 1 → x :: 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 : NAT) = [])
assert replicate_1 : (replicate 1 (2 : NAT) = [2])
assert replicate_2 : (replicate 2 (2 : NAT) = [2; 2])
assert replicate_3 : (replicate 3 (2 : NAT) = [2; 2; 2])
lemma replicate_length : (∀ n x. (length (replicate n x) = n))
lemma replicate_index : (∀ n x i. i < n ⟶ index (replicate n x) i = 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 (l_1, l_2) = splitAt (n−1) xs in
         (x::l_1, l_2)
       end
  end
declare termination_argument splitAt = automatic

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


assert splitAt_1 : (splitAt 0 [(1 : NAT); 2; 3; 4; 5; 6] = ([], [1; 2; 3; 4; 5; 6]))
assert splitAt_2 : (splitAt 2 [(1 : NAT); 2; 3; 4; 5; 6] = ([1; 2], [3; 4; 5; 6]))
assert splitAt_3 : (splitAt 100 [(1 : NAT); 2; 3; 4; 5; 6] = ([1; 2; 3; 4; 5; 6], []))

lemma splitAt_append : (∀ n xs.
  let (xs_1, xs_2) = splitAt n xs in
  (xs = xs_1 ++ xs_2))

lemma splitAt_length : (∀ n xs.
  let (xs_1, xs_2) = splitAt n xs in
  ((length xs_1 = n) ∨
   ((length xs_1 = length xs) ∧ null xs_2)))
```

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

(* take n xs returns the prefix of xs of length n, or xs itself if n > length xs *)
```
val $take$: $\forall\ \alpha.\ \text{NAT} \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$
let $take\ n\ l\ =\ \text{fst}\ (\text{splitAt}\ n\ l)$

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

assert $take_1$: $(\text{take}\ 0\ [(1 : \text{NAT}); 2; 3; 4; 5; 6] = [])$
assert $take_2$: $(\text{take}\ 2\ [(1 : \text{NAT}); 2; 3; 4; 5; 6] = [1; 2])$
assert $take_3$: $(\text{take}\ 100\ [(1 : \text{NAT}); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])$

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

(* [drop n xs] drops the first [n] elements of [xs]. It returns the empty list, if [n] > [length
xs]. *)
```
val $drop$: $\forall\ \alpha.\ \text{NAT} \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$
let $drop\ n\ l\ =\ \text{snd}\ (\text{splitAt}\ n\ l)$

declare hol target_rep function drop = `'DROP'`
declare $isabelle$ target_rep function drop = `'List.drop'`

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

lemma $splitAt\_take\_drop$: $(\forall\ n\ xs.\ \text{splitAt}\ n\ xs = (\text{take}\ n\ xs,\ \text{drop}\ n\ xs))$

let inline $\{hol\}$ $splitAt\ n\ xs\ =\ (\text{take}\ n\ xs,\ \text{drop}\ n\ xs)$

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

(* [dropWhile p xs] drops the first elements of [xs] that satisfy [p]. *)
```
val $dropWhile$: $\forall\ \alpha.\ (\alpha \rightarrow \mathbb{B}) \rightarrow \text{LIST}\ \alpha \rightarrow \text{LIST}\ \alpha$
let rec $dropWhile\ p\ l$ = match l with
  | $[] \rightarrow []$
  | $x :: xs \rightarrow$ if $p\ x$ then $\text{dropWhile}\ p\ xs$ else l
end
declare termination_argument dropWhile = automatic

assert $dropWhile_0$: $(\text{dropWhile}\ ((>)\ 3)\ [(1 : \text{NAT}); 2; 3; 4; 5; 6] = [3; 4; 5; 6])$
assert $dropWhile_1$: $(\text{dropWhile}\ ((\geq)\ 5)\ [(1 : \text{NAT}); 2; 3; 4; 5; 6] = [6])$
assert $dropWhile_2$: $(\text{dropWhile}\ ((>)\ 100)\ [(1 : \text{NAT}); 2; 3; 4; 5; 6] = [])$
assert $dropWhile_3$: $(\text{dropWhile}\ ((<)\ 10)\ [(1 : \text{NAT}); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])$

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

```
(* [takeWhile p xs] takes the first elements of [xs] that satisfy [p]. *)
```
val $takeWhile\ :\ \forall\ \alpha.\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \textsc{list}\ \alpha\ \rightarrow\ \textsc{list}\ \alpha$
let rec $takeWhile\ p\ l\ =$ match l with
  | $[]\ \rightarrow\ []$
  | $x :: xs\ \rightarrow$ if $p\ x$ then x::takeWhile $p\ xs$ else $[]$
end
declare termination_argument takeWhile = automatic

assert $takeWhile_0$: $(\text{takeWhile}\ ((>)\ 3)\ [(1 : \textsc{nat}); 2; 3; 4; 5; 6] = [1; 2])$
assert $takeWhile_1$: $(\text{takeWhile}\ ((\geq)\ 5)\ [(1 : \textsc{nat}); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5])$
assert $takeWhile_2$: $(\text{takeWhile}\ ((>)\ 100)\ [(1 : \textsc{nat}); 2; 3; 4; 5; 6] = [1; 2; 3; 4; 5; 6])$
assert $takeWhile_3$: $(\text{takeWhile}\ ((<)\ 10)\ [(1 : \textsc{nat}); 2; 3; 4; 5; 6] = [])$

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

val $isPrefixOf\ :\ \forall\ \alpha.\ Eq\ \alpha\ \Rightarrow\ \textsc{list}\ \alpha\ \rightarrow\ \textsc{list}\ \alpha\ \rightarrow\ \mathbb{B}$
let rec $isPrefixOf\ l_1\ l_2\ =$ match $(l_1,\ l_2)$ with
  | $([],\ \_)\ \rightarrow$ true
  | $(\_ :: \_,\ [])\ \rightarrow$ false
  | $(x :: xs,\ y :: ys)\ \rightarrow\ (x = y) \wedge \text{isPrefixOf}\ xs\ ys$
end
declare termination_argument isPrefixOf = automatic

declare *hol* target_rep function isPrefixOf = `'isPREFIX'`

assert $isPrefixOf_0$: $(\text{isPrefixOf}\ []\ [(0 : \textsc{nat}); 1; 2; 3; 4])$
assert $isPrefixOf_1$: $(\text{isPrefixOf}\ [0]\ [(0 : \textsc{nat}); 1; 2; 3; 4])$
assert $isPrefixOf_2$: $(\text{isPrefixOf}\ [0; 1; 2]\ [(0 : \textsc{nat}); 1; 2; 3; 4])$
assert $isPrefixOf_3$: $\neg\ (\text{isPrefixOf}\ [0; 2]\ [(0 : \textsc{nat}); 1; 2; 3; 4])$
assert $isPrefixOf_4$: $\neg\ (\text{isPrefixOf}\ [(0 : \textsc{nat}); 1; 2; 3; 4]\ [])$

lemma $isPrefixOf\_alt\_def\ :\ \forall\ l_1\ l_2.\ \text{isPrefixOf}\ l_1\ l_2 \longleftrightarrow (\exists\ l_3.\ l_2 = (l_1\ +\!\!+\ l_3))$
lemma $isPrefixOf\_sym\ :\ \forall\ l.\ \text{isPrefixOf}\ l\ l$
lemma $isPrefixOf\_trans\ :\ \forall\ l_1\ l_2\ l_3.\ \text{isPrefixOf}\ l_1\ l_2 \longrightarrow \text{isPrefixOf}\ l_2\ l_3 \longrightarrow \text{isPrefixOf}\ l_1\ l_3$
lemma $isPrefixOf\_antisym\ :\ \forall\ l_1\ l_2.\ \text{isPrefixOf}\ l_1\ l_2 \longrightarrow \text{isPrefixOf}\ l_2\ l_1 \longrightarrow (l_1 = l_2)$

```
(* ------------------------- *)
(* update                    *)
(* ------------------------- *)
```
val $update\ :\ \forall\ \alpha.\ \textsc{list}\ \alpha\ \rightarrow\ \textsc{nat}\ \rightarrow\ \alpha\ \rightarrow\ \textsc{list}\ \alpha$
let rec $update\ l\ n\ e\ =$
  match l with
    | $[]\ \rightarrow\ []$
    | $x\ ::\ xs\ \rightarrow$ if $n = 0$ then $e :: xs$ else $x :: (\text{update}\ xs\ (n - 1)\ e)$
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 $list\_update_1$: $(\text{update}\ []\ 2\ (3 : \textsc{nat}) = [])$
assert $list\_update_2$: $(\text{update}\ [1; 2; 3; 4; 5]\ 0\ (0 : \textsc{nat}) = [0; 2; 3; 4; 5])$
assert $list\_update_3$: $(\text{update}\ [1; 2; 3; 4; 5]\ 1\ (0 : \textsc{nat}) = [1; 0; 3; 4; 5])$

assert $list\_update_4$: (update [1; 2; 3; 4; 5] 2 (0 : NAT) = [1; 2; 0; 4; 5])
assert $list\_update_5$: (update [1; 2; 3; 4; 5] 5 (0 : 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* : $\forall$ α. *Eq* α $\Rightarrow$ α $\rightarrow$ LIST α $\rightarrow$ $\mathbb{B}$
val *elemBy* : $\forall$ α. (α $\rightarrow$ α $\rightarrow$ $\mathbb{B}$) $\rightarrow$ α $\rightarrow$ LIST α $\rightarrow$ $\mathbb{B}$

let *elemBy* *eq* *e* *l* = any (*eq* *e*) *l*
let *elem* = elemBy (=)

declare *hol* target_rep function elem = `'MEM'`
declare *ocaml* target_rep function elem = `'List.mem'`
declare *isabelle* target_rep function elem e l = `'Set.member'` e (`'set'` l)

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

lemma *elem_spec* : (($\forall$ e. $\neg$ (elem e [])) $\wedge$
  ($\forall$ e x xs. (elem e (x :: xs)) = (($e = x$) $\vee$ (elem e xs))))

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

val *find* : $\forall$ α. (α $\rightarrow$ $\mathbb{B}$) $\rightarrow$ LIST α $\rightarrow$ MAYBE α `(* previously not of maybe type *)`
let rec *find* P l = match l with
  | [] $\rightarrow$ Nothing

```
  | x :: xs → if P x then Just x else find P xs
end
declare termination_argument find = automatic

declare isabelle target_rep function find = 'List.find'
declare {ocaml, hol} rename function find = list_find_opt

assert find_1 : ((find (fun n → n > (3 : NAT)) []) = Nothing)
assert find_2 : ((find (fun n → n > (3 : NAT)) [2; 1; 3]) = Nothing)
assert find_3 : ((find (fun n → n > (3 : NAT)) [2; 1; 5; 4]) = Just 5)
assert find_4 : ((find (fun n → n > (3 : NAT)) [2; 1; 4; 5; 4]) = Just 4)

lemma find_in : (∀ P l x. (find P l = Just x) ⟶ P x ∧ elem x l)
lemma find_not_in : (∀ P l. (find P l = Nothing) = (¬ (any P l)))


(* ----------------------------- *)
(* Lookup in an associative list *)
(* ----------------------------- *)
val lookup : ∀ α β. Eq α ⇒ α → LIST (α * β) → MAYBE β
val lookupBy : ∀ α β. (α → α → 𝔹) → α → LIST (α * β) → MAYBE β

(* DPM: eta-expansion for Coq backend type-inference. *)
let lookupBy eq k m = Maybe.map (fun x → snd x) (find (fun (k', _) → eq k k') m)
let inline lookup = lookupBy (=)

declare isabelle target_rep function 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 $\rightarrow$ $\neg$ (P x)) l)

val *reversePartition* : $\forall\, \alpha.\ (\alpha \rightarrow \mathbb{B}) \rightarrow$ LIST α $\rightarrow$ 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 $\rightarrow$ $x > (4 : \text{NAT})$) [] = ([], []))
assert $partition_1$: (partition (fun x $\rightarrow$ $x > (4 : \text{NAT})$) [1; 2; 4; 5; 2; 7; 6] = ([5; 7; 6], [1; 2; 4; 2]))
lemma *partition_fst* : ($\forall$ P l. fst (partition P l) = filter P l)
lemma *partition_snd* : ($\forall$ P l. snd (partition P l) = filter (fun x $\rightarrow$ $\neg$ (P x)) l)

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

val *deleteFirst* : $\forall\, \alpha.\ (\alpha \rightarrow \mathbb{B}) \rightarrow$ LIST α $\rightarrow$ MAYBE (LIST α)
let rec *deleteFirst* P l = match l with
| [] $\rightarrow$ Nothing
| x :: xs $\rightarrow$ if (P x) then Just xs else Maybe.map (fun xs' $\rightarrow$ $x :: xs'$) (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 $deleteFirst_1$: (deleteFirst (fun x $\rightarrow$ $x > (5 : \text{NAT})$) [3; 6; 7; 1] = Just [3; 7; 1])
assert $deleteFirst_2$: (deleteFirst (fun x $\rightarrow$ $x > (15 : \text{NAT})$) [3; 6; 7; 1] = Nothing)
assert $deleteFirst_3$: (deleteFirst (fun x $\rightarrow$ $x > (2 : \text{NAT})$) [3; 6; 7; 1] = Just [6; 7; 1])

val *delete* : $\forall\, \alpha.\ Eq\ \alpha \Rightarrow \alpha \rightarrow$ LIST α $\rightarrow$ LIST α
val *deleteBy* : $\forall\, \alpha.\ (\alpha \rightarrow \alpha \rightarrow \mathbb{B}) \rightarrow \alpha \rightarrow$ LIST α $\rightarrow$ 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_remove$_1$
declare {*ocaml*, *hol*} rename function deleteBy = list_delete

assert $delete_1$: (delete (6 : NAT) [(3 : NAT); 6; 7; 1] = [3; 7; 1])
assert $delete_2$: (delete (4 : NAT) [(3 : NAT); 6; 7; 1] = [3; 6; 7; 1])
assert $delete_3$: (delete (3 : NAT) [(3 : NAT); 6; 7; 1] = [6; 7; 1])
assert $delete_4$: (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 → 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 unzip1 : (unzip ([] : LIST (NAT * NAT)) = ([], []))
assert unzip2 : (unzip [((1 : NAT), (2 : NAT)); (2, 3); (3, 4)] = ([1; 2; 3], [2; 3; 4]))


instance ∀ α. SetType α ⇒ (SetType (LIST α))
 let setElemCompare = lexicographicCompareBy setElemCompare
end

(* ------------------------ *)
(* distinct elements        *)
(* ------------------------ *)

val allDistinct : ∀ α. Eq α ⇒ LIST α → 𝔹
let rec allDistinct l =
  match l with
    | [] → true
    | (x :: l') → ¬ (elem x l') ∧ allDistinct l'
  end
declare termination_argument allDistinct = automatic

declare hol target_rep function allDistinct = 'ALL_DISTINCT'

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

```
(* 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 $\{isabelle, hol, ocaml, coq\}$ rename module $=$ lem_list_extra

open import *Bool Maybe Basic_classes Tuple Num List Function_extra*

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

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

let *head* l $=$ match l with $\mid x :: xs\ \rightarrow\ x \mid []\ \rightarrow$ failwith "$List\_extra.headofemptylist$" end

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

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

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

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

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

let *tail* l $=$ match l with $\mid x :: xs\ \rightarrow\ xs \mid []\ \rightarrow$ failwith "$List\_extra.tailofemptylist$" end

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

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

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

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 $(x_2 :: xs)$ | $[]\ \rightarrow$ failwith "$List\_extra.lastofemptylist$" end

declare compile_message last $=$ "$lastisonlydefinedonnon{-}emptylistandshouldthereforebeavoided.Usemachinginsteadandha$

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

assert $last\_simple_1$: (last $[(3 : \text{NAT}); 1] = 1$)
assert $last\_simple_2$: (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 | $[x]\ \rightarrow\ []$ | $x_1 :: x_2 :: xs\ \rightarrow\ x_1$::(init $(x_2$::$xs)$) | $[]\ \rightarrow$ failwith "$List\_extra.initofemptylist$" end

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

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

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

lemma $init\_last\_append$: ($\forall\ l.$ length $l > 0 \longrightarrow (l =$ (init l) ++ [last l]))
lemma $init\_last\_dest$: ($\forall\ l.$ length $l > 0 \longrightarrow$ (dest_init $l =$ Just (init l, 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 $foldl_1\ f\ x\_xs\ =$ match x_xs with | $(x\ ::\ xs)\ \rightarrow$ foldl $f\ x\ xs$ | $[]\ \rightarrow$ failwith "$List\_extra.foldl1ofemptylist$" end

declare compile_message foldl_1 $=$ "$foldl1isonlydefinedonnon{-}emptylists.Betterusefoldlorexplicitpatternmatching.$"

val $foldr_1$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha)\ \rightarrow$ LIST $\alpha\ \rightarrow\ \alpha$
let $foldr_1\ f\ x\_xs\ =$ match x_xs with | $(x\ ::\ xs)\ \rightarrow$ foldr $f\ x\ xs$ | $[]\ \rightarrow$ failwith "$List\_extra.foldr1ofemptylist$" end

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 $e\ \rightarrow\ e$ | Nothing $\rightarrow$ failwith "$List\_extra.nth$" end
declare compile_message $\mathrm{foldl}_1\ =$ "$nthisundefinedfortoolargeindices, usecarefully$"

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 : $(\mathrm{nth}\ [0;1;2;3;4;5]\ 0 = (0 : \mathrm{NAT}))$
assert nth_1 : $(\mathrm{nth}\ [0;1;2;3;4;5]\ 1 = (1 : \mathrm{NAT}))$
assert nth_2 : $(\mathrm{nth}\ [0;1;2;3;4;5]\ 2 = (2 : \mathrm{NAT}))$
assert nth_3 : $(\mathrm{nth}\ [0;1;2;3;4;5]\ 3 = (3 : \mathrm{NAT}))$
assert nth_4 : $(\mathrm{nth}\ [0;1;2;3;4;5]\ 4 = (4 : \mathrm{NAT}))$
assert nth_5 : $(\mathrm{nth}\ [0;1;2;3;4;5]\ 5 = (5 : \mathrm{NAT}))$

