2.8. interval.ak

//// In a eUTxO-based blockchain like Cardano, the management of time can be

//// finicky.

////

//// Indeed, in order to maintain a complete determinism in the execution of

//// scripts, it is impossible to introduce a notion of \_"current time"\_ since

//// the execution would then depend on factor that are external to the

//// transaction itself: the ineluctable stream of time flowing in our universe.

////

//// Hence, to work around that, we typically define time intervals, which gives

//// window -- a.k.a intervals -- within which the transaction can be executed.

//// From within a script, it isn't possible to know when exactly the script is

//// executed, but we can reason about the interval bounds to validate pieces of

//// logic.

/// A type to represent intervals of values. Interval are inhabited by a type

/// `a` which is useful for non-infinite intervals that have a finite

/// lower-bound and/or upper-bound.

///

/// This allows to represent all kind of mathematical intervals:

///

/// ```aiken

/// // [1; 10]

/// let i0: Interval<PosixTime> = Interval

/// { lower\_bound:

/// IntervalBound { bound\_type: Finite(1), is\_inclusive: True }

/// , upper\_bound:

/// IntervalBound { bound\_type: Finite(10), is\_inclusive: True }

/// }

/// ```

///

/// ```aiken

/// // (20; infinity)

/// let i1: Interval<PosixTime> = Interval

/// { lower\_bound:

/// IntervalBound { bound\_type: Finite(20), is\_inclusive: False }

/// , upper\_bound:

/// IntervalBound { bound\_type: PositiveInfinity, is\_inclusive: False }

/// }

/// ```

pub type Interval<a> {

lower\_bound: IntervalBound<a>,

upper\_bound: IntervalBound<a>,

}

/// An interval bound, either inclusive or exclusive.

pub type IntervalBound<a> {

bound\_type: IntervalBoundType<a>,

is\_inclusive: Bool,

}

/// Return the highest bound of the two.

///

/// ```aiken

/// let ib1 = IntervalBound { bound\_type: Finite(0), is\_inclusive: False }

/// let ib2 = IntervalBound { bound\_type: Finite(1), is\_inclusive: False }

///

/// interval.max(ib1, ib2) == ib2

/// ```

pub fn max(

left: IntervalBound<Int>,

right: IntervalBound<Int>,

) -> IntervalBound<Int> {

when compare\_bound(left, right) is {

Less -> right

Equal -> left

Greater -> left

}

}

/// Return the smallest bound of the two.

///

/// ```aiken

/// let ib1 = IntervalBound { bound\_type: Finite(0), is\_inclusive: False }

/// let ib2 = IntervalBound { bound\_type: Finite(1), is\_inclusive: False }

///

/// interval.min(ib1, ib2) == ib1

/// ```

pub fn min(

left: IntervalBound<Int>,

right: IntervalBound<Int>,

) -> IntervalBound<Int> {

when compare\_bound(left, right) is {

Less -> left

Equal -> left

Greater -> right

}

}

fn compare\_bound(

left: IntervalBound<Int>,

right: IntervalBound<Int>,

) -> Ordering {

when compare\_bound\_type(left.bound\_type, right.bound\_type) is {

Less -> Less

Greater -> Greater

Equal ->

if left.is\_inclusive == right.is\_inclusive {

Equal

} else if left.is\_inclusive {

Greater

} else {

Less

}

}

}

/// A type of interval bound. Where finite, a value of type `a` must be

/// provided. `a` will typically be an `Int`, representing a number of seconds or

/// milliseconds.

pub type IntervalBoundType<a> {

NegativeInfinity

Finite(a)

PositiveInfinity

}

fn compare\_bound\_type(

left: IntervalBoundType<Int>,

right: IntervalBoundType<Int>,

) -> Ordering {

when left is {

NegativeInfinity ->

when right is {

NegativeInfinity -> Equal

\_ -> Less

}

PositiveInfinity ->

when right is {

PositiveInfinity -> Equal

\_ -> Greater

}

Finite(left) ->

when right is {

NegativeInfinity -> Greater

PositiveInfinity -> Less

Finite(right) ->

if left < right {

Less

} else if left == right {

Equal

} else {

Greater

}

}

}

}

// TODO: Replace 'Int' with a generic 'a' once we have comparable traits.

/// Checks whether an element is contained within the interval.

///

/// ```aiken

/// let iv =

/// Interval {

/// lower\_bound: IntervalBound {

/// bound\_type: Finite(14),

/// is\_inclusive: True

/// },

/// upper\_bound: IntervalBound {

/// bound\_type: Finite(42),

/// is\_inclusive: False

/// },

/// }

///

/// interval.contains(iv, 25) == True

/// interval.contains(iv, 0) == False

/// interval.contains(iv, 14) == True

/// interval.contains(iv, 42) == False

/// ```

pub fn contains(self: Interval<Int>, elem: Int) -> Bool {

let is\_greater\_than\_lower\_bound =

when self.lower\_bound.bound\_type is {

NegativeInfinity -> True

Finite(lower\_bound) ->

if self.lower\_bound.is\_inclusive {

elem >= lower\_bound

} else {

elem > lower\_bound

}

PositiveInfinity -> False

}

let is\_smaller\_than\_upper\_bound =

when self.upper\_bound.bound\_type is {

NegativeInfinity -> False

Finite(upper\_bound) ->

if self.upper\_bound.is\_inclusive {

elem <= upper\_bound

} else {

elem < upper\_bound

}

PositiveInfinity -> True

}

is\_greater\_than\_lower\_bound && is\_smaller\_than\_upper\_bound

}

test contains\_1() {

let iv = everything()

contains(iv, 14)

}

test contains\_2() {

let iv = entirely\_before(15)

contains(iv, 14)

}

test contains\_3() {

let iv = before(14)

contains(iv, 14)

}

test contains\_4() {

let iv = entirely\_before(14)

!contains(iv, 14)

}

test contains\_5() {

let iv = entirely\_after(13)

contains(iv, 14)

}

test contains\_6() {

let iv = after(14)

contains(iv, 14)

}

test contains\_7() {

let iv = entirely\_after(14)

!contains(iv, 14)

}

test contains\_8() {

let iv = between(42, 1337)

!contains(iv, 14)

}

test contains\_9() {

let iv = between(0, 42)

contains(iv, 14)

}

test contains\_10() {

let iv = between(0, 42)

contains(iv, 42)

}

test contains\_11() {

let iv = entirely\_between(0, 42)

!contains(iv, 0)

}

test contains\_12() {

let iv = empty()

!contains(iv, 14)

}

/// Create an interval that contains every possible values. i.e. (-INF, +INF)

///

/// ```aiken

/// interval.contains(everything(), 0) == True

/// interval.contains(everything(), 1000) == True

/// ```

pub fn everything() -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: NegativeInfinity,

is\_inclusive: True,

},

upper\_bound: IntervalBound {

bound\_type: PositiveInfinity,

is\_inclusive: True,

},

}

}

/// Create an empty interval that contains no value.

///

/// ```aiken

/// interval.contains(empty(), 0) == False

/// interval.contains(empty(), 1000) == False

/// ```

pub fn empty() -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: PositiveInfinity,

is\_inclusive: True,

},

upper\_bound: IntervalBound {

bound\_type: NegativeInfinity,

is\_inclusive: True,

},

}

}

/// Create an interval that includes all values between two bounds, including the bounds. i.e. [lower\_bound, upper\_bound]

///

/// ```aiken

/// interval.between(10, 100) == Interval {

/// lower\_bound: IntervalBound { bound\_type: Finite(10), is\_inclusive: True },

/// upper\_bound: IntervalBound { bound\_type: Finite(100), is\_inclusive: True },

/// }

/// ```

pub fn between(lower\_bound: a, upper\_bound: a) -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: Finite(lower\_bound),

is\_inclusive: True,

},

upper\_bound: IntervalBound {

bound\_type: Finite(upper\_bound),

is\_inclusive: True,

},

}

}

/// Create an interval that includes all values between two bounds, excluding the bounds. i.e. (lower\_bound, upper\_bound)

///

/// ```aiken

/// interval.entirely\_between(10, 100) == Interval {

/// lower\_bound: IntervalBound { bound\_type: Finite(10), is\_inclusive: False },

/// upper\_bound: IntervalBound { bound\_type: Finite(100), is\_inclusive: False },

/// }

/// ```

pub fn entirely\_between(lower\_bound: a, upper\_bound: a) -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: Finite(lower\_bound),

is\_inclusive: False,

},

upper\_bound: IntervalBound {

bound\_type: Finite(upper\_bound),

is\_inclusive: False,

},

}

}

/// Create an interval that includes all values greater than the given bound. i.e [lower\_bound, +INF)

///

/// ```aiken

/// interval.after(10) == Interval {

/// lower\_bound: IntervalBound { bound\_type: Finite(10), is\_inclusive: True },

/// upper\_bound: IntervalBound { bound\_type: PositiveInfinity, is\_inclusive: True },

/// }

/// ```

pub fn after(lower\_bound: a) -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: Finite(lower\_bound),

is\_inclusive: True,

},

upper\_bound: IntervalBound {

bound\_type: PositiveInfinity,

is\_inclusive: True,

},

}

}

// TODO: Replace 'Int' with a generic 'a' once we have comparable traits.