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

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

val $findNonPure$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow$ LIST $\alpha\ \rightarrow\ \alpha$
let $findNonPure\ P\ l\ =$ match $(\mathrm{find}\ P\ l)$ with
| Just $e\ \rightarrow\ e$
| Nothing $\rightarrow$ failwith "$List\_extra.findNonPure$"
end

declare compile_message findNonPure $=$ "$findNonPureisundefinedifnoelementwiththepropertyisinthelist.Betterusefind.$"

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

val $zipSameLength$: $\forall\ \alpha\ \beta.$ LIST $\alpha\ \rightarrow$ LIST $\beta\ \rightarrow$ LIST $(\alpha\ *\ \beta)$
let rec $zipSameLength\ l_1\ l_2\ =$ match $(l_1,\ l_2)$ with
| $(x\ ::\ xs,\ y\ ::\ ys)\ \rightarrow\ (x,\ y) :: \mathrm{zipSameLength}\ xs\ ys$
| $([],\ [])\ \rightarrow\ []$
| $\_\ \rightarrow$ failwith "$List\_extra.zipSameLengthofdifferentlengthlists$"

end
declare termination_argument zipSameLength $=$ automatic

declare compile_message zipSameLength $=$ "$zipSameLengthisundefinedifthetwolistshavedifferentlengths$"

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

assert $zipSameLength_1$: $(\mathrm{zipSameLength}\ [(1 : \mathrm{NAT});\ 2;3;4;5]\ [(2 : \mathrm{NAT});\ 3;4;5;6] = [(1,\ 2);(2,\ 3);(3,\ 4);(4,\ 5);(5,\ 6)])$

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*, *coq*} rename module = lem_set_helpers

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

```
(* ------------------------ *)
(* 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.Itsresultmaydifferbetwe*
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 element 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 $\{isabelle,\ hol,\ ocaml,\ coq\}$ 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.List*
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 <> s_1 s_2 = ¬ (setEqual s_1 s_2)
end

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

val *setCompareBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \text{ORDERING})\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{ORDERING}$
declare *coq* target_rep function setCompareBy = `'set_compare_by'`
declare *ocaml* target_rep function setCompareBy = `'Pset.compare_by'`

val *setCompare* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{ORDERING}$
let inline {*coq*} *setCompare* = setCompareBy setElemCompare
declare *ocaml* target_rep function setCompare = `'Pset.compare'`

instance $\forall\ \alpha.\ \mathit{SetType}\ \alpha\ \Rightarrow\ (\mathit{SetType}\ (\text{SET}\ \alpha))$
let *setElemCompare* = setCompare
end

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

val *empty* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha\ \Rightarrow\ \text{SET}\ \alpha$
val *emptyBy* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \text{ORDERING})\ \rightarrow\ \text{SET}\ \alpha$

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 = `'∅'`
declare *tex* target_rep function empty = `'$\emptyset$'`

assert empty_0 : $(\emptyset\ :\ \text{SET}\ \mathbb{B}) = \{\}$
assert empty_1 : $(\emptyset\ :\ \text{SET NAT}) = \{\}$
assert empty_2 : $(\emptyset\ :\ \text{SET (LIST NAT)}) = \{\}$
assert empty_3 : $(\emptyset\ :\ \text{SET (SET NAT)}) = \{\}$

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

val *any* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha\ \Rightarrow\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let inline *any* $P\ s\ =\ (\exists\ e \in s.\ P\ e)$

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 any_0 : any (fun $(x : \text{NAT})\ \rightarrow\ x > 5)\ \{3, 4, 6\}$
assert any_1 : ¬ (any (fun $(x : \text{NAT})\ \rightarrow\ x > 10)\ \{3, 4, 6\})$

val all : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ (\alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
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] : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
val $memberBy$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \text{ORDERING})\ \rightarrow\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$

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 : $((1 : \text{NAT}) \in \{(2 : \text{NAT}),\ 3,\ 1\})$
assert in_2 : $(\neg\ ((1 : \text{NAT}) \in \{2,\ 3,\ 4\}))$
assert in_3 : $(\neg\ ((1 : \text{NAT}) \in \{\}))$
assert in_4 : $((1 : \text{NAT}) \in \{1,\ 2,\ 1,\ 3,\ 1,\ 4\})$

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

val NIN [notMember] : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
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 : $\neg\ ((1 : \text{NAT}) \notin \{2,\ 3,\ 1\})$
assert nin_2 : $((1 : \text{NAT}) \notin \{2,\ 3,\ 4\})$
assert nin_3 : $((1 : \text{NAT}) \notin \{\})$
assert nin_4 : $\neg\ ((1 : \text{NAT}) \notin \{1,\ 2,\ 1,\ 3,\ 1,\ 4\})$

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

val $insert$: $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \alpha$ (* 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 $insert_1$: $((\text{insert } (2 : \text{NAT}) \{3, 4\}) = \{2, 3, 4\})$
assert $insert_2$: $((\text{insert } (3 : \text{NAT}) \{3, 4\}) = \{3, 4\})$
assert $insert_3$: $((\text{insert } (3 : \text{NAT}) \{\}) = \{3\})$

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

val *null* : $\forall\, \alpha.\ \mathit{SetType}\ \alpha \Rightarrow \text{SET}\ \alpha \rightarrow \mathbb{B}$ `(* 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 $null_1$: $(\text{null } (\{\} : \text{SET NAT}))$
assert $null_2$: $(\neg\ (\text{null } \{(1 : \text{NAT})\}))$

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

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

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

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

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

val *size* : $\forall\, \alpha.\ \mathit{SetType}\ \alpha \Rightarrow \text{SET}\ \alpha \rightarrow \text{NAT}$

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

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

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

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

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

```
(* 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 set_case = `'set_CASE'`
declare *isabelle* target_rep function set_case = `'set_case'`
declare *coq* target_rep function set_case = `'set_case'`
declare *ocaml* target_rep function set_case = `'Pset.set_case'`

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

assert $\mathit{set\_patterns}_0$: (
  match $(\{\}\ :\ \textsc{set nat})$ with
    | $\emptyset$ $\rightarrow$ true
    | _ $\rightarrow$ false
  end
)

assert $\mathit{set\_patterns}_1$: $\neg$ (
  match $\{(2 : \textsc{nat})\}$ with
    | $\emptyset$ $\rightarrow$ true
    | _ $\rightarrow$ false
  end
)

assert $\mathit{set\_patterns}_2$: $\neg$ (
  match $\{(3\ :\ \textsc{nat}),\ 4\}$ with
    | $\emptyset$ $\rightarrow$ true
    | _ $\rightarrow$ false
  end
)

assert $\mathit{set\_patterns}_3$: (
  match $(\{2\}\ :\ \textsc{set nat})$ with
    | $\emptyset$ $\rightarrow$ 0
    | singleton x $\rightarrow$ x
    | _ $\rightarrow$ 1
  end
) $= 2$

assert $\mathit{set\_patterns}_4$: (
  match $(\{\}\ :\ \textsc{set nat})$ with
    | $\emptyset$ $\rightarrow$ 0
    | 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* : $\forall\ \alpha.\ SetType\ \alpha,\ Ord\ \alpha \Rightarrow \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha * \text{SET}\ \alpha$
let *split* p s = (filter ((<) p) s, filter ((>) p) s)
declare {*hol*} rename function split = SET_SPLIT

val *splitMember* : $\forall\ \alpha.\ SetType\ \alpha,\ Ord\ \alpha \Rightarrow \alpha \rightarrow \text{SET}\ \alpha \rightarrow \text{SET}\ \alpha * \mathbb{B} * \text{SET}\ \alpha$
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 $bigunion_0$: $(\bigcup \{\{(1 : \textsc{nat})\}\} = \{1\})$
assert $bigunion_1$: $(\bigcup \{\{(1 : \textsc{nat}), 2, 3\}\ ,\ \{3, 2, 4\}\} = \{1, 2, 3, 4\})$
assert $bigunion_2$: $(\bigcup \{\{(1 : \textsc{nat}), 2, 3\}\ ,\ \{3, 2, 4\},\ \{\}\} = \{1, 2, 3, 4\})$
assert $bigunion_3$: $(\bigcup \{\{(1 : \textsc{nat}), 2, 3\}\ ,\ \{3, 2, 4\},\ \{5\}\} = \{1, 2, 3, 4, 5\})$
lemma *bigunion_in* : $(\forall\ e\ bs.\ e \in \bigcup bs \longleftrightarrow (\exists\ s.\ s \in bs \land e \in s))$

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

val *differenceBy* : $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \textsc{ordering}) \rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha$
val *difference* : $\forall\ \alpha.\ SetType\ \alpha \Rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha$
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 $\{(1 : \textsc{nat}), 2, 3\}\ \{3, 2, 4\} = \{1\}$)
lemma *difference_in* : $(\forall\ e\ s_1\ s_2.\ e \in (\text{difference}\ s_1\ s_2) \longleftrightarrow (e \in s_1 \land \neg\ (e \in s_2)))$

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

val *intersection* : $\forall\ \alpha.\ SetType\ \alpha \Rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha$
val *intersectionBy* : $\forall\ \alpha.\ (\alpha \rightarrow \alpha \rightarrow \textsc{ordering}) \rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \alpha$

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 $intersection_1$: $(\{1, 2, 3\} \cap \{(3 : \textsc{nat}), 2, 4\} = \{2, 3\})$
lemma *intersection_in* : $(\forall\ e\ s_1\ s_2.\ e \in (s_1 \cap s_2) \longleftrightarrow (e \in s_1 \land e \in s_2))$

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

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

val *mapBy* : $\forall\ \alpha\ \beta.\ (\beta \rightarrow \beta \rightarrow \textsc{ordering}) \rightarrow (\alpha \rightarrow \beta) \rightarrow \textsc{set}\ \alpha \rightarrow \textsc{set}\ \beta$

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

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

(* In order to avoid providing an comparison function for sets of sets,
   it might be better to combine bigunion and map sometimes into a single operation. *)
```

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

let inline *bigunionMap f bs* $= \bigcup (\text{map } f\ bs)$

declare *ocaml* target_rep function bigunionMapBy $=$ `'Pset.map_union'`
let inline $\{ocaml\}$ *bigunionMap* $=$ bigunionMapBy setElemCompare

assert $bigunionmap_0$: $(\text{bigunionMap } (\text{fun } n \rightarrow \{n,\ 2 * n,\ 3 * n\})\ \{(1 : \text{NAT})\} = \{1, 2, 3\})$
assert $bigunionmap_1$: $(\text{bigunionMap } (\text{fun } n \rightarrow \{n,\ 2 * n,\ 3 * n\})\ \{(2 : \text{NAT}), 8\} = \{2, 4, 6, 8, 16, 24\})$

```
(* ------------------------ *)
(* 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 : NAT); 4; 3] = {2, 3, 4})
assert $fromList_2$: (fromList [(2 : 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* = $\{\ (a,\ b) \mid \forall\ a \in sa\ b \in sb\ a \mid \text{true}\ \}$
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 $n \rightarrow \{n{*}2,\ n * 3\}$) = {(2, 4), (2, 6), (3, 6), (3, 9)})
lemma $Sigma_2$: $(\forall\ sa\ sb\ a\ b.\ ((a,\ b) \in \text{sigma}\ sa\ sb) \longleftrightarrow ((a \in sa) \wedge (b \in sb\ a)))$

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

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

declare *ocaml* target_rep function crossBy = 'Pset.cross'

let *cross* $s_1\ s_2$ = $\{\ (e_1,\ e_2) \mid \forall\ e_1 \in s_1\ e_2 \in s_2 \mid \text{true}\ \}$

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* : $\forall\ s_1\ s_2.\ s_1 \times s_2 = \text{sigma}\ s_1\ (\text{const}\ s_2)$
assert $cross_1$: ({(2 : NAT), 3} × {true, false} = {(2, true), (3, true), (2, false), (3, false)})

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

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

```
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 lfp_empty_0 : leastFixedPoint 0 (map (fun x → x)) ({} : SET NAT) = {}
assert lfp_empty_1 : leastFixedPoint 1 (map (fun x → x)) ({} : SET NAT) = {}
assert lfp_saturate_neg_1 : leastFixedPoint 1 (map (fun x → −x)) ({1, 2, 3} : SET INT) = {−3, −2, −1, 1, 2, 3}

assert lfp_saturate_neg_2 : leastFixedPoint 2 (map (fun x → −x)) ({1, 2, 3} : SET INT) = {−3, −2, −1, 1, 2, 3}

assert lfp_saturate_mod_3 : leastFixedPoint 3 (map (fun x → (2*x) mod 5)) ({1} : SET NAT) = {1, 2, 3, 4}

assert lfp_saturate_mod_4 : leastFixedPoint 4 (map (fun x → (2*x) mod 5)) ({1} : SET NAT) = {1, 2, 3, 4}

assert lfp_saturate_mod_5 : leastFixedPoint 5 (map (fun x → (2*x) mod 5)) ({1} : SET NAT) = {1, 2, 3, 4}

assert lfp_termination : {1, 3, 5, 7, 9} ⊆ leastFixedPoint 5 (map (fun x → 2+x)) {(1 : ℕ)}
```

12 Map

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

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

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

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

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

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

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

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 $\forall\ 'k\ 'v.\ Eq\ 'k,\ Eq\ 'v\ \Rightarrow\ (Eq\ (\textsc{map}\ 'k\ 'v))$
  let = = mapEqual
  let <> $m_1\ m_2$ = $\neg$ (mapEqual $m_1\ m_2$)
end

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

class (*MapKeyType* α)
  val $\{ocaml,\ coq\}$ *mapKeyCompare* : $\alpha\ \rightarrow\ \alpha\ \rightarrow\ \textsc{ordering}$
end

default_instance $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ (\ MapKeyType\ \alpha\ )$
  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 $lookupBy$: $\forall$ 'k 'v. ('k $\rightarrow$ 'k $\rightarrow$ ORDERING) $\rightarrow$ 'k $\rightarrow$ MAP 'k 'v $\rightarrow$ MAYBE 'v
declare *coq* target_rep function lookupBy = `'fmap_lookup_by'`

val $lookup$: $\forall$ 'k 'v. *MapKeyType* 'k $\Rightarrow$ 'k $\rightarrow$ MAP 'k 'v $\rightarrow$ 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 $\mathbb{B}$) = Nothing)
assert $find\_insert_0$: (lookup 16 (insert (16 : NAT) true empty) = Just true)

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

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

val *findWithDefault* : $\forall$ 'k 'v. *MapKeyType* 'k $\Rightarrow$ 'k $\rightarrow$ 'v $\rightarrow$ MAP 'k 'v $\rightarrow$ 'v
let inline *findWithDefault* k v m = fromMaybe v (lookup k m)

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

val *fromList* : $\forall$ 'k 'v. *MapKeyType* 'k $\Rightarrow$ LIST ('k $*$ 'v) $\rightarrow$ MAP 'k 'v
let *fromList* l = foldl (fun m (k, v) $\rightarrow$ 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* : $\forall$ 'k 'v. *MapKeyType* 'k, *SetType* 'k, *SetType* 'v $\Rightarrow$ MAP 'k 'v $\rightarrow$ SET ('k $*$ 'v)
val *toSetBy* : $\forall$ 'k 'v. (('k $*$ 'v) $\rightarrow$ ('k $*$ 'v) $\rightarrow$ ORDERING) $\rightarrow$ MAP 'k 'v $\rightarrow$ 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 $\rightarrow$ 'k $\rightarrow$ ORDERING) $\rightarrow$ MAP 'k 'v $\rightarrow$ SET 'k
val *domain* : $\forall$ 'k 'v. *MapKeyType* 'k, *SetType* 'k $\Rightarrow$ MAP 'k 'v $\rightarrow$ 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 $\Rightarrow$ MAP 'k 'v $\rightarrow$ SET 'v
val *rangeBy* : $\forall$ 'k 'v. ('v $\rightarrow$ 'v $\rightarrow$ ORDERING) $\rightarrow$ MAP 'k 'v $\rightarrow$ 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 = `'fmap_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 $\Rightarrow$ 'k $\rightarrow$ MAP 'k 'v $\rightarrow$ $\mathbb{B}$
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 $\Rightarrow$ 'k $\rightarrow$ MAP 'k 'v $\rightarrow$ $\mathbb{B}$
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 any0 : (any (fun _k v → v) (insert 36 false (insert (16 : NAT) true empty)))
assert any1 : (¬ (any (fun _k v → v) (insert 36 false (insert (16 : NAT) false empty))))
assert any2 : (any (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) true empty)))
assert any3 : (¬ (any (fun _k v → ¬ v) (insert 36 true (insert (16 : NAT) true empty))))

assert all0 : (all (fun _k v → v) (insert 36 true (insert (16 : NAT) true empty)))
assert all1 : (¬ (all (fun _k v → v) (insert 36 true (insert (16 : NAT) false empty))))
assert all2 : (all (fun _k v → ¬ v) (insert 36 false (insert (16 : NAT) false empty)))
assert all3 : (¬ (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_insert1 : (¬ (member (5 : NAT) (delete 5 (insert 5 true empty))))
assert delete_insert2 : (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 = ('@' 'List.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*, *coq*} rename module = lem_map_extra

open import *Bool Basic_classes Function Function_extra Maybe List Num Set Map*

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

val *find* : ∀ 'k 'v. *MapKeyType* 'k ⇒ 'k → MAP 'k 'v → 'v
let *find* k m = match (lookup k m) with Just x → x | Nothing → failwith "$Map\_extra.find$" 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 = "$findisonlydefinedifthekeyisfound.Uselookupinsteadandhandlethenot-foundcaseexplicitly.$"
assert $find\_insert_1$: (find 16 (insert (16 : NAT) true empty) = true)
assert $find\_insert_2$: (find 36 (insert 36 false (insert (16 : NAT) true empty)) = false)

```
(* ------------------------------------------------------------------------ *)
(* from sets / domain / range                                               *)
(* ------------------------------------------------------------------------ *)
```

val *fromSet* : ∀ 'k 'v. *MapKeyType* 'k ⇒ ('k → 'v) → SET 'k → MAP 'k 'v
let *fromSet* f s = Set_helpers.fold (fun k m → Map.insert k (f k) m) s Map.empty

declare compile_message fromSet = "$fromSetonlyworksforfinitesets, usecarefully.$"

declare *ocaml* target_rep function fromSet = `'Pmap.from_set'`
declare *hol* target_rep function fromSet = `'FUN_FMAP'`

assert $fromSet_0$: (fromSet succ (∅ : SET NAT) = Map.empty)
assert $fromSet_1$: (fromSet succ {(2 : NAT), 3, 4}) = Map.fromList [(2, 3); (3, 4); (4, 5)]