/// Check whether the interval is entirely after the point "a"

///

/// ```aiken

/// interval.is\_entirely\_after(interval.after(10), 5) == True

/// interval.is\_entirely\_after(interval.after(10), 10) == False

/// interval.is\_entirely\_after(interval.after(10), 15) == False

/// interval.is\_entirely\_after(interval.between(10, 20), 30) == False

/// interval.is\_entirely\_after(interval.between(10, 20), 5) == True

pub fn is\_entirely\_after(self: Interval<Int>, point: Int) -> Bool {

when self.lower\_bound.bound\_type is {

Finite(low) ->

if self.lower\_bound.is\_inclusive {

point < low

} else {

point <= low

}

\_ -> False

}

}

test is\_entirely\_after\_1() {

is\_entirely\_after(after(10), 5)

}

test is\_entirely\_after\_2() {

!is\_entirely\_after(after(10), 10)

}

test is\_entirely\_after\_3() {

!is\_entirely\_after(after(10), 15)

}

test is\_entirely\_after\_4() {

!is\_entirely\_after(between(10, 20), 30)

}

test is\_entirely\_after\_5() {

is\_entirely\_after(between(10, 20), 5)

}

test is\_entirely\_after\_6() {

is\_entirely\_after(entirely\_after(10), 10)

}

test is\_entirely\_after\_7() {

!is\_entirely\_after(before(10), 5)

}

test is\_entirely\_after\_8() {

!is\_entirely\_after(before(10), 15)

}

test is\_entirely\_after\_9() {

!is\_entirely\_after(entirely\_before(10), 5)

}

// TODO: Replace 'Int' with a generic 'a' once we have comparable traits.

/// Check whether the interval is entirely before the point "a"

///

/// ```aiken

/// interval.is\_entirely\_before(interval.before(10), 15) == True

/// interval.is\_entirely\_before(interval.before(10), 10) == False

/// interval.is\_entirely\_before(interval.before(10), 5) == False

/// interval.is\_entirely\_before(interval.between(10, 20), 30) == True

/// interval.is\_entirely\_before(interval.between(10, 20), 5) == False

pub fn is\_entirely\_before(self: Interval<Int>, point: Int) -> Bool {

when self.upper\_bound.bound\_type is {

Finite(hi) ->

if self.upper\_bound.is\_inclusive {

hi < point

} else {

hi <= point

}

\_ -> False

}

}

test is\_entirely\_before\_1() {

is\_entirely\_before(before(10), 15)

}

test is\_entirely\_before\_2() {

!is\_entirely\_before(before(10), 10)

}

test is\_entirely\_before\_3() {

!is\_entirely\_before(before(10), 5)

}

test is\_entirely\_before\_4() {

is\_entirely\_before(between(10, 20), 30)

}

test is\_entirely\_before\_5() {

!is\_entirely\_before(between(10, 20), 5)

}

test is\_entirely\_before\_6() {

is\_entirely\_before(entirely\_before(10), 10)

}

test is\_entirely\_before\_7() {

!is\_entirely\_before(after(10), 15)

}

test is\_entirely\_before\_8() {

!is\_entirely\_before(after(10), 5)

}

test is\_entirely\_before\_9() {

!is\_entirely\_before(entirely\_after(10), 5)

}

/// Create an interval that includes all values after (and not including) the given bound. i.e (lower\_bound, +INF)

///

/// ```aiken

/// interval.entirely\_after(10) == Interval {

/// lower\_bound: IntervalBound { bound\_type: Finite(10), is\_inclusive: False },

/// upper\_bound: IntervalBound { bound\_type: PositiveInfinity, is\_inclusive: True },