14 Maybe_extra

```
(******************************************************************************)
(* extra functions for maybe / option                                         *)
(*                                                                            *)
(******************************************************************************)
```

declare {*isabelle*, *hol*, *ocaml*, *coq*} rename module = lem_maybe_extra

open import *Basic_classes Maybe Function_extra*

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

val *fromJust* : $\forall\ \alpha.$ MAYBE $\alpha \ \rightarrow\ \alpha$
let *fromJust op* = match *op* with | Just $v \ \rightarrow\ v$ | Nothing $\rightarrow$ failwith "*fromJustofNothing*" end
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*, *coq*} 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.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 = `'Either.Left'`
declare *ocaml* target_rep function Right = `'Either.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* : $\forall\, \alpha\, \beta.\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ (\text{EITHER}\ \alpha\ \beta)\ \rightarrow\ (\text{EITHER}\ \alpha\ \beta)\ \rightarrow\ \mathbb{B}$
val *eitherEqualBy* : $\forall\, \alpha\, \beta.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ (\beta\ \rightarrow\ \beta\ \rightarrow\ \mathbb{B})\ \rightarrow\ (\text{EITHER}\ \alpha\ \beta)\ \rightarrow\ (\text{EITHER}\ \alpha\ \beta)\ \rightarrow\ \mathbb{B}$

let *eitherEqualBy eql eqr* (*left* : EITHER α β) (*right* : EITHER α β) =
  match (*left*, *right*) with
    | (Left l, Left l') $\rightarrow$ *eql* l l'
    | (Right r, Right r') $\rightarrow$ *eqr* r r'
    | _ $\rightarrow$ 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 $\forall\, \alpha\, \beta.\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ (Eq\ (\text{EITHER}\ \alpha\ \beta))$
  let = = eitherEqual

let <> x y = ¬ (eitherEqual x y)
end

assert $either\_equal_1$: (((Left false) : EITHER $\mathbb{B}$ $\mathbb{B}$) = Left false)
assert $either\_equal_2$: (((Left true) : EITHER $\mathbb{B}$ $\mathbb{B}$) ≠ Left false)
assert $either\_equal_3$: (((Left true) : EITHER $\mathbb{B}$ $\mathbb{B}$) = Left true)
assert $either\_equal_4$: (((Right false) : EITHER $\mathbb{B}$ $\mathbb{B}$) = Right false)
assert $either\_equal_5$: (((Right false) : EITHER $\mathbb{B}$ $\mathbb{B}$) ≠ Right true)
assert $either\_equal_6$: (((Right true) : EITHER $\mathbb{B}$ $\mathbb{B}$) ≠ Left true)
assert $either\_equal_7$: (((Left true) : EITHER $\mathbb{B}$ $\mathbb{B}$) ≠ Right true)

assert $either\_pattern_1$: (match (Left true) with Left x → x | Right y → ¬ y end)
assert $either\_pattern_2$: (match (Right false) with Left x → x | Right y → ¬ y end)
assert $either\_pattern_3$: (¬ (match (Left false) with Left x → x | Right y → ¬ y end))
assert $either\_pattern_4$: (¬ (match (Right true) with Left x → x | Right y → ¬ y end))

```
(* -------------------------------------------------------------------------- *)
(* Utility functions.                                                         *)
(* -------------------------------------------------------------------------- *)
```

val $isLeft$: ∀ α β. EITHER α β → $\mathbb{B}$
let inline $isLeft$ = function
  | Left _ → true
  | Right _ → false
end

declare hol target_rep function isLeft = `'ISL'`

assert $isLeft_1$: (isLeft ((Left true) : EITHER $\mathbb{B}$ $\mathbb{B}$))
assert $isLeft_2$: (¬ (isLeft ((Right true) : EITHER $\mathbb{B}$ $\mathbb{B}$)))

val $isRight$: ∀ α β. EITHER α β → $\mathbb{B}$
let inline $isRight$ = function
  | Right _ → true
  | Left _ → false
end

declare hol target_rep function isRight = `'ISR'`

assert $isRight_1$: (isRight ((Right true) : EITHER $\mathbb{B}$ $\mathbb{B}$))
assert $isRight_2$: (¬ (isRight ((Left true) : EITHER $\mathbb{B}$ $\mathbb{B}$)))

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 $either_1$: (either ((fun b → ¬ b)) (fun b → b) (Left true) = false)
assert $either_2$: (either ((fun b → ¬ b)) (fun b → b) (Left false) = true)
assert $either_3$: (either ((fun b → ¬ b)) (fun b → b) (Right true) = true)
assert $either_4$: (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 $\{isabelle,\ ocaml,\ hol,\ coq\}$ rename module $=$ lem_relation

open import *Bool Basic_classes Tuple Set Num*
open import $\{hol\}$ *set_relationTheory*

```
(* ========================================================================== *)
(* The type of relations                                                      *)
(* ========================================================================== *)
```

type $\textsc{rel\_pred}\ \alpha\ \beta\ =\ \alpha\ \rightarrow\ \beta\ \rightarrow\ \mathbb{B}$
type $\textsc{rel\_set}\ \alpha\ \beta\ =\ \textsc{set}\ (\alpha\ *\ \beta)$

```
(* 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 $\textsc{rel}\ \alpha\ \beta\ =\ \textsc{rel\_set}\ \alpha\ \beta$

val $relToSet\ :\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \textsc{rel}\ \alpha\ \beta\ \rightarrow\ \textsc{rel\_set}\ \alpha\ \beta$
val $relFromSet\ :\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \textsc{rel\_set}\ \alpha\ \beta\ \rightarrow\ \textsc{rel}\ \alpha\ \beta$

let inline $relToSet\ s\ =\ s$
let inline $relFromSet\ r\ =\ r$

val $relEq\ :\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta\ \Rightarrow\ \textsc{rel}\ \alpha\ \beta\ \rightarrow\ \textsc{rel}\ \alpha\ \beta\ \rightarrow\ \mathbb{B}$
let $relEq\ r_1\ r_2\ =\ (\mathrm{relToSet}\ r_1 = \mathrm{relToSet}\ r_2)$

```
(*
instance forall 'a 'b. SetType 'a, SetType 'b => (Eq (rel 'a 'b))
  let (=) = relEq
end
*)
```

lemma $relToSet\_inv:\ (\forall\ r.\ \mathrm{relFromSet}\ (\mathrm{relToSet}\ r) = r)$

val $relToPred\ :\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \textsc{rel}\ \alpha\ \beta\ \rightarrow\ \textsc{rel\_pred}\ \alpha\ \beta$
val $relFromPred\ :\ \forall\ \alpha\ \beta.\ SetType\ \alpha,\ SetType\ \beta,\ Eq\ \alpha,\ Eq\ \beta\ \Rightarrow\ \textsc{set}\ \alpha\ \rightarrow\ \textsc{set}\ \beta\ \rightarrow\ \textsc{rel\_pred}\ \alpha\ \beta\ \rightarrow$
$\textsc{rel}\ \alpha\ \beta$

let $relToPred\ r\ =\ (\mathsf{fun}\ x\ y\ \rightarrow\ (x,\ y) \in \mathrm{relToSet}\ r)$
let $relFromPred\ xs\ ys\ p\ =\ \mathrm{Set.filter}\ (\mathsf{fun}\ (x,\ y)\ \rightarrow\ p\ x\ y)\ (xs \times ys)$

let inline $\{hol\}\ relToPred\ r\ x\ y\ =\ (x,\ y) \in \mathrm{relToSet}\ r$

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 : ∀ α. SetType α, Eq α ⇒ SET α → REL α α
let relIdOn s = relFromPred s s (=)

val relId : ∀ α. SetType α, Eq α ⇒ REL α α
let ~{coq, ocaml} relId = {(x, x) | ∀ x | true}

lemma relId_spec : (∀ x y s. (inRel x y (relIdOn s) ⟷ (x ∈ s ∧ (x = y))))

assert rel_id0 : inRel (0 : NAT) 0 (relIdOn {0, 1, 2, 3})
assert rel_id1 : inRel (2 : NAT) 2 (relIdOn {0, 1, 2, 3})
assert rel_id2 : ¬ (inRel (5 : NAT) 5 (relIdOn {0, 1, 2, 3}))
assert rel_id3 : ¬ (inRel (0 : NAT) 2 (relIdOn {0, 1, 2, 3}))

(* ----------------------- *)
(* relation union          *)
(* ----------------------- *)

val relUnion : ∀ α β. SetType α, SetType β ⇒ REL α β → REL α β → REL α β
let inline relUnion r1 r2 = relFromSet ((relToSet r1) ∪ (relToSet r2))

lemma in_rel_union : (∀ a b r1 r2. inRel a b (relUnion r1 r2) = inRel a b r1 ∨ inRel a b r2)
assert rel_union0 : relUnion (relAdd (2 : NAT) true relEmpty) (relAdd 5 false relEmpty) =
            relFromSet {(5, false), (2, true)}

(* ----------------------- *)
(* relation intersection   *)
(* ----------------------- *)

val relIntersection : ∀ α β. SetType α, SetType β, Eq α, Eq β ⇒ REL α β → REL α β → REL α β
let inline relIntersection r1 r2 = relFromSet ((relToSet r1) ∩ (relToSet r2))

lemma in_rel_inter : (∀ a b r1 r2. inRel a b (relIntersection r1 r2) = inRel a b r1 ∧ inRel a b r2)
assert rel_inter0 : relIntersection (relAdd (2 : NAT) true (relAdd 7 false relEmpty))
                        (relAdd 7 false (relAdd 2 false relEmpty)) =
            relFromSet {(7, false)}

(* ----------------------- *)
(* Relation Composition    *)
(* ----------------------- *)

val relComp : ∀ α β γ. SetType α, SetType β, SetType γ, Eq α, Eq β ⇒ REL α β → REL β γ → REL α γ

let relComp r1 r2 = relFromSet {(e1, e3) | ∀ (e1, e2) ∈ (relToSet r1) (e2', e3) ∈ (relToSet r2) | e2 = e2'}

declare hol target_rep function relComp = 'rcomp'

lemma rel_comp1 : (∀ r1 r2 e1 e2 e3. (inRel e1 e2 r1 ∧ inRel e2 e3 r2) ⟶ inRel e1 e3 (relComp r1 r2))
lemma ~{coq, ocaml} rel_comp2 : (∀ r. (relComp r relId = r) ∧ (relComp relId r = r))
lemma rel_comp3 : (∀ r. (relComp r relEmpty = relEmpty) ∧ (relComp relEmpty r = relEmpty))

assert rel_comp0 : (relComp (relFromSet {((2 : NAT), (4 : NAT)), (2, 8)}) (relFromSet {(4, (3 : NAT)), (2, 8)}) =
            relFromSet {(2, 3)})

(* ----------------------- *)
(* restrict                *)
(* ----------------------- *)

val relRestrict : ∀ α. SetType α, Eq α ⇒ REL α α → SET α → REL α α
```

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 $\{((2 : \text{NAT}),\ (4 : \text{NAT})),\ (2,\ 2),\ (2,\ 8)\}$) $\{2,\ 8\}$ =
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* : $\forall\ \alpha\ \beta$. *SetType* α, *SetType* β $\Rightarrow$ REL $\alpha\ \beta$ $\rightarrow$ REL $\beta\ \alpha$
let *relConverse* r = relFromSet (Set.map swap (relToSet r))

declare {*hol*} rename function relConverse = lem_converse

assert *rel_converse*$_0$: relConverse (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (4, 5)\}$) =
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* : $\forall\ \alpha\ \beta$. *SetType* α, *SetType* β $\Rightarrow$ REL $\alpha\ \beta$ $\rightarrow$ 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 $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (4, 5)\}$) = $\{2, 3, 4\}$
assert *rel_domain*$_1$: relDomain (relFromSet $\{((5 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (4, 5)\}$) = $\{3, 4, 5\}$
assert *rel_domain*$_2$: relDomain (relFromSet $\{((3 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (4, 5)\}$) = $\{3, 4\}$

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

val *relRange* : $\forall\ \alpha\ \beta$. *SetType* α, *SetType* β $\Rightarrow$ REL $\alpha\ \beta$ $\rightarrow$ 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 $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (4, 5)\}$) = $\{3, 4, 5\}$
assert *rel_range*$_1$: relRange (relFromSet $\{((5 : \text{NAT}),\ (6 : \text{NAT})),\ (3,\ 4),\ (4, 5)\}$) = $\{4, 5, 6\}$
assert *rel_range*$_2$: relRange (relFromSet $\{((3 : \text{NAT}),\ (5 : \text{NAT})),\ (3,\ 4),\ (4, 5)\}$) = $\{4, 5\}$

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

```
(* 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 $x\ y\ r$) $s = (x \in s \wedge y \in s \wedge$ 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 $\mathit{is\_antisymmetric}_0$: isAntisymmetric (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT})),\ (3,\ 3),\ (3,\ 4),\ (4, 5)\}$)
assert $\mathit{is\_antisymmetric}_1$: $\neg$ (isAntisymmetric (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (4, 5),\ (2,\ 4)\}$))

lemma *is_antisymmetric_empty* : $\forall$ r. isAntisymmetricOn r {}
lemma *is_antisymmetric_sing* : $\forall$ r x. isAntisymmetricOn r $\{x\}$

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

val *isTransitiveOn* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let *isTransitiveOn* r s = ($\forall$ $e_1 \in s$ $e_2 \in s$ $e_3 \in s$. (inRel e_1 e_2 r) $\longrightarrow$ (inRel e_2 e_3 r) $\longrightarrow$ (inRel e_1 e_3 r))

declare $\{hol\}$ rename function isTransitiveOn = lem_transitive_on

val *isTransitive* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let *isTransitive* r = ($\forall$ $(e_1,\ e_2) \in$ relToSet r $e_3 \in$ relApply r $\{e_2\}$. inRel e_1 e_3 r)

declare *hol* target_rep function isTransitive = `'transitive'`

assert $\mathit{is\_transitive\_on}_0$: isTransitiveOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (2, 4),\ (4, 5),\ (5,\ 4)\}$) $\{2, 3, 4\}$

assert $\mathit{is\_transitive\_on}_1$: $\neg$ (isTransitiveOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (2, 4),\ (4, 5),\ (5,\ 4)\}$) $\{2, 3, 4, 5\}$)

assert $\mathit{is\_transitive}_0$: $\neg$ (isTransitive (relFromSet $\{((2 : \text{NAT}),\ (2 : \text{NAT})),\ (3,\ 3),\ (3,\ 4),\ (4, 5)\}$))
assert $\mathit{is\_transitive}_1$: isTransitive (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (2,\ 4)\ \}$)

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

val *isTotalOn* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let *isTotalOn* r s = ($\forall$ $e_1 \in s$ $e_2 \in s$. (inRel e_1 e_2 r) $\lor$ (inRel e_2 e_1 r))

declare $\{hol\}$ rename function isTotalOn = lem_is_total_on

val *isTotal* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml,\ coq\}$ *isTotal* r = ($\forall$ e_1 e_2. (inRel e_1 e_2 r) $\lor$ (inRel e_2 e_1 r))
declare $\{hol\}$ rename function isTotal = lem_is_total

val *isTrichotomousOn* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let *isTrichotomousOn* r s = ($\forall$ $e_1 \in s$ $e_2 \in s$. (inRel e_1 e_2 r) $\lor$ $(e_1 = e_2)$ $\lor$ (inRel e_2 e_1 r))

declare $\{hol\}$ rename function isTrichotomousOn = lem_is_trichotomous_on

val *isTrichotomous* : $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml,\ coq\}$ *isTrichotomous* r = ($\forall$ e_1 e_2. (inRel e_1 e_2 r) $\lor$ $(e_1 = e_2)$ $\lor$ (inRel e_2 e_1 r))

declare $\{hol\}$ rename function isTrichotomous = lem_is_trichotomous

assert $\mathit{is\_total\_on}_0$: isTotalOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (3, 3),\ (4, 4)\}$) $\{3, 4\}$
assert $\mathit{is\_total\_on}_1$: $\neg$ (isTotalOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (3, 3),\ (4, 4)\}$) $\{2, 4\}$)

assert $\mathit{is\_trichotomous\_on}_0$: isTrichotomousOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4)\}$) $\{3, 4\}$
assert $\mathit{is\_trichotomous\_on}_1$: $\neg$ (isTrichotomousOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4)\}$) $\{2, 3, 4\}$)

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

val $\mathit{isSingleValued}$: $\forall\ \alpha\ \beta.\ \mathit{SetType}\ \alpha,\ \mathit{SetType}\ \beta,\ \mathit{Eq}\ \alpha,\ \mathit{Eq}\ \beta\ \Rightarrow\ \text{REL}\ \alpha\ \beta\ \rightarrow\ \mathbb{B}$
let $\mathit{isSingleValued}\ r\ =\ (\forall\ (e_1,\ \mathit{e2a}) \in \text{relToSet}\ r\ \mathit{e2b} \in \text{relApply}\ r\ \{e_1\}.\ \mathit{e2a} = \mathit{e2b})$

declare $\{\mathit{hol}\}$ rename function isSingleValued = lem_is_single_valued

assert $\mathit{is\_single\_valued}_0$: isSingleValued (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4)\}$)
assert $\mathit{is\_single\_valued}_1$: $\neg$ (isSingleValued (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (2, 4)\ ,\ (3,\ 4)\}$))

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

val $\mathit{isEquivalenceOn}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $\mathit{isEquivalenceOn}\ r\ s$ = isReflexiveOn $r\ s\ \wedge$ isSymmetricOn $r\ s\ \wedge$ isTransitiveOn $r\ s$

declare $\{\mathit{hol}\}$ rename function isEquivalenceOn = lem_is_equivalence_on

val $\mathit{isEquivalence}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{\mathit{ocaml},\ \mathit{coq}\}\ \mathit{isEquivalence}\ r$ = isReflexive $r\ \wedge$ isSymmetric $r\ \wedge$ isTransitive r

declare $\{\mathit{hol}\}$ rename function isEquivalence = lem_is_equivalence

assert $\mathit{is\_equivalence}_0$: isEquivalenceOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2, 2),\ (3, 3),\ (4, 4)\}$) $\{2, 3, 4\}$

assert $\mathit{is\_equivalence}_1$: $\neg$ (isEquivalenceOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2,\ 4),\ (2, 2),\ (3, 3),\ (4, 4)\}$) $\{2, 3, 4$

assert $\mathit{is\_equivalence}_2$: $\neg$ (isEquivalenceOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2, 2),\ (3, 3),\ \}$) $\{2, 3, 4\}$)

```
(* ----------------------- *)
(* well founded            *)
(* ----------------------- *)
```

val $\mathit{isWellFounded}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{\mathit{ocaml},\ \mathit{coq}\}\ \mathit{isWellFounded}\ r\ =\ (\forall\ P.\ (\forall\ x.\ (\forall\ y.\ \text{inRel}\ y\ x\ r \longrightarrow P\ x) \longrightarrow P\ x) \longrightarrow (\forall\ x.\ P\ x))$

declare hol target_rep function isWellFounded r = `'WF'` (`'reln_to_rel'` r)

```
(* ========================================================================== *)
(* Orders                                                                     *)
(* ========================================================================== *)
```

```
(* ----------------------- *)
(* pre- or quasiorders     *)
(* ----------------------- *)
```

val $isPreorderOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isPreorderOn\ r\ s\ =\ \text{isReflexiveOn}\ r\ s \wedge \text{isTransitiveOn}\ r\ s$

declare $\{hol\}$ rename function isPreorderOn = lem_is_preorder_on

val $isPreorder$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml,\ coq\}\ isPreorder\ r\ =\ \text{isReflexive}\ r \wedge \text{isTransitive}\ r$

declare $\{hol\}$ rename function isPreorder = lem_is_preorder

assert $is\_preorder_0$: isPreorderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2,\ 2),\ (3,\ 3),\ (4,\ 4)\}$) $\{2,\ 3,\ 4\}$

assert $is\_preorder_1$: $\neg$ (isPreorderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (2,\ 2),\ (3,\ 3)\}$) $\{2,\ 3,\ 4\}$)
assert $is\_preorder_2$: $\neg$ (isPreorderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (2,\ 2),\ (3,\ 3),\ (4,\ 4)\}$) $\{2,\ 3,\ 4\}$)

```
(* ----------------------- *)
(* partial orders          *)
(* ----------------------- *)
```

val $isPartialOrderOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isPartialOrderOn\ r\ s\ =\ \text{isReflexiveOn}\ r\ s \wedge \text{isTransitiveOn}\ r\ s \wedge \text{isAntisymmetricOn}\ r\ s$

declare $\{hol\}$ rename function isPartialOrderOn = lem_is_partial_order_on

assert $is\_partialorder_0$: isPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (2,\ 2),\ (3,\ 3),\ (4,\ 4)\}$) $\{2,\ 3,\ 4\}$

assert $is\_partialorder_1$: $\neg$ (isPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2),\ (2,\ 2),\ (3,\ 3),\ (4,\ 4)\}$) $\{2,\ 3,\ 4\}$)

assert $is\_partialorder_2$: $\neg$ (isPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (2,\ 2),\ (3,\ 3)\}$) $\{2,\ 3,\ 4\}$)
assert $is\_partialorder_3$: $\neg$ (isPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (2,\ 2),\ (3,\ 3),\ (4,\ 4)\}$) $\{2,\ 3,\ 4\}$)

val $isStrictPartialOrderOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isStrictPartialOrderOn\ r\ s\ =\ \text{isIrreflexiveOn}\ r\ s \wedge \text{isTransitiveOn}\ r\ s$

declare $\{hol\}$ rename function isStrictPartialOrderOn = lem_is_strict_partial_order_on

lemma $isStrictPartialOrderOn\_antisym$: $(\forall\ r\ s.\ \text{isStrictPartialOrderOn}\ r\ s \longrightarrow \text{isAntisymmetricOn}\ r\ s)$

assert $is\_strict\_partialorder\_on_0$: isStrictPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}))\}$) $\{2,\ 3,\ 4\}$
assert $is\_strict\_partialorder\_on_1$: isStrictPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4),\ (2,\ 4)\}$) $\{2,\ 3,\ 4\}$

assert $is\_strict\_partialorder\_on_2$: $\neg$ (isStrictPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 4)\}$) $\{2,\ 3,\ 4\}$)

assert $is\_strict\_partialorder\_on_3$: $\neg$ (isStrictPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3,\ 2)\}$) $\{2,\ 3,\ 4\}$)

assert $is\_strict\_partialorder\_on_4$: $\neg$ (isStrictPartialOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (2,\ 2)\}$) $\{2,\ 3,\ 4\}$)

val $isStrictPartialOrder$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$

let $isStrictPartialOrder\ r\ =\ \text{isIrreflexive}\ r \wedge \text{isTransitive}\ r$

declare $\{hol\}$ rename function isStrictPartialOrder $=$ lem_is_strict_partial_order

assert $is\_strict\_partialorder_0$: isStrictPartialOrder (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}))\}$)
assert $is\_strict\_partialorder_1$: isStrictPartialOrder (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3, 4),\ (2, 4)\}$)
assert $is\_strict\_partialorder_2$: $\neg$ (isStrictPartialOrder (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3, 4)\}$))
assert $is\_strict\_partialorder_3$: $\neg$ (isStrictPartialOrder (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3, 2)\}$))
assert $is\_strict\_partialorder_4$: $\neg$ (isStrictPartialOrder (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (2, 2)\}$))

val $isPartialOrder$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml,\ coq\}$ $isPartialOrder\ r\ =\ \text{isReflexive}\ r \wedge \text{isTransitive}\ r \wedge \text{isAntisymmetric}\ r$

declare $\{hol\}$ rename function isPartialOrder $=$ lem_is_partial_order

```
(* ----------------------- *)
(* total / linear orders   *)
(* ----------------------- *)
```

val $isTotalOrderOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isTotalOrderOn\ r\ s\ =\ \text{isPartialOrderOn}\ r\ s \wedge \text{isTotalOn}\ r\ s$

declare $\{hol\}$ rename function isTotalOrderOn $=$ lem_is_total_order_on

val $isStrictTotalOrderOn$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \mathbb{B}$
let $isStrictTotalOrderOn\ r\ s\ =\ \text{isStrictPartialOrderOn}\ r\ s \wedge \text{isTrichotomousOn}\ r\ s$

declare $\{hol\}$ rename function isStrictTotalOrderOn $=$ lem_is_strict_total_order_on

val $isTotalOrder$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml,\ coq\}$ $isTotalOrder\ r\ =\ \text{isPartialOrder}\ r \wedge \text{isTotal}\ r$

declare $\{hol\}$ rename function isTotalOrder $=$ lem_is_total_order

val $isStrictTotalOrder$: $\forall\ \alpha.\ SetType\ \alpha,\ Eq\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \mathbb{B}$
let $\sim\{ocaml,\ coq\}$ $isStrictTotalOrder\ r\ =\ \text{isStrictPartialOrder}\ r \wedge \text{isTrichotomous}\ r$

declare $\{hol\}$ rename function isStrictTotalOrder $=$ lem_is_strict_total_order

assert $is\_totalorder\_on_0$: isTotalOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})), (2, 2), (3, 3), (4, 4)\}$) $\{2, 3\}$
assert $is\_totalorder\_on_1$: $\neg$ (isTotalOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT})), (2, 2), (3, 3), (4, 4)\}$) $\{2, 3, 4\}$)

assert $is\_totalorder\_on_2$: $\neg$ (isTotalOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}))\}$) $\{2, 3\}$)

assert $is\_strict\_totalorder\_on_0$: isStrictTotalOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}))\}$) $\{2, 3\}$
assert $is\_strict\_totalorder\_on_1$: $\neg$ (isStrictTotalOrderOn (relFromSet $\{((2 : \text{NAT}),\ (3 : \text{NAT}))\}$) $\{2, 3, 4\}$)

```
(* ========================================================================== *)
(* closures                                                                   *)
(* ========================================================================== *)

(* ----------------------- *)
(* transitive closure      *)
(* ----------------------- *)
```

val $\mathit{transitiveClosure}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$
val $\mathit{transitiveClosureByEq}$: $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$
val $\mathit{transitiveClosureByCmp}$: $\forall\ \alpha.\ (\alpha\ *\ \alpha\ \rightarrow\ \alpha\ *\ \alpha\ \rightarrow\ \text{ORDERING})\ \rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$

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 $\{\mathit{coq}\}$ $\mathit{transitiveClosure}$ = transitiveClosureByEq (=)
let inline $\{\mathit{ocaml}\}$ $\mathit{transitiveClosure}$ = transitiveClosureByCmp setElemCompare

lemma $\mathit{transitiveClosure\_spec}_1$: $(\forall\ r.\ \text{isSubrel}\ r\ (\text{transitiveClosure}\ r))$
lemma $\mathit{transitiveClosure\_spec}_2$: $(\forall\ r.\ \text{isTransitive}\ (\text{transitiveClosure}\ r))$
lemma $\mathit{transitiveClosure\_spec}_3$: $(\forall\ r_1\ r_2.\ ((\text{isTransitive}\ r_2) \wedge (\text{isSubrel}\ r_1\ r_2)) \longrightarrow \text{isSubrel}\ (\text{transitiveClosure}\ r_1)\ r_2)$

lemma $\mathit{transitiveClosure\_spec}_4$: $(\forall\ r.\ \text{isTransitive}\ r \longrightarrow (\text{transitiveClosure}\ r = r))$

assert $\mathit{transitive\_closure}_0$: $(\text{transitiveClosure}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3, 4)\}) =$
$\text{relFromSet}\ \{(2, 3),\ (2, 4),\ (3, 4)\})$
assert $\mathit{transitive\_closure}_1$: $(\text{transitiveClosure}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3, 4),\ (4, 5),\ (7,\ 9)\}) =$
$\text{relFromSet}\ \{(2, 3),\ (2, 4),\ (2, 5),\ (3, 4),\ (3, 5),\ (4, 5),\ (7,\ 9)\})$

```
(* ----------------------- *)
(* transitive closure step *)
(* ----------------------- *)
```

val $\mathit{transitiveClosureAdd}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \alpha\ \rightarrow\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$

let $\mathit{transitiveClosureAdd}\ x\ y\ r$ =
$(\text{relUnion}\ (\text{relAdd}\ x\ y\ r)\ (\text{relUnion}\ (\text{relFromSet}\ \{(x,\ z) \mid \forall\ z \in \text{relRange}\ r \mid \text{inRel}\ y\ z\ r\})$
$(\text{relFromSet}\ \{(z,\ y) \mid \forall\ z \in \text{relDomain}\ r \mid \text{inRel}\ z\ x\ r\})))$

declare $\{\mathit{hol}\}$ rename function transitiveClosureAdd = tc_insert

lemma $\mathit{transitive\_closure\_add\_thm}$: $\forall\ x\ y\ r.\ \text{isTransitive}\ r \longrightarrow (\text{transitiveClosureAdd}\ x\ y\ r = \text{transitiveClosure}\ (\text{relAdd}\ x\ y\ r))$

assert $\mathit{transitive\_closure\_add}_0$: $\text{transitiveClosureAdd}\ (2 : \text{NAT})\ (3 : \text{NAT})\ \{\} = \text{relFromSet}\ \{(2,\ 3)\}$
assert $\mathit{transitive\_closure\_add}_1$: $\text{transitiveClosureAdd}\ (3 : \text{NAT})\ (4 : \text{NAT})\ \{(2, 3)\} = \text{relFromSet}\ \{(2,\ 3),\ (3, 4),\ (2, 4)\}$

assert $\mathit{transitive\_closure\_add}_2$: $\text{transitiveClosureAdd}\ (4 : \text{NAT})\ (5 : \text{NAT})\ \{(2,\ 3),\ (3, 4),\ (2, 4)\} =$
$\text{relFromSet}\ \{(2,\ 3),\ (3, 4),\ (2, 4),\ (4, 5),\ (2, 5),\ (3, 5)\}$

```
(* ========================================================================== *)
(* reflexiv closures                                                          *)
(* ========================================================================== *)
```

val $\mathit{reflexivTransitiveClosureOn}$: $\forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \mathit{Eq}\ \alpha\ \Rightarrow\ \text{REL}\ \alpha\ \alpha\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{REL}\ \alpha\ \alpha$
let $\mathit{reflexivTransitiveClosureOn}\ r\ s$ = transitiveClosure (relUnion r (relIdOn s))
declare $\{\mathit{hol}\}$ rename function reflexivTransitiveClosureOn = reflexiv_transitive_closure_on

assert $\mathit{reflexiv\_transitive\_closure}_0$: $(\text{reflexivTransitiveClosureOn}\ (\text{relFromSet}\ \{((2 : \text{NAT}),\ (3 : \text{NAT})),\ (3, 4)\})\ \{2, 3, 4\} =$
$\text{relFromSet}\ \{(2, 3),\ (2, 4),\ (3, 4),\ (2, 2),\ (3, 3),\ (4, 4)\})$

val $\mathit{reflexivTransitiveClosure}\ :\ \forall\ \alpha.\ \mathit{SetType}\ \alpha,\ \ \mathit{Eq}\ \alpha\ \Rightarrow\ \textsc{rel}\ \alpha\ \alpha\ \rightarrow\ \textsc{rel}\ \alpha\ \alpha$
let $\sim\{\mathit{ocaml},\ \mathit{coq}\}\ \mathit{reflexivTransitiveClosure}\ r\ =\ \mathrm{transitiveClosure}\ (\mathrm{relUnion}\ r\ \mathrm{relId})$

17 Sorting

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



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

declare {isabelle, hol, ocaml, coq} rename module = lem_sorting

open import Bool Basic_classes Maybe List Num

open import {isabelle} ~~/src/HOL/Library/Permutation
open import {coq} Coq.Lists.List
open import {hol} sortingTheory permLib
open import {isabelle} $LIB_DIR/Lem

(* ------------------------- *)
(* permutations              *)
(* ------------------------- *)

val isPermutation : ∀ α. Eq α ⇒ list α → list α → bool
val isPermutationBy : ∀ α. (α → α → bool) → list α → list α → bool

let rec isPermutationBy eq l1 l2 = match l1 with
  | [] → null l2
  | (x :: xs) → begin
      match deleteFirst (eq x) l2 with
        | Nothing → false
        | Just ys → 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 : (¬ (isPermutation [(2 : nat)] []))
assert perm_3 : (isPermutation [(2 : nat); 1; 3; 5; 4] [1; 2; 3; 4; 5])
assert perm_4 : (¬ (isPermutation [(2 : nat); 3; 3; 5; 4] [1; 2; 3; 4; 5]))
assert perm_5 : (¬ (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 : (∀ l. isPermutation l l)
lemma isPermutation_2 : (∀ l1 l2. isPermutation l1 l2 ⟷ isPermutation l2 l1)
lemma isPermutation_3 : (∀ l1 l2 l3. isPermutation l1 l2 ⟶ isPermutation l2 l3 ⟶ isPermutation l1 l3)
lemma isPermutation_4 : (∀ l1 l2. isPermutation l1 l2 ⟶ (length l1 = length l2))
```

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
  | [] $\rightarrow$ true
  | x_1 :: xs $\rightarrow$
    match xs with
      | [] $\rightarrow$ true
      | x_2 :: _ $\rightarrow$ $(cmp\ x_1\ x_2 \wedge \text{isSortedBy}\ cmp\ xs)$
    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 $isSorted_5$: ($\neg$ (isSorted [(3 : NAT); 2]))
assert $isSorted_6$: ($\neg$ (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
  | [] $\rightarrow$ $[e]$

```
  | x :: xs -> if cmp x e then x :: (insertBy cmp e xs) else (e :: x :: xs)
end
declare termination_argument insertBy = automatic

let inline insert = insertBy (≤)

let insertSortBy cmp l = List.foldl (fun l e -> insertBy cmp e l) [] l
let inline insertSort = insertSortBy (≤)

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 : (∀ l e cmp. ((∀ x y z. cmp x y ∧ cmp y z ⟶ cmp x z) ∧ isSortedBy cmp l) ⟶ isSortedBy cmp (insertBy cmp

lemma insertBy_2 : (∀ l e cmp. length (insertBy cmp e l) = length l + 1)
lemma insertBy_3 : (∀ l e_1 e_2 cmp. elem e_1 (insertBy cmp e_2 l) = ((e_1 = e_2) ∨ elem e_1 l))

lemma insertSort_1 : (∀ l cmp. isPermutation (insertSort l) l)
lemma insertSort_2 : (∀ l cmp. isSorted (insertSort l))


(* ----------------------- *)
(* general sorting         *)
(* ----------------------- *)

val sort : ∀ α. Ord α ⇒ LIST α -> LIST α
val sortBy : ∀ α. (α -> α -> 𝔹) -> LIST α -> LIST α
val sortByOrd : ∀ α. (α -> α -> ORDERING) -> LIST α -> LIST α

let inline sortBy = insertSortBy
declare isabelle target_rep function sortBy = 'sort_by'
declare hol target_rep function sortBy = 'QSORT'
declare ocaml target_rep function sortByOrd = 'List.sort'

let inline ~{ocaml} sort = sortBy (≤)
let inline {ocaml} sort = sortByOrd compare


assert sort_1 : (sort ([] : LIST NAT) = [])
assert sort_2 : (sort ([6; 4; 3; 8; 1; 2] : LIST NAT) = [1; 2; 3; 4; 6; 8])
assert sort_3 : (sort ([5; 4; 5; 2; 4] : LIST NAT) = [2; 4; 4; 5; 5])

lemma sort_4 : (∀ l cmp. isPermutation (sort l) l)
lemma sort_5 : (∀ l cmp. isSorted (sort l))
```

18 String

```
(**************************************************************************)
(* A library for strings                                                  *)
(**************************************************************************)

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

declare $\{ocaml, isabelle, hol, coq\}$ rename module $=$ lem_string

open import *Bool Basic_classes List*
open import $\{ocaml\}$ *Xstring*
open import $\{hol\}$ *stringTheory*
open import $\{coq\}$ *Coq.Strings.Ascii Coq.Strings.String*

```
(* ----------------------- *)
(* basic instantiations    *)
(* ----------------------- *)

(* set up the string and char types correctly for the backends and make
   sure that parsing and equality checks work *)
```

declare *ocaml* target_rep type CHAR $=$ `'char'`
declare *hol* target_rep type CHAR $=$ `'char'`
declare *isabelle* target_rep type CHAR $=$ `'char'`
declare *coq* target_rep type CHAR $=$ `'ascii'`

declare *ocaml* target_rep type STRING $=$ `'string'`
declare *hol* target_rep type STRING $=$ `'string'`
declare *isabelle* target_rep type STRING $=$ `'string'`
declare *coq* target_rep type STRING $=$ `'string'`

assert $char\_simple_0$: $\neg$ (#'0' = ((#'1') : CHAR))
assert $char\_simple_1$: $\neg$ (#'X' = #'Y')
assert $char\_simple_2$: $\neg$ (#'$\backslash xAF$' = #'$\backslash x00$')
assert $char\_simple_3$: $\neg$ (#'' = #'@')
assert $char\_simple_4$: $\neg$ (#'$\backslash\backslash$' = #'$\backslash n$')
assert $char\_simple_5$: (#'$\backslash x20$' = #'')
assert $char\_simple_6$: $\neg$ ([#'$\backslash x20$'; #''; #'$\backslash x60$'; #'$\backslash x27$'; #'$\sim$'; #'$\backslash\backslash$'] = [])

assert $string\_simple_0$: $\neg$ ("*Hello*" = ("*Goodby*" : STRING))
assert $string\_simple_1$: $\neg$ ("$Hello\backslash nWorld$" = "$Goodby\backslash x20!$")
assert $string\_simple_2$: $\neg$ ("$123\_\backslash\backslash\backslash t - +!?X\_\&$" = "!"")
assert $string\_simple_3$: ("$HelloWorld$" = ("$Hello\backslash x20World$" : STRING))

```
(* ------------------------------------------- *)
(* translations between strings and char lists *)
(* ------------------------------------------- *)
```

val *toCharList* : STRING $\rightarrow$ LIST CHAR
declare *ocaml* target_rep function toCharList $=$ `'Xstring.explode'`
declare *hol* target_rep function toCharList $=$ `'EXPLODE'`
declare *isabelle* target_rep function toCharList s $=$ `''`s
declare *coq* target_rep function toCharList $=$ `'string_to_char_list'` `(* TODO: check *)`

assert $toCharList_0$: (toCharList "*Hello*" = [#'H'; #'e'; #'l'; #'l'; #'o'])

```
assert toCharList_1 : (toCharList "H\nA" = [#'H'; #'\n'; #'A'])

val toString : LIST CHAR → STRING
declare ocaml target_rep function toString = 'Xstring.implode'
declare hol target_rep function toString = 'IMPLODE'
declare isabelle target_rep function toString s = ''s
declare coq target_rep function toString = 'string_from_char_list' (* TODO: check *)

assert toString_0 : (toString [#'H'; #'e'; #'l'; #'l'; #'o'] = "Hello")
assert toString_1 : (toString [#'H'; #'\n'; #'A'] = "H\nA")


(* ----------------------- *)
(* generating strings      *)
(* ----------------------- *)

val makeString : NAT → CHAR → STRING
let makeString len c = toString (replicate len c)
declare ocaml target_rep function makeString = 'String.make'
declare isabelle target_rep function makeString = 'List.replicate'
declare hol target_rep function makeString = 'REPLICATE'
declare coq target_rep function makeString = 'string_make_string'

assert makeString_0 : (makeString 0 #'a' = "")
assert makeString_1 : (makeString 5 #'a' = "aaaaa")
assert makeString_2 : (makeString 3 #'c' = "ccc")

(* ----------------------- *)
(* length                  *)
(* ----------------------- *)

val stringLength : STRING → NAT
declare hol target_rep function stringLength = 'STRLEN'
declare ocaml target_rep function stringLength = 'String.length'
declare isabelle target_rep function stringLength = 'List.length'
declare coq target_rep function stringLength = 'String.length' (* TODO: check *)

assert stringLength_0 : (stringLength "" = 0)
assert stringLength_1 : (stringLength "abc" = 3)
assert stringLength_2 : (stringLength "123456" = 6)

(* ----------------------- *)
(* string concatenation    *)
(* ----------------------- *)

val ↑ [stringAppend] : STRING → STRING → STRING
let inline stringAppend x y = (toString ((toCharList x) ++ (toCharList y)))
declare ocaml target_rep function stringAppend = infix '↑'
declare hol target_rep function stringAppend = 'STRCAT'
declare isabelle target_rep function stringAppend = infix '@'
declare coq target_rep function stringAppend = 'String.append'

assert stringAppend_0 : (↑ "Hello" ↑ "" "World!" = "HelloWorld!")

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

```
val stringEquality : STRING → STRING → 𝔹
declare coq target_rep function stringEquality = 'string_equal'
let inline {ocaml, hol, isabelle} stringEquality = unsafe_structural_equality

instance (Eq STRING)
  let = = stringEquality
  let <> l r = ¬ (stringEquality l r)
end


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

val string_case : ∀ α. STRING → α → (CHAR → STRING → α) → α

let string_case s c_empty c_cons =
  match (toCharList s) with
    | [] → c_empty
    | c :: cs → c_cons c (toString cs)
  end
declare ocaml target_rep function string_case = 'Xstring.string_case'
declare hol target_rep function string_case = 'string_case'
declare isabelle target_rep function string_case s c_e c_c = 'list_case' c_e c_c s

val empty_string : STRING
let inline empty_string = ""

assert empty_string_0 : (empty_string = "")
assert empty_string_1 : ¬ (empty_string = "xxx")

val cons_string : CHAR → STRING → STRING
let inline cons_string c s = toString (c :: toCharList s)

assert string_cons_0 : (cons_string #'a' empty_string = "a")
assert string_cons_1 : (cons_string #'x' "yz" = "xyz")

declare ocaml target_rep function cons_string = 'Xstring.cons_string'
declare hol target_rep function cons_string = 'STRING'
declare isabelle target_rep function cons_string = infix '#'


declare pattern_match exhaustive STRING = [ empty_string; cons_string ] string_case

assert string_patterns_0 : (
  match "" with
    | empty_string → true
    | _ → false
  end
)

assert string_patterns_1 : (
  match "abc" with
    | empty_string → ""
    | cons_string c s → (↑ makeString 5 c s)
  end = "aaaaabc"
)
```

19 Word

```
(*****************************************************************************)
(* A generic library for machine words.                                      *)
(*****************************************************************************)
```

declare {*isabelle*, *coq*, *hol*, *ocaml*} rename module = Lem_word

open import *Bool Maybe Num Basic_classes List*

open import {*isabelle*} ~~/*src/HOL/Word/Word*
open import {*hol*} *wordsTheory wordsLib*

```
(* ========================================================================== *)
(* Define general purpose word, i.e. sequences of bits of arbitrary length    *)
(* ========================================================================== *)
```

type BITSEQUENCE = BITSEQ of
  MAYBE NAT * (* length of the sequence, Nothing means infinite length *)
  $\mathbb{B}$ * (* sign of the word, used to fill up after concrete value is exhausted *)
  LIST $\mathbb{B}$ (* the initial part of the sequence, least significant bit first *)

val *boolListFrombitSeq* : NAT → BITSEQUENCE → LIST $\mathbb{B}$

let rec *boolListFrombitSeqAux n s bl* =
  if $n = 0$ then [] else
  match *bl* with
    | [] → replicate *n s*
    | *b* :: *bl′* → *b* :: (boolListFrombitSeqAux ($n-1$) *s bl′*)
  end
declare termination_argument boolListFrombitSeqAux = automatic

let *boolListFrombitSeq n* (BitSeq _ *s bl*) = boolListFrombitSeqAux *n s bl*

assert *boolListFrombitSeq*$_0$: boolListFrombitSeq 5 (BitSeq Nothing false [true; false; true]) = [true; false; true; false; false]

assert *boolListFrombitSeq*$_1$: boolListFrombitSeq 5 (BitSeq Nothing true [true; false; true]) = [true; false; true; true; true]

assert *boolListFrombitSeq*$_2$: boolListFrombitSeq 2 (BitSeq Nothing true [true; false; true]) = [true; false]

lemma *boolListFrombitSeq_len* : ∀ *n bs*. (List.length (boolListFrombitSeq *n bs*) = *n*)

val *bitSeqFromBoolList* : LIST $\mathbb{B}$ → MAYBE BITSEQUENCE
let *bitSeqFromBoolList bl* =
  match dest_init *bl* with
    | Nothing → Nothing
    | Just (*bl′*, *s*) → Just (BitSeq (Just (List.length *bl*)) *s bl′*)
  end

assert *bitSeqFromBoolList*$_0$: bitSeqFromBoolList [] = Nothing
assert *bitSeqFromBoolList*$_1$: bitSeqFromBoolList [true; false; false] = Just (BitSeq (Just 3) false [true; false])

assert *bitSeqFromBoolList*$_2$: bitSeqFromBoolList [true; false; true] = Just (BitSeq (Just 3) true [true; false])

lemma *bitSeqFromBoolList_nothing* : ∀ *bl*. (isNothing (bitSeqFromBoolList *bl*) ⟷ List.null *bl*)

```
(* cleans up the representation of a bitSequence without changing its semantics *)
val cleanBitSeq : bitSequence → bitSequence
let cleanBitSeq (BitSeq len s bl) = match len with
  | Nothing → (BitSeq len s (List.reverse (dropWhile ((=) s) (List.reverse bl))))
  | Just n → (BitSeq len s (List.reverse (dropWhile ((=) s) (List.reverse (List.take (n−1) bl)))))
end

assert cleanBitSeq_0 : cleanBitSeq (BitSeq Nothing false [true; false; true; false; false]) = (BitSeq Nothing false [true; false; true])

assert cleanBitSeq_1 : cleanBitSeq (BitSeq Nothing true [true; false; true; false; false]) = (BitSeq Nothing true [true; false; true; false; f

assert cleanBitSeq_2 : cleanBitSeq (BitSeq (Just 4) true [true; false; true; false; false]) = (BitSeq (Just 4) true [true; false])



val bitSeqTestBit : bitSequence → nat → maybe 𝔹
let bitSeqTestBit (BitSeq len s bl) pos =
  match len with
    | Nothing → if pos < length bl then index bl pos else Just s
    | Just l → if (pos ≥ l) then Nothing else
              if (pos = (l − 1) ∨ pos ≥ length bl) then Just s else
              index bl pos
  end

val bitSeqSetBit : bitSequence → nat → 𝔹 → bitSequence
let bitSeqSetBit (BitSeq len s bl) pos v =
  let bl′ = if (pos < length bl) then bl else bl ++ replicate pos s in
  let bl′′ = List.update bl′ pos v in
  let bs′ = BitSeq len s bl′′ in
  cleanBitSeq bs′


val resizeBitSeq : maybe nat → bitSequence → bitSequence
let resizeBitSeq new_len bs =
  let (BitSeq len s bl) = cleanBitSeq bs in
  let shorten_opt = match (new_len, len) with
     | (Nothing, _) → Nothing
     | (Just l1, Nothing) → Just l1
     | (Just l1, Just l2) → if (l1 < l2) then Just l1 else Nothing
  end in
  match shorten_opt with
    | Nothing → BitSeq new_len s bl
    | Just l1 → (
        let bl′ = List.take l1 (bl ++ [s]) in
        match dest_init bl′ with
          | Nothing → (BitSeq len s bl) (* do nothing if size 0 is requested *)
          | Just (bl′′, s′) → cleanBitSeq (BitSeq new_len s′ bl′′)
 end)
  end

assert resizeBitSeq_0 : (resizeBitSeq Nothing (BitSeq (Just 5) true [false; false]) = (BitSeq Nothing true [false; false]))

assert resizeBitSeq_1 : (resizeBitSeq (Just 3) (BitSeq Nothing true [false; true; false; false]) = (BitSeq (Just 3) false [false; true]))

assert resizeBitSeq_2 : (resizeBitSeq (Just 3) (BitSeq Nothing false [false; true; true; false]) = (BitSeq (Just 3) true [false]))

assert resizeBitSeq_3 : (resizeBitSeq (Just 3) (BitSeq (Just 10) false [false; true; true; false]) = (BitSeq (Just 3) true [false]))
```

```
assert resizeBitSeq_4 : (resizeBitSeq (Just 10) (BitSeq (Just 3) false [false; true; true; false]) = (BitSeq (Just 10) false [false; true]))


val bitSeqNot : BITSEQUENCE → BITSEQUENCE
let bitSeqNot (BitSeq len s bl) = BitSeq len (¬ s) (List.map (fun b → ¬ b) bl)

assert bitSeqNot_0 : (bitSeqNot (BitSeq (Just 2) true [false; true])) = BitSeq (Just 2) false [true; false]

val bitSeqBinop : (𝔹 → 𝔹 → 𝔹) → BITSEQUENCE → BITSEQUENCE → BITSEQUENCE

val bitSeqBinopAux : (𝔹 → 𝔹 → 𝔹) → 𝔹 → LIST 𝔹 → 𝔹 → LIST 𝔹 → LIST 𝔹
let rec bitSeqBinopAux binop s1 bl1 s2 bl2 =
  match (bl1, bl2) with
    | ([], []) → []
    | (b1 :: bl1', []) → (binop b1 s2) :: bitSeqBinopAux binop s1 bl1' s2 []
    | ([], b2 :: bl2') → (binop s1 b2) :: bitSeqBinopAux binop s1 [] s2 bl2'
    | (b1 :: bl1', b2 :: bl2') → (binop b1 b2) :: bitSeqBinopAux binop s1 bl1' s2 bl2'
  end
declare termination_argument bitSeqBinopAux = automatic

let bitSeqBinop binop bs1 bs2 = (
  let (BitSeq len1 s1 bl1) = cleanBitSeq bs1 in
  let (BitSeq len2 s2 bl2) = cleanBitSeq bs2 in

  let len = match (len1, len2) with
    | (Just l1, Just l2) → Just (max l1 l2)
    | _ → Nothing
  end in
  let s = binop s1 s2 in
  let bl = bitSeqBinopAux binop s1 bl1 s2 bl2 in
  cleanBitSeq (BitSeq len s bl)
)

let bitSeqAnd = bitSeqBinop (∧)
let bitSeqOr = bitSeqBinop (∨)
let bitSeqXor = bitSeqBinop xor

val bitSeqShiftLeft : BITSEQUENCE → NAT → BITSEQUENCE
let bitSeqShiftLeft (BitSeq len s bl) n = cleanBitSeq (BitSeq len s (replicate n false ++ bl))

val bitSeqArithmeticShiftRight : BITSEQUENCE → NAT → BITSEQUENCE
let bitSeqArithmeticShiftRight bs n =
  let (BitSeq len s bl) = cleanBitSeq bs in
  cleanBitSeq (BitSeq len s (drop n bl))

val bitSeqLogicalShiftRight : BITSEQUENCE → NAT → BITSEQUENCE
let bitSeqLogicalShiftRight bs n =
  if (n = 0) then cleanBitSeq bs else
  let (BitSeq len s bl) = cleanBitSeq bs in
  match len with
    | Nothing → cleanBitSeq (BitSeq len s (drop n bl))
    | Just l → cleanBitSeq (BitSeq len false ((drop n bl) ++ replicate l s))
  end


(* integerFromBoolList sign bl creates an integer from a list of bits
   (least significant bit first) and an explicitly given sign bit.
```

```
   It uses two's complement encoding. *)
val integerFromBoolList : (𝔹 * list 𝔹) → ℤ

let rec integerFromBoolListAux (acc : ℤ) (bl : list 𝔹) =
  match bl with
    | [] → acc
    | (true :: bl') → integerFromBoolListAux ((acc * 2) + 1) bl'
    | (false :: bl') → integerFromBoolListAux (acc * 2) bl'
  end
declare termination_argument integerFromBoolListAux = automatic

let integerFromBoolList (sign, bl) =
   if sign then
     −(integerFromBoolListAux 0 (List.reverseMap (fun b → ¬ b) bl) + 1)
   else integerFromBoolListAux 0 (List.reverse bl)

assert integerFromBoolList_0 : integerFromBoolList (false, [false; true; false]) = 2
assert integerFromBoolList_1 : integerFromBoolList (false, [false; true; false; true]) = 10
assert integerFromBoolList_2 : integerFromBoolList (true, [false; true; false; true]) = −6
assert integerFromBoolList_3 : integerFromBoolList (true, [false; true]) = −2
assert integerFromBoolList_4 : integerFromBoolList (true, [true; false]) = −3

(* [boolListFromInteger i] creates a sign bit and a list of booleans from an integer. The len_opt
tells it when to stop.*)
val boolListFromInteger : ℤ → 𝔹 * list 𝔹

let rec boolListFromNatural acc (remainder : ℕ) =
 if (remainder > 0) then
   (boolListFromNatural (((remainder mod 2) = 1) :: acc)
      (remainder / 2))
 else
   List.reverse acc
declare termination_argument boolListFromNatural = automatic

let boolListFromInteger (i : ℤ) =
  if (i < 0) then
    (true, List.map (fun b → ¬ b) (boolListFromNatural [] (naturalFromInteger (−(i + 1)))))
  else
    (false, boolListFromNatural [] (naturalFromInteger i))

assert boolListFromInteger_0 : boolListFromInteger 2 = (false, [false; true])
assert boolListFromInteger_1 : boolListFromInteger 10 = (false, [false; true; false; true])
assert boolListFromInteger_2 : boolListFromInteger (−6) = (true, [false; true; false])
assert boolListFromInteger_3 : boolListFromInteger (−2) = (true, [false])
assert boolListFromInteger_4 : boolListFromInteger (−3) = (true, [true; false])

lemma boolListFromInteger_inverse_1 : (∀ i. integerFromBoolList (boolListFromInteger i) = i)
lemma boolListFromInteger_inverse_2 : (∀ s bl i. boolListFromInteger (integerFromBoolList (s, bl)) =
   (s, List.reverse (dropWhile ((=) s) (List.reverse bl))))


(* [bitSeqFromInteger len_opt i] encodes [i] as a bitsequence with [len_opt] bits. If there
are not enough
   bits, truncation happens *)
val bitSeqFromInteger : maybe nat → ℤ → bitSequence
let bitSeqFromInteger len_opt i =
  let (s, bl) = boolListFromInteger i in
  resizeBitSeq len_opt (BitSeq Nothing s bl)
```

```
assert bitSeqFromInteger_0 : (bitSeqFromInteger Nothing 5 = BitSeq Nothing false [true; false; true])
assert bitSeqFromInteger_1 : (bitSeqFromInteger (Just 2) 5 = BitSeq (Just 2) false [true])
assert bitSeqFromInteger_2 : (bitSeqFromInteger Nothing (−5) = BitSeq Nothing true [true; true; false])
assert bitSeqFromInteger_3 : (bitSeqFromInteger (Just 3) (−5) = BitSeq (Just 3) false [true; true])
assert bitSeqFromInteger_4 : (bitSeqFromInteger (Just 2) (−5) = BitSeq (Just 2) true [])
assert bitSeqFromInteger_5 : (bitSeqFromInteger (Just 5) (−5) = BitSeq (Just 5) true [true; true; false])


val integerFromBitSeq : bitSequence → ℤ
let integerFromBitSeq bs =
  let (BitSeq len s bl) = cleanBitSeq bs in
  integerFromBoolList (s, bl)


assert integerFromBitSeq_0 : (integerFromBitSeq (BitSeq Nothing false [true; false; true]) = 5)
assert integerFromBitSeq_1 : (integerFromBitSeq (BitSeq (Just 2) false [true]) = 1)
assert integerFromBitSeq_2 : (integerFromBitSeq (BitSeq Nothing true [true; true; false]) = (−5))
assert integerFromBitSeq_3 : (integerFromBitSeq (BitSeq (Just 2) true [true; true; false]) = (−1))

lemma integerFromBitSeq_inv : (∀ i. integerFromBitSeq (bitSeqFromInteger Nothing i) = i)
assert integerFromBitSeq_inv_0 : (integerFromBitSeq (bitSeqFromInteger Nothing 10)) = 10
assert integerFromBitSeq_inv_1 : (integerFromBitSeq (bitSeqFromInteger Nothing (−1932))) = (−1932)
assert integerFromBitSeq_inv_2 : (integerFromBitSeq (bitSeqFromInteger Nothing 343)) = 343


(* Now we can via translation to integers map arithmetic operations to bitSequences *)

val bitSeqArithUnaryOp : (ℤ → ℤ) → bitSequence → bitSequence
let bitSeqArithUnaryOp uop bs =
  let (BitSeq len _ _) = bs in
  bitSeqFromInteger len (uop (integerFromBitSeq bs))

val bitSeqArithBinOp : (ℤ → ℤ → ℤ) → bitSequence → bitSequence → bitSequence
let bitSeqArithBinOp binop bs_1 bs_2 =
  let (BitSeq len_1 _ _) = bs_1 in
  let (BitSeq len_2 _ _) = bs_2 in
  let len = match (len_1, len_2) with
    | (Just l_1, Just l_2) → Just (max l_1 l_2)
    | _ → Nothing
  end in
  bitSeqFromInteger len (binop (integerFromBitSeq bs_1) (integerFromBitSeq bs_2))

val bitSeqArithBinTest : ∀ α. (ℤ → ℤ → α) → bitSequence → bitSequence → α
let bitSeqArithBinTest binop bs_1 bs_2 = binop (integerFromBitSeq bs_1) (integerFromBitSeq bs_2)


(* now instantiate the number interface for bit-sequences *)

val bitSeqFromNumeral : numeral → bitSequence
let inline bitSeqFromNumeral n = bitSeqFromInteger Nothing (integerFromNumeral n)

instance (Numeral bitSequence)
  let fromNumeral n = bitSeqFromNumeral n
end

val bitSeqEq : bitSequence → bitSequence → 𝔹
let inline bitSeqEq = unsafe_structural_equality
```

instance (Eq BITSEQUENCE)
  let $=$ = bitSeqEq
  let $<>$ n_1 n_2 = $\neg$ (bitSeqEq n_1 n_2)
end

val $bitSeqLess$: BITSEQUENCE $\rightarrow$ BITSEQUENCE $\rightarrow$ $\mathbb{B}$
let $bitSeqLess$ bs_1 bs_2 = bitSeqArithBinTest ($<$) bs_1 bs_2

val $bitSeqLessEqual$: BITSEQUENCE $\rightarrow$ BITSEQUENCE $\rightarrow$ $\mathbb{B}$
let $bitSeqLessEqual$ bs_1 bs_2 = bitSeqArithBinTest ($\leq$) bs_1 bs_2

val $bitSeqGreater$: BITSEQUENCE $\rightarrow$ BITSEQUENCE $\rightarrow$ $\mathbb{B}$
let $bitSeqGreater$ bs_1 bs_2 = bitSeqArithBinTest ($>$) bs_1 bs_2

val $bitSeqGreaterEqual$: BITSEQUENCE $\rightarrow$ BITSEQUENCE $\rightarrow$ $\mathbb{B}$
let $bitSeqGreaterEqual$ bs_1 bs_2 = bitSeqArithBinTest ($\geq$) bs_1 bs_2

val $bitSeqCompare$: BITSEQUENCE $\rightarrow$ BITSEQUENCE $\rightarrow$ ORDERING
let $bitSeqCompare$ bs_1 bs_2 = bitSeqArithBinTest compare bs_1 bs_2

instance (Ord BITSEQUENCE)
  let $compare$ = bitSeqCompare
  let $<$ = bitSeqLess
  let $<=$ = bitSeqLessEqual
  let $>$ = bitSeqGreater
  let $>=$ = bitSeqGreaterEqual
end

instance ($SetType$ BITSEQUENCE)
  let $setElemCompare$ = bitSeqCompare
end

(* arithmetic negation, don't mix up with bitwise negation *)
val $bitSeqNegate$: BITSEQUENCE $\rightarrow$ BITSEQUENCE
let $bitSeqNegate$ bs = bitSeqArithUnaryOp integerNegate bs

instance ($NumNegate$ BITSEQUENCE)
  let $\sim$ = bitSeqNegate
end

val $bitSeqAdd$: BITSEQUENCE $\rightarrow$ BITSEQUENCE $\rightarrow$ BITSEQUENCE
let $bitSeqAdd$ bs_1 bs_2 = bitSeqArithBinOp ($+$) bs_1 bs_2

instance ($NumAdd$ BITSEQUENCE)
  let $+$ = bitSeqAdd
end

val $bitSeqMinus$: BITSEQUENCE $\rightarrow$ BITSEQUENCE $\rightarrow$ BITSEQUENCE
let $bitSeqMinus$ bs_1 bs_2 = bitSeqArithBinOp ($-$) bs_1 bs_2

instance ($NumMinus$ BITSEQUENCE)
  let $-$ = bitSeqMinus
end

val $bitSeqSucc$: BITSEQUENCE $\rightarrow$ BITSEQUENCE
let $bitSeqSucc$ bs = bitSeqArithUnaryOp succ bs

```
instance (NumSucc bitSequence)
  let succ = bitSeqSucc
end

val bitSeqPred : bitSequence → bitSequence
let bitSeqPred bs = bitSeqArithUnaryOp pred bs

instance (NumPred bitSequence)
  let pred = bitSeqPred
end

val bitSeqMult : bitSequence → bitSequence → bitSequence
let bitSeqMult bs1 bs2 = bitSeqArithBinOp integerMult bs1 bs2

instance (NumMult bitSequence)
  let * = bitSeqMult
end


val bitSeqPow : bitSequence → nat → bitSequence
let bitSeqPow bs n = bitSeqArithUnaryOp (fun i → integerPow i n) bs

instance ( NumPow bitSequence )
  let ** = bitSeqPow
end

val bitSeqDiv : bitSequence → bitSequence → bitSequence
let bitSeqDiv bs1 bs2 = bitSeqArithBinOp integerDiv bs1 bs2

instance ( NumIntegerDivision bitSequence )
  let div = bitSeqDiv
end

instance ( NumDivision bitSequence )
  let / = bitSeqDiv
end

val bitSeqMod : bitSequence → bitSequence → bitSequence
let bitSeqMod bs1 bs2 = bitSeqArithBinOp integerMod bs1 bs2

instance ( NumRemainder bitSequence )
  let mod = bitSeqMod
end

val bitSeqMin : bitSequence → bitSequence → bitSequence
let bitSeqMin bs1 bs2 = bitSeqArithBinOp integerMin bs1 bs2

val bitSeqMax : bitSequence → bitSequence → bitSequence
let bitSeqMax bs1 bs2 = bitSeqArithBinOp integerMax bs1 bs2

instance ( OrdMaxMin bitSequence )
  let max = bitSeqMax
  let min = bitSeqMin
end

assert bitSequence_test1 : (2 + (5 : bitSequence) = 7)
assert bitSequence_test2 : (8 − (7 : bitSequence) = 1)
assert bitSequence_test3 : (7 − (8 : bitSequence) = −1)
```

assert $bitSequence\_test_4$: $(7 * (8 : \text{BITSEQUENCE}) = 56)$
assert $bitSequence\_test_5$: $((7 : \text{BITSEQUENCE})^2 = 49)$
assert $bitSequence\_test_6$: $(\text{div } 11\ (4 : \text{BITSEQUENCE}) = 2)$
assert $bitSequence\_test6a$: $(\text{div } (-\ 11)\ (4 : \text{BITSEQUENCE}) = -3)$
assert $bitSequence\_test_7$: $(11\ /\ (4 : \text{BITSEQUENCE}) = 2)$
assert $bitSequence\_test7a$: $(-11\ /\ (4 : \text{BITSEQUENCE}) = -3)$
assert $bitSequence\_test_8$: $(11 \text{ mod } (4 : \text{BITSEQUENCE}) = 3)$
assert $bitSequence\_test8a$: $(-11 \text{ mod } (4 : \text{BITSEQUENCE}) = 1)$
assert $bitSequence\_test_9$: $(11 < (12 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{10}$: $(11 \leq (12 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{11}$: $(12 \leq (12 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{12}$: $(\neg\ (12 < (12 : \text{BITSEQUENCE})))$
assert $bitSequence\_test_{13}$: $(12 > (11 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{14}$: $(12 \geq (11 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{15}$: $(12 \geq (12 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{16}$: $(\neg\ (12 > (12 : \text{BITSEQUENCE})))$
assert $bitSequence\_test_{17}$: $(\text{min } 12\ (12 : \text{BITSEQUENCE}) = 12)$
assert $bitSequence\_test_{18}$: $(\text{min } 10\ (12 : \text{BITSEQUENCE}) = 10)$
assert $bitSequence\_test_{19}$: $(\text{min } 12\ (10 : \text{BITSEQUENCE}) = 10)$
assert $bitSequence\_test_{20}$: $(\text{max } 12\ (12 : \text{BITSEQUENCE}) = 12)$
assert $bitSequence\_test_{21}$: $(\text{max } 10\ (12 : \text{BITSEQUENCE}) = 12)$
assert $bitSequence\_test_{22}$: $(\text{max } 12\ (10 : \text{BITSEQUENCE}) = 12)$
assert $bitSequence\_test_{23}$: $(\text{succ } 12 = (13 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{24}$: $(\text{succ } 0 = (1 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{25}$: $(\text{pred } 12 = (11 : \text{BITSEQUENCE}))$
assert $bitSequence\_test_{26}$: $(\text{pred } 0 = -(1 : \text{BITSEQUENCE}))$

```
(* ========================================================================== *)
(* Interface for bitoperations                                                *)
(* ========================================================================== *)
```

class ($WordNot\ \alpha$)
  val $lnot$: $\alpha \rightarrow \alpha$
end

class ($WordAnd\ \alpha$)
  val $land$ [conjunction] : $\alpha \rightarrow \alpha \rightarrow \alpha$
end

class ($WordOr\ \alpha$)
  val lor [inclusive_or] : $\alpha \rightarrow \alpha \rightarrow \alpha$
end

class ($WordXor\ \alpha$)
  val $lxor$ [exclusive_or] : $\alpha \rightarrow \alpha \rightarrow \alpha$
end

class ($WordLsl\ \alpha$)
  val lsl [left_shift] : $\alpha \rightarrow \text{NAT} \rightarrow \alpha$
end

class ($WordLsr\ \alpha$)
  val lsr [logicial_right_shift] : $\alpha \rightarrow \text{NAT} \rightarrow \alpha$
end

```
class ( WordAsr α )
  val asr [arithmetic_right_shift] : α → NAT → α
end

(* ----------------------- *)
(* bitSequence             *)
(* ----------------------- *)

instance ( WordNot BITSEQUENCE)
  let lnot = bitSeqNot
end

instance ( WordAnd BITSEQUENCE)
  let land = bitSeqAnd
end

instance ( WordOr BITSEQUENCE)
  let lor = bitSeqOr
end

instance ( WordXor BITSEQUENCE)
  let lxor = bitSeqXor
end

instance (WordLsl BITSEQUENCE)
  let lsl = bitSeqShiftLeft
end

instance (WordLsr BITSEQUENCE)
  let lsr = bitSeqLogicalShiftRight
end

instance (WordAsr BITSEQUENCE)
  let asr = bitSeqArithmeticShiftRight
end

assert bitSequence_bittest1 : ((6 : BITSEQUENCE) land 5 = 4)
assert bitSequence_bittest2 : ((6 : BITSEQUENCE) lor 5 = 7)
assert bitSequence_bittest3 : ((6 : BITSEQUENCE) lxor 5 = 3)
assert bitSequence_bittest4 : ((12 : BITSEQUENCE) land 9 = 8)
assert bitSequence_bittest5 : ((12 : BITSEQUENCE) lor 9 = 13)
assert bitSequence_bittest6 : ((12 : BITSEQUENCE) lxor 9 = 5)

assert bitSequence_bittest7 : (lnot (12 : BITSEQUENCE) = −13)
assert bitSequence_bittest8 : (lnot (27 : BITSEQUENCE) = −28)
assert bitSequence_bittest9 : ((27 : BITSEQUENCE) lsl 0 = 27)
assert bitSequence_bittest10 : ((27 : BITSEQUENCE) lsl 1 = 54)
assert bitSequence_bittest11 : ((27 : BITSEQUENCE) lsl 2 = 108)
assert bitSequence_bittest12 : ((27 : BITSEQUENCE) lsl 3 = 216)
assert bitSequence_bittest13 : ((27 : BITSEQUENCE) lsr 0 = 27)
assert bitSequence_bittest14 : ((27 : BITSEQUENCE) lsr 1 = 13)
assert bitSequence_bittest15 : ((27 : BITSEQUENCE) lsr 2 = 6)
assert bitSequence_bittest16 : ((27 : BITSEQUENCE) lsr 3 = 3)
assert bitSequence_bittest17 : ((27 : BITSEQUENCE) asr 0 = 27)
assert bitSequence_bittest18 : ((27 : BITSEQUENCE) asr 1 = 13)
assert bitSequence_bittest19 : ((27 : BITSEQUENCE) asr 2 = 6)
assert bitSequence_bittest20 : ((27 : BITSEQUENCE) asr 3 = 3)
```

assert $bitSequence\_bittest_{21}$: $((-(27 : \text{BITSEQUENCE})) \text{ lsr } 0 = -(27))$
assert $bitSequence\_bittest_{22}$: $((-(27 : \text{BITSEQUENCE}) \text{ asr } 0) = -(27))$
assert $bitSequence\_bittest_{23}$: $((-(27 : \text{BITSEQUENCE})) \text{ lsr } 1 = -(14))$
assert $bitSequence\_bittest_{24}$: $((-(27 : \text{BITSEQUENCE})) \text{ asr } 1 = -(14))$

```
(* ---------------------- *)
(* int32                  *)
(* ---------------------- *)
```

val *int32Lnot* : $\text{INT}_{32} \rightarrow \text{INT}_{32}$
declare *ocaml* target_rep function int32Lnot = `'Int32.lognot'`
declare *hol* target_rep function int32Lnot w = (`'~'` w)
declare *isabelle* target_rep function int32Lnot w = (`'NOT'` w)
declare *coq* target_rep function int32Lnot = `'TODO'`

instance (*WordNot* INT_{32})
  let *lnot* = int32Lnot
end

val *int32Lor* : $\text{INT}_{32} \rightarrow \text{INT}_{32} \rightarrow \text{INT}_{32}$
declare *ocaml* target_rep function int32Lor = `'Int32.logor'`
declare *hol* target_rep function int32Lor = `'word_or'`
declare *isabelle* target_rep function int32Lor = infix `'OR'`
declare *coq* target_rep function int32Lor = `'TODO'`

instance (*WordOr* INT_{32})
  let *lor* = int32Lor
end

val *int32Lxor* : $\text{INT}_{32} \rightarrow \text{INT}_{32} \rightarrow \text{INT}_{32}$
declare *ocaml* target_rep function int32Lxor = `'Int32.logxor'`
declare *hol* target_rep function int32Lxor = `'word_xor'`
declare *isabelle* target_rep function int32Lxor = infix `'XOR'`
declare *coq* target_rep function int32Lxor = `'TODO'`

instance (*WordXor* INT_{32})
  let *lxor* = int32Lxor
end

val *int32Land* : $\text{INT}_{32} \rightarrow \text{INT}_{32} \rightarrow \text{INT}_{32}$
declare *ocaml* target_rep function int32Land = `'Int32.logand'`
declare *hol* target_rep function int32Land = `'word_and'`
declare *isabelle* target_rep function int32Land = infix `'AND'`
declare *coq* target_rep function int32Land = `'TODO'`

instance (*WordAnd* INT_{32})
  let *land* = int32Land
end

val *int32Lsl* : $\text{INT}_{32} \rightarrow \text{NAT} \rightarrow \text{INT}_{32}$
declare *ocaml* target_rep function int32Lsl = `'Int32.shift_left'`
declare *hol* target_rep function int32Lsl = `'word_lsl'`
declare *isabelle* target_rep function int32Lsl = infix `'<<'`
declare *coq* target_rep function int32Lsl = `'TODO'`

instance (*WordLsl* INT_{32})

```
  let lsl = int32Lsl
end

val int32Lsr : INT32 → NAT → INT32
declare ocaml target_rep function int32Lsr = 'Int32.shift_right_logical'
declare hol target_rep function int32Lsr = 'word_lsr'
declare isabelle target_rep function int32Lsr = infix '>>'
declare coq target_rep function int32Lsr = 'TODO'

instance (WordLsr INT32)
  let lsr = int32Lsr
end


val int32Asr : INT32 → NAT → INT32
declare ocaml target_rep function int32Asr = 'Int32.shift_right'
declare hol target_rep function int32Asr = 'word_asr'
declare isabelle target_rep function int32Asr = infix '>>>'
declare coq target_rep function int32Asr = 'TODO'

instance (WordAsr INT32)
  let asr = int32Asr
end


assert int32_bittest1 : ((6 : INT32) land 5 = 4)
assert int32_bittest2 : ((6 : INT32) lor 5 = 7)
assert int32_bittest3 : ((6 : INT32) lxor 5 = 3)
assert int32_bittest4 : ((12 : INT32) land 9 = 8)
assert int32_bittest5 : ((12 : INT32) lor 9 = 13)
assert int32_bittest6 : ((12 : INT32) lxor 9 = 5)

assert int32_bittest7 : (lnot (12 : INT32) = −13)
assert int32_bittest8 : (lnot (27 : INT32) = −28)
assert int32_bittest9 : ((27 : INT32) lsl 0 = 27)
assert int32_bittest10 : ((27 : INT32) lsl 1 = 54)
assert int32_bittest11 : ((27 : INT32) lsl 2 = 108)
assert int32_bittest12 : ((27 : INT32) lsl 3 = 216)
assert int32_bittest13 : ((27 : INT32) lsr 0 = 27)
assert int32_bittest14 : ((27 : INT32) lsr 1 = 13)
assert int32_bittest15 : ((27 : INT32) lsr 2 = 6)
assert int32_bittest16 : ((27 : INT32) lsr 3 = 3)
assert int32_bittest17 : ((27 : INT32) asr 0 = 27)
assert int32_bittest18 : ((27 : INT32) asr 1 = 13)
assert int32_bittest19 : ((27 : INT32) asr 2 = 6)
assert int32_bittest20 : ((27 : INT32) asr 3 = 3)
assert int32_bittest21 : ((−(27 : INT32)) lsr 0 = −(27))
assert int32_bittest22 : ((−(27 : INT32) asr 0) = −(27))
assert int32_bittest23 : ((−(27 : INT32)) lsr 2 = 1073741817)
assert int32_bittest24 : ((−(27 : INT32)) asr 2 = −(7))


(* ----------------------- *)
(* int64                   *)
(* ----------------------- *)

val int64Lnot : INT64 → INT64
declare ocaml target_rep function int64Lnot = 'Int64.lognot'
```

declare *hol* target_rep function int64Lnot w = ('~' w)
declare *isabelle* target_rep function int64Lnot w = ('NOT' w)
declare *coq* target_rep function int64Lnot = 'TODO'

instance (*WordNot* INT_{64})
  let *lnot* = int64Lnot
end

val *int64Lor* : $\text{INT}_{64} \rightarrow \text{INT}_{64} \rightarrow \text{INT}_{64}$
declare *ocaml* target_rep function int64Lor = 'Int64.logor'
declare *hol* target_rep function int64Lor = 'word_or'
declare *isabelle* target_rep function int64Lor = infix 'OR'
declare *coq* target_rep function int64Lor = 'TODO'

instance (*WordOr* INT_{64})
  let *lor* = int64Lor
end

val *int64Lxor* : $\text{INT}_{64} \rightarrow \text{INT}_{64} \rightarrow \text{INT}_{64}$
declare *ocaml* target_rep function int64Lxor = 'Int64.logxor'
declare *hol* target_rep function int64Lxor = 'word_xor'
declare *isabelle* target_rep function int64Lxor = infix 'XOR'
declare *coq* target_rep function int64Lxor = 'TODO'

instance (*WordXor* INT_{64})
  let *lxor* = int64Lxor
end

val *int64Land* : $\text{INT}_{64} \rightarrow \text{INT}_{64} \rightarrow \text{INT}_{64}$
declare *ocaml* target_rep function int64Land = 'Int64.logand'
declare *hol* target_rep function int64Land = 'word_and'
declare *isabelle* target_rep function int64Land = infix 'AND'
declare *coq* target_rep function int64Land = 'TODO'

instance (*WordAnd* INT_{64})
  let *land* = int64Land
end

val *int64Lsl* : $\text{INT}_{64} \rightarrow \text{NAT} \rightarrow \text{INT}_{64}$
declare *ocaml* target_rep function int64Lsl = 'Int64.shift_left'
declare *hol* target_rep function int64Lsl = 'word_lsl'
declare *isabelle* target_rep function int64Lsl = infix '<<'
declare *coq* target_rep function int64Lsl = 'TODO'

instance (*WordLsl* INT_{64})
  let *lsl* = int64Lsl
end

val *int64Lsr* : $\text{INT}_{64} \rightarrow \text{NAT} \rightarrow \text{INT}_{64}$
declare *ocaml* target_rep function int64Lsr = 'Int64.shift_right_logical'
declare *hol* target_rep function int64Lsr = 'word_lsr'
declare *isabelle* target_rep function int64Lsr = infix '>>'
declare *coq* target_rep function int64Lsr = 'TODO'

instance (*WordLsr* INT_{64})
  let *lsr* = int64Lsr
end

val $int64Asr$: INT_{64} $\rightarrow$ NAT $\rightarrow$ INT_{64}
declare $ocaml$ target_rep function int64Asr = 'Int64.shift_right'
declare hol target_rep function int64Asr = 'word_asr'
declare $isabelle$ target_rep function int64Asr = infix '>>>'
declare coq target_rep function int64Asr = 'TODO'

instance ($WordAsr$ INT_{64})
let asr = int64Asr
end

assert $int64\_bittest_1$: $((6 : \text{INT}_{64}) \text{ land } 5 = 4)$
assert $int64\_bittest_2$: $((6 : \text{INT}_{64}) \text{ lor } 5 = 7)$
assert $int64\_bittest_3$: $((6 : \text{INT}_{64}) \text{ lxor } 5 = 3)$
assert $int64\_bittest_4$: $((12 : \text{INT}_{64}) \text{ land } 9 = 8)$
assert $int64\_bittest_5$: $((12 : \text{INT}_{64}) \text{ lor } 9 = 13)$
assert $int64\_bittest_6$: $((12 : \text{INT}_{64}) \text{ lxor } 9 = 5)$

assert $int64\_bittest_7$: $(\text{lnot } (12 : \text{INT}_{64}) = -13)$
assert $int64\_bittest_8$: $(\text{lnot } (27 : \text{INT}_{64}) = -28)$
assert $int64\_bittest_9$: $((27 : \text{INT}_{64}) \text{ lsl } 0 = 27)$
assert $int64\_bittest_{10}$: $((27 : \text{INT}_{64}) \text{ lsl } 1 = 54)$
assert $int64\_bittest_{11}$: $((27 : \text{INT}_{64}) \text{ lsl } 2 = 108)$
assert $int64\_bittest_{12}$: $((27 : \text{INT}_{64}) \text{ lsl } 3 = 216)$
assert $int64\_bittest_{13}$: $((27 : \text{INT}_{64}) \text{ lsr } 0 = 27)$
assert $int64\_bittest_{14}$: $((27 : \text{INT}_{64}) \text{ lsr } 1 = 13)$
assert $int64\_bittest_{15}$: $((27 : \text{INT}_{64}) \text{ lsr } 2 = 6)$
assert $int64\_bittest_{16}$: $((27 : \text{INT}_{64}) \text{ lsr } 3 = 3)$
assert $int64\_bittest_{17}$: $((27 : \text{INT}_{64}) \text{ asr } 0 = 27)$
assert $int64\_bittest_{18}$: $((27 : \text{INT}_{64}) \text{ asr } 1 = 13)$
assert $int64\_bittest_{19}$: $((27 : \text{INT}_{64}) \text{ asr } 2 = 6)$
assert $int64\_bittest_{20}$: $((27 : \text{INT}_{64}) \text{ asr } 3 = 3)$
assert $int64\_bittest_{21}$: $((-(27 : \text{INT}_{64})) \text{ lsr } 0 = -(27))$
assert $int64\_bittest_{22}$: $((-(27 : \text{INT}_{64}) \text{ asr } 0) = -(27))$
assert $int64\_bittest_{23}$: $((-(27 : \text{INT}_{64})) \text{ lsr } 34 = 1073741823)$
assert $int64\_bittest_{24}$: $((-(27 : \text{INT}_{64})) \text{ asr } 2 = -(7))$

```
(* ----------------------- *)
(* Words via bit sequences *)
(* ----------------------- *)
```

val $defaultLnot$: $\forall\, \alpha.$ (BITSEQUENCE $\rightarrow$ α) $\rightarrow$ (α $\rightarrow$ BITSEQUENCE) $\rightarrow$ α $\rightarrow$ α
let $defaultLnot$ $fromBitSeq$ $toBitSeq$ x = $fromBitSeq$ (bitSeqNegate ($toBitSeq$ x))

val $defaultLand$: $\forall\, \alpha.$ (BITSEQUENCE $\rightarrow$ α) $\rightarrow$ (α $\rightarrow$ BITSEQUENCE) $\rightarrow$ α $\rightarrow$ α $\rightarrow$ α
let $defaultLand$ $fromBitSeq$ $toBitSeq$ x_1 x_2 = $fromBitSeq$ (bitSeqAnd ($toBitSeq$ x_1) ($toBitSeq$ x_2))

val $defaultLor$: $\forall\, \alpha.$ (BITSEQUENCE $\rightarrow$ α) $\rightarrow$ (α $\rightarrow$ BITSEQUENCE) $\rightarrow$ α $\rightarrow$ α $\rightarrow$ α
let $defaultLor$ $fromBitSeq$ $toBitSeq$ x_1 x_2 = $fromBitSeq$ (bitSeqOr ($toBitSeq$ x_1) ($toBitSeq$ x_2))

val $defaultLxor$: $\forall\, \alpha.$ (BITSEQUENCE $\rightarrow$ α) $\rightarrow$ (α $\rightarrow$ BITSEQUENCE) $\rightarrow$ α $\rightarrow$ α $\rightarrow$ α
let $defaultLxor$ $fromBitSeq$ $toBitSeq$ x_1 x_2 = $fromBitSeq$ (bitSeqXor ($toBitSeq$ x_1) ($toBitSeq$ x_2))

val $defaultLsl$: $\forall\, \alpha.$ (BITSEQUENCE $\rightarrow$ α) $\rightarrow$ (α $\rightarrow$ BITSEQUENCE) $\rightarrow$ α $\rightarrow$ NAT $\rightarrow$ α
let $defaultLsl$ $fromBitSeq$ $toBitSeq$ x n = $fromBitSeq$ (bitSeqShiftLeft ($toBitSeq$ x) n)

val $defaultLsr$: $\forall\, \alpha.$ (BITSEQUENCE $\rightarrow$ α) $\rightarrow$ (α $\rightarrow$ BITSEQUENCE) $\rightarrow$ α $\rightarrow$ NAT $\rightarrow$ α

let *defaultLsr fromBitSeq toBitSeq x n* = *fromBitSeq* (bitSeqLogicalShiftRight (*toBitSeq x*) *n*)

val *defaultAsr* : $\forall\, \alpha.$ (BITSEQUENCE $\rightarrow$ α) $\rightarrow$ (α $\rightarrow$ BITSEQUENCE) $\rightarrow$ α $\rightarrow$ NAT $\rightarrow$ α
let *defaultAsr fromBitSeq toBitSeq x n* = *fromBitSeq* (bitSeqArithmeticShiftRight (*toBitSeq x*) *n*)

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

val *integerLnot* : $\mathbb{Z} \rightarrow \mathbb{Z}$
let *integerLnot* i = $-(i + 1)$

instance (*WordNot* $\mathbb{Z}$)
  let *lnot* = integerLnot
end

val *integerLor* : $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$
let *integerLor* i_1 i_2 = defaultLor integerFromBitSeq (bitSeqFromInteger Nothing) i_1 i_2
declare *ocaml* target_rep function integerLor = `'Big_int.or_big_int'`

instance (*WordOr* $\mathbb{Z}$)
  let *lor* = integerLor
end

val *integerLxor* : $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$
let *integerLxor* i_1 i_2 = defaultLxor integerFromBitSeq (bitSeqFromInteger Nothing) i_1 i_2
declare *ocaml* target_rep function integerLxor = `'Big_int.xor_big_int'`

instance (*WordXor* $\mathbb{Z}$)
  let *lxor* = integerLxor
end

val *integerLand* : $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$
let *integerLand* i_1 i_2 = defaultLand integerFromBitSeq (bitSeqFromInteger Nothing) i_1 i_2
declare *ocaml* target_rep function integerLand = `'Big_int.and_big_int'`

instance (*WordAnd* $\mathbb{Z}$)
  let *land* = integerLand
end

val *integerLsl* : $\mathbb{Z}$ $\rightarrow$ NAT $\rightarrow$ $\mathbb{Z}$
let *integerLsl* i n = defaultLsl integerFromBitSeq (bitSeqFromInteger Nothing) i n
declare *ocaml* target_rep function integerLsl = `'Big_int.shift_left_big_int'`

instance (*WordLsl* $\mathbb{Z}$)
  let *lsl* = integerLsl
end

val *integerAsr* : $\mathbb{Z}$ $\rightarrow$ NAT $\rightarrow$ $\mathbb{Z}$
let *integerAsr* i n = defaultAsr integerFromBitSeq (bitSeqFromInteger Nothing) i n
declare *ocaml* target_rep function integerAsr = `'Big_int.shift_right_big_int'`

instance (*WordLsr* $\mathbb{Z}$)
  let *lsr* = integerAsr
end

instance (*WordAsr* $\mathbb{Z}$)

let asr = integerAsr
end

assert $integer\_bittest_1$: $((6 : \mathbb{Z})$ land $5 = 4)$
assert $integer\_bittest_2$: $((6 : \mathbb{Z})$ lor $5 = 7)$
assert $integer\_bittest_3$: $((6 : \mathbb{Z})$ lxor $5 = 3)$
assert $integer\_bittest_4$: $((12 : \mathbb{Z})$ land $9 = 8)$
assert $integer\_bittest_5$: $((12 : \mathbb{Z})$ lor $9 = 13)$
assert $integer\_bittest_6$: $((12 : \mathbb{Z})$ lxor $9 = 5)$

assert $integer\_bittest_7$: (lnot $(12 : \mathbb{Z}) = -13)$
assert $integer\_bittest_8$: (lnot $(27 : \mathbb{Z}) = -28)$
assert $integer\_bittest_9$: $((27 : \mathbb{Z})$ lsl $0 = 27)$
assert $integer\_bittest_{10}$: $((27 : \mathbb{Z})$ lsl $1 = 54)$
assert $integer\_bittest_{11}$: $((27 : \mathbb{Z})$ lsl $2 = 108)$
assert $integer\_bittest_{12}$: $((27 : \mathbb{Z})$ lsl $3 = 216)$
assert $integer\_bittest_{13}$: $((27 : \mathbb{Z})$ lsr $0 = 27)$
assert $integer\_bittest_{14}$: $((27 : \mathbb{Z})$ lsr $1 = 13)$
assert $integer\_bittest_{15}$: $((27 : \mathbb{Z})$ lsr $2 = 6)$
assert $integer\_bittest_{16}$: $((27 : \mathbb{Z})$ lsr $3 = 3)$
assert $integer\_bittest_{17}$: $((27 : \mathbb{Z})$ asr $0 = 27)$
assert $integer\_bittest_{18}$: $((27 : \mathbb{Z})$ asr $1 = 13)$
assert $integer\_bittest_{19}$: $((27 : \mathbb{Z})$ asr $2 = 6)$
assert $integer\_bittest_{20}$: $((27 : \mathbb{Z})$ asr $3 = 3)$
assert $integer\_bittest_{22}$: $((-(27 : \mathbb{Z})$ asr $0) = -(27))$
assert $integer\_bittest_{24}$: $((-(27 : \mathbb{Z}))$ asr $2 = -(7))$

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

(* sometimes it is convenient to be able to perform bit-operations on ints.
   However, since int is not well-defined (it has different size on different systems),
   it should be used very carefully and only for operations that don't depend on the
   bitwidth of int *)
```

val $intFromBitSeq$: BITSEQUENCE $\rightarrow$ INT
let $intFromBitSeq\ bs$ = intFromInteger (integerFromBitSeq (resizeBitSeq (Just 31) bs))

val $bitSeqFromInt$: INT $\rightarrow$ BITSEQUENCE
let $bitSeqFromInt\ i$ = bitSeqFromInteger (Just 31) (integerFromInt i)

val $intLnot$: INT $\rightarrow$ INT
let $intLnot\ i$ = $-(i + 1)$
declare $ocaml$ target_rep function intLnot = `'lnot'`

instance ($WordNot$ INT)
let $lnot$ = intLnot
end

val $intLor$: INT $\rightarrow$ INT $\rightarrow$ INT
let $intLor\ i_1\ i_2$ = defaultLor intFromBitSeq bitSeqFromInt $i_1\ i_2$
declare $ocaml$ target_rep function intLor = infix `'lor'`

```
instance (WordOr INT)
  let lor = intLor
end

val intLxor : INT → INT → INT
let intLxor i1 i2 = defaultLxor intFromBitSeq bitSeqFromInt i1 i2
declare ocaml target_rep function intLxor = infix 'lxor'

instance (WordXor INT)
  let lxor = intLxor
end

val intLand : INT → INT → INT
let intLand i1 i2 = defaultLand intFromBitSeq bitSeqFromInt i1 i2
declare ocaml target_rep function intLand = infix 'land'

instance (WordAnd INT)
  let land = intLand
end

val intLsl : INT → NAT → INT
let intLsl i n = defaultLsl intFromBitSeq bitSeqFromInt i n
declare ocaml target_rep function intLsl = infix 'lsl'

instance (WordLsl INT)
  let lsl = intLsl
end

val intAsr : INT → NAT → INT
let intAsr i n = defaultAsr intFromBitSeq bitSeqFromInt i n
declare ocaml target_rep function intAsr = infix 'asr'

instance (WordAsr INT)
  let asr = intAsr
end



assert int_bittest1 : ((6 : INT) land 5 = 4)
assert int_bittest2 : ((6 : INT) lor 5 = 7)
assert int_bittest3 : ((6 : INT) lxor 5 = 3)
assert int_bittest4 : ((12 : INT) land 9 = 8)
assert int_bittest5 : ((12 : INT) lor 9 = 13)
assert int_bittest6 : ((12 : INT) lxor 9 = 5)

assert int_bittest7 : (lnot (12 : INT) = −13)
assert int_bittest8 : (lnot (27 : INT) = −28)
assert int_bittest9 : ((27 : INT) lsl 0 = 27)
assert int_bittest10 : ((27 : INT) lsl 1 = 54)
assert int_bittest11 : ((27 : INT) lsl 2 = 108)
assert int_bittest12 : ((27 : INT) lsl 3 = 216)
assert int_bittest17 : ((27 : INT) asr 0 = 27)
assert int_bittest18 : ((27 : INT) asr 1 = 13)
assert int_bittest19 : ((27 : INT) asr 2 = 6)
assert int_bittest20 : ((27 : INT) asr 3 = 3)
assert int_bittest22 : ((−(27 : INT) asr 0) = −(27))
assert int_bittest24 : ((−(27 : INT)) asr 2 = −(7))
```

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

(* some operations work also on positive numbers *)
```

val *naturalFromBitSeq* : BITSEQUENCE → ℕ
let *naturalFromBitSeq bs* = naturalFromInteger (integerFromBitSeq *bs*)

val *bitSeqFromNatural* : MAYBE NAT → ℕ → BITSEQUENCE
let *bitSeqFromNatural len n* = bitSeqFromInteger *len* (integerFromNatural *n*)

val *naturalLor* : ℕ → ℕ → ℕ
let *naturalLor* i_1 i_2 = defaultLor naturalFromBitSeq (bitSeqFromNatural Nothing) i_1 i_2
declare *ocaml* target_rep function naturalLor = `'Big_int.or_big_int'`

instance (*WordOr* ℕ)
  let *lor* = naturalLor
end

val *naturalLxor* : ℕ → ℕ → ℕ
let *naturalLxor* i_1 i_2 = defaultLxor naturalFromBitSeq (bitSeqFromNatural Nothing) i_1 i_2
declare *ocaml* target_rep function naturalLxor = `'Big_int.xor_big_int'`

instance (*WordXor* ℕ)
  let *lxor* = naturalLxor
end

val *naturalLand* : ℕ → ℕ → ℕ
let *naturalLand* i_1 i_2 = defaultLand naturalFromBitSeq (bitSeqFromNatural Nothing) i_1 i_2
declare *ocaml* target_rep function naturalLand = `'Big_int.and_big_int'`

instance (*WordAnd* ℕ)
  let *land* = naturalLand
end

val *naturalLsl* : ℕ → NAT → ℕ
let *naturalLsl i n* = defaultLsl naturalFromBitSeq (bitSeqFromNatural Nothing) *i n*
declare *ocaml* target_rep function naturalLsl = `'Big_int.shift_left_big_int'`

instance (*WordLsl* ℕ)
  let *lsl* = naturalLsl
end

val *naturalAsr* : ℕ → NAT → ℕ
let *naturalAsr i n* = defaultAsr naturalFromBitSeq (bitSeqFromNatural Nothing) *i n*
declare *ocaml* target_rep function naturalAsr = `'Big_int.shift_right_big_int'`

instance (*WordLsr* ℕ)
  let *lsr* = naturalAsr
end

instance (*WordAsr* ℕ)
  let *asr* = naturalAsr
end

assert $natural\_bittest_1$: $((6 : \mathbb{N})$ land $5 = 4)$
assert $natural\_bittest_2$: $((6 : \mathbb{N})$ lor $5 = 7)$
assert $natural\_bittest_3$: $((6 : \mathbb{N})$ lxor $5 = 3)$
assert $natural\_bittest_4$: $((12 : \mathbb{N})$ land $9 = 8)$
assert $natural\_bittest_5$: $((12 : \mathbb{N})$ lor $9 = 13)$
assert $natural\_bittest_6$: $((12 : \mathbb{N})$ lxor $9 = 5)$

assert $natural\_bittest_9$: $((27 : \mathbb{N})$ lsl $0 = 27)$
assert $natural\_bittest_{10}$: $((27 : \mathbb{N})$ lsl $1 = 54)$
assert $natural\_bittest_{11}$: $((27 : \mathbb{N})$ lsl $2 = 108)$
assert $natural\_bittest_{12}$: $((27 : \mathbb{N})$ lsl $3 = 216)$
assert $natural\_bittest_{13}$: $((27 : \mathbb{N})$ lsr $0 = 27)$
assert $natural\_bittest_{14}$: $((27 : \mathbb{N})$ lsr $1 = 13)$
assert $natural\_bittest_{15}$: $((27 : \mathbb{N})$ lsr $2 = 6)$
assert $natural\_bittest_{16}$: $((27 : \mathbb{N})$ lsr $3 = 3)$
assert $natural\_bittest_{17}$: $((27 : \mathbb{N})$ asr $0 = 27)$
assert $natural\_bittest_{18}$: $((27 : \mathbb{N})$ asr $1 = 13)$
assert $natural\_bittest_{19}$: $((27 : \mathbb{N})$ asr $2 = 6)$
assert $natural\_bittest_{20}$: $((27 : \mathbb{N})$ asr $3 = 3)$

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

(* sometimes it is convenient to be able to perform bit-operations on nats.
   However, since nat is not well-defined (it has different size on different systems),
   it should be used very carefully and only for operations that don't depend on the
   bitwidth of nat *)
```

val *natFromBitSeq* : BITSEQUENCE → NAT
let *natFromBitSeq* *bs* = natFromNatural (naturalFromBitSeq (resizeBitSeq (Just 31) *bs*))

val *bitSeqFromNat* : NAT → BITSEQUENCE
let *bitSeqFromNat* *i* = bitSeqFromNatural (Just 31) (naturalFromNat *i*)

val *natLor* : NAT → NAT → NAT
let *natLor* i_1 i_2 = defaultLor natFromBitSeq bitSeqFromNat i_1 i_2
declare *ocaml* target_rep function natLor = infix `'lor'`

instance (*WordOr* NAT)
  let *lor* = natLor
end

val *natLxor* : NAT → NAT → NAT
let *natLxor* i_1 i_2 = defaultLxor natFromBitSeq bitSeqFromNat i_1 i_2
declare *ocaml* target_rep function natLxor = infix `'lxor'`

instance (*WordXor* NAT)
  let *lxor* = natLxor
end

val *natLand* : NAT → NAT → NAT

```
let natLand i1 i2 = defaultLand natFromBitSeq bitSeqFromNat i1 i2
declare ocaml target_rep function natLand = infix 'land'

instance (WordAnd NAT)
  let land = natLand
end

val natLsl : NAT → NAT → NAT
let natLsl i n = defaultLsl natFromBitSeq bitSeqFromNat i n
declare ocaml target_rep function natLsl = infix 'lsl'

instance (WordLsl NAT)
  let lsl = natLsl
end

val natAsr : NAT → NAT → NAT
let natAsr i n = defaultAsr natFromBitSeq bitSeqFromNat i n
declare ocaml target_rep function natAsr = infix 'asr'

instance (WordAsr NAT)
  let asr = natAsr
end



assert nat_bittest1 : ((6 : NAT) land 5 = 4)
assert nat_bittest2 : ((6 : NAT) lor 5 = 7)
assert nat_bittest3 : ((6 : NAT) lxor 5 = 3)
assert nat_bittest4 : ((12 : NAT) land 9 = 8)
assert nat_bittest5 : ((12 : NAT) lor 9 = 13)
assert nat_bittest6 : ((12 : NAT) lxor 9 = 5)

assert nat_bittest9 : ((27 : NAT) lsl 0 = 27)
assert nat_bittest10 : ((27 : NAT) lsl 1 = 54)
assert nat_bittest11 : ((27 : NAT) lsl 2 = 108)
assert nat_bittest12 : ((27 : NAT) lsl 3 = 216)
assert nat_bittest17 : ((27 : NAT) asr 0 = 27)
assert nat_bittest18 : ((27 : NAT) asr 1 = 13)
assert nat_bittest19 : ((27 : NAT) asr 2 = 6)
assert nat_bittest20 : ((27 : NAT) asr 3 = 3)
```

20 Pervasives

declare {*isabelle*, *ocaml*, *hol*, *coq*} rename module = Lem_pervasives

include import *Basic_classes Bool Tuple Maybe Either Function Num Map Set List String Word*

import *Sorting Relation*

21 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 $\{hol,\ isabelle,\ ocaml,\ coq\}$ 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 $=$ "$choose is non-deterministic and only defined for non-empty sets. Its result may differ betwee$
$level representation of sets and be different for two representations of the same set.$"

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 $\sim\{coq\}$ $choose_0$: $\text{choose}\ \{(2 : \textsc{nat})\} = 2$
assert $\sim\{coq\}$ $choose_1$: $\text{choose}\ \{(5 : \textsc{nat})\} = 5$
assert $\sim\{coq\}$ $choose_2$: $\text{choose}\ \{(6 : \textsc{nat})\} = 6$
assert $\sim\{coq\}$ $choose_3$: $\text{choose}\ \{(6 : \textsc{nat}),\ 1,\ 2\} \in \{6,\ 1,\ 2\}$

```
(* ----------------------------*)
(* universal set               *)
(* --------------------------- *)
```

val *universal* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \textsc{set}\ \alpha$
declare compile_message universal $=$ "$universal sets are usually infinite and only available in HOL and Isabelle$"

let $\{hol,\ isabelle\}$ *universal* $=$ $\{\ x \mid \forall\ x \mid$ 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* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha$
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 $(\{\} \ :\ \text{SET NAT}) = []$
assert $toList_1$: toList $\{(6 : \text{NAT}),\ 1,\ 2\} \in \{[1;2;6],\ [1;6;2],\ [2;1;6],\ [2;6;1],\ [6;1;2],\ [6;2;1]\}$
assert $toList_2$: toList $(\{(2 : \text{NAT})\} \ :\ \text{SET NAT}) = [2]$

```
(* ----------------------------*)
(* 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* : $\forall\ \alpha.\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathbb{B})\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha$
declare *isabelle* target_rep function toOrderedListBy = 'ordered_list_of_set'

val *toOrderedList* : $\forall\ \alpha.\ SetType\ \alpha,\ Ord\ \alpha\ \Rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{LIST}\ \alpha$
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 $(\{\} \ :\ \text{SET NAT}) = []$
assert $toOrderedList_1$: toOrderedList $\{(6 : \text{NAT}),\ 1,\ 2\} = [1;2;6]$
assert $toOrderedList_2$: toOrderedList $(\{(2 : \text{NAT})\} \ :\ \text{SET NAT}) = [2]$

```
(* ----------------------------*)
(* unbounded fixed point       *)
(* --------------------------- *)

(* Is NOT supported by the coq backend! *)
```

val *leastFixedPointUnbounded* : $\forall\ \alpha.\ SetType\ \alpha\ \Rightarrow\ (\text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \alpha)\ \rightarrow\ \text{SET}\ \alpha\ \rightarrow\ \text{SET}\ \alpha$
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 $x\ \rightarrow\ x$)) $(\{\} \ :\ \text{SET NAT}) = \{\}$

assert $lfp\_saturate\_neg$: leastFixedPointUnbounded (map (fun $x \to -x$)) ($\{1,\ 2,\ 3\}$: SET INT) $= \{-3,\ -2,\ -1,\ 1,\ 2,\ 3\}$

assert $lfp\_saturate\_mod$: leastFixedPointUnbounded (map (fun $x \to (2{*}x) \bmod 5$)) ($\{1\}$: SET NAT) $= \{1,\ 2,\ 3,\ 4\}$

22 String_extra

```
(******************************************************************************)
(* String functions                                                           *)
(******************************************************************************)

open import Basic_classes
open import Num
open import List
open import String
open import List_extra
open import {hol} stringLib
open import {hol} ASCIInumbersTheory

declare {isabelle, ocaml, hol, coq} rename module = lem_string_extra


(******************************************************************************)
(* Character's to numbers                                                     *)
(******************************************************************************)

val ord : CHAR → NAT
declare hol target_rep function ord = 'ORD'
declare ocaml target_rep function ord = 'code'
(* TODO: The Isabelle and Coq representations are taken from a quick Google
   search, they might not be the best options *)
declare isabelle target_rep function ord = 'nat_of_char'
declare coq target_rep function ord = 'nat_of_ascii'

val chr : NAT → CHAR
declare hol target_rep function chr = 'CHR'
declare ocaml target_rep function chr = 'Char.chr'
(* TODO: The Isabelle and Coq representations are taken from a quick Google
   search, they might not be the best options *)
declare isabelle target_rep function chr = 'char_of_nat'
declare coq target_rep function chr = 'ascii_of_nat'

(******************************************************************************)
(* Converting to strings                                                      *)
(******************************************************************************)

class (Show α)
  val show : α → STRING
end

val natToStringHelper : NAT → LIST CHAR → LIST CHAR
let rec natToStringHelper n acc =
  if n = 0 then
    acc
  else
    natToStringHelper (n / 10) (chr (n mod 10 + 48) :: acc)

val natToString : NAT → STRING
let natToString n = toString (natToStringHelper n [])

declare hol target_rep function natToString = 'num_to_dec_string'

instance (Show NAT)
```

```
  let show  =  natToString
end

val naturalToStringHelper  :  ℕ  →  LIST CHAR  →  LIST CHAR
let rec naturalToStringHelper n acc  =
  if n = 0 then
    acc
  else
    naturalToStringHelper (n / 10) (chr (natFromNatural (n mod 10 + 48)) :: acc)

val naturalToString  :  ℕ  →  STRING
let naturalToString n  =  toString (naturalToStringHelper n [])

declare hol target_rep function naturalToString  =  'num_to_dec_string'

instance (Show ℕ)
  let show  =  naturalToString
end


(******************************************************************************)
(* List-like operations                                                       *)
(******************************************************************************)

val nth  :  STRING  →  NAT  →  CHAR
let nth s n  =  List_extra.nth (toCharList s) n

declare hol target_rep function nth l n  =  'SUB' (l,  n)
declare ocaml target_rep function nth  =  'String.get'

val stringConcat  :  LIST STRING  →  STRING
let stringConcat s  =
  List.foldr ↑ "" s

declare hol target_rep function stringConcat  =  'CONCAT'
declare ocaml target_rep function stringConcat s  =  'String.concat' "" s

(******************************************************************************)
(* String comparison                                                          *)
(******************************************************************************)

val stringCompare  :  STRING  →  STRING  →  ORDERING

(* TODO: *)
let inline stringCompare x y  =  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"

23 Pervasives_extra

declare {*isabelle*, *ocaml*, *hol*, *coq*} rename module = Lem_pervasives_extra

include import *Pervasives*
include import *Function_extra Maybe_extra Map_extra Set_extra Set_helpers List_extra String_extra*