/// }

/// ```

pub fn entirely\_after(lower\_bound: a) -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: Finite(lower\_bound),

is\_inclusive: False,

},

upper\_bound: IntervalBound {

bound\_type: PositiveInfinity,

is\_inclusive: True,

},

}

}

/// Create an interval that includes all values before (and including) the given bound. i.e (-INF, upper\_bound]

///

/// ```aiken

/// interval.before(100) == Interval {

/// lower\_bound: IntervalBound { bound\_type: NegativeInfinity, is\_inclusive: True },

/// upper\_bound: IntervalBound { bound\_type: Finite(100), is\_inclusive: True },

/// }

/// ```

pub fn before(upper\_bound: a) -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: NegativeInfinity,

is\_inclusive: True,

},

upper\_bound: IntervalBound {

bound\_type: Finite(upper\_bound),

is\_inclusive: True,

},

}

}

/// Create an interval that includes all values before (and not including) the given bound. i.e (-INF, upper\_bound)

///

/// ```aiken

/// interval.entirely\_before(10) == Interval {

/// lower\_bound: IntervalBound { bound\_type: NegativeInfinity, is\_inclusive: True },

/// upper\_bound: IntervalBound { bound\_type: Finite(10), is\_inclusive: False },

/// }

/// ```

pub fn entirely\_before(upper\_bound: a) -> Interval<a> {

Interval {

lower\_bound: IntervalBound {

bound\_type: NegativeInfinity,

is\_inclusive: True,

},

upper\_bound: IntervalBound {

bound\_type: Finite(upper\_bound),

is\_inclusive: False,

},

}

}

/// Tells whether an interval is empty; i.e. that is contains no value.

///

/// ```aiken

/// let iv1 = interval.empty()

///

/// let iv2 = Interval {

/// lower\_bound: IntervalBound { bound\_type: Finite(0), is\_inclusive: False },

/// upper\_bound: IntervalBound { bound\_type: Finite(0), is\_inclusive: False },

/// }

///

/// let iv3 = Interval {

/// lower\_bound: IntervalBound { bound\_type: Finite(0), is\_inclusive: False },

/// upper\_bound: IntervalBound { bound\_type: Finite(100), is\_inclusive: False },

/// }

///

/// interval.is\_empty(iv1) == True

/// interval.is\_empty(iv2) == True

/// interval.is\_empty(iv3) == False

///

/// // Note: Two empty intervals are not necessarily equal.

/// iv1 != iv2

/// ```

pub fn is\_empty(self: Interval<Int>) -> Bool {

let ordering =

compare\_bound\_type(self.lower\_bound.bound\_type, self.upper\_bound.bound\_type)

when ordering is {

Greater -> True

Equal -> !(self.lower\_bound.is\_inclusive && self.upper\_bound.is\_inclusive)

Less -> {

let is\_open\_interval =

!self.lower\_bound.is\_inclusive && !self.upper\_bound.is\_inclusive

if is\_open\_interval {

when (self.lower\_bound.bound\_type, self.upper\_bound.bound\_type) is {

(Finite(lower\_bound), Finite(upper\_bound)) ->

lower\_bound + 1 == upper\_bound

\_ -> False

}

} else {

False

}

}

}

}

/// Computes the largest interval contains in the two given intervals, if any.

///

/// ```aiken

/// let iv1 = interval.between(0, 10)

/// let iv2 = interval.between(2, 14)

/// interval.intersection(iv1, iv2) == interval.between(2, 10)

///

/// let iv1 = interval.entirely\_before(10)

/// let iv2 = interval.entirely\_after(0)

/// interval.intersection(iv1, iv2) == interval.entirely\_between(0, 10)

///

/// let iv1 = interval.between(0, 1)

/// let iv2 = interval.between(2, 3)

/// interval.intersection(iv1, iv2) |> interval.is\_empty

/// ```

pub fn intersection(iv1: Interval<Int>, iv2: Interval<Int>) -> Interval<Int> {

Interval {

lower\_bound: max(iv1.lower\_bound, iv2.lower\_bound),

upper\_bound: min(iv1.upper\_bound, iv2.upper\_bound),

}

}

test intersection\_1() {

let iv1 = between(0, 10)

let iv2 = between(2, 14)

intersection(iv1, iv2) == between(2, 10)

}

test intersection\_2() {

let iv1 = between(0, 1)

let iv2 = between(1, 2)

intersection(iv1, iv2) == between(1, 1)

}

test intersection\_3() {

let iv1 = between(0, 1)

let iv2 = entirely\_between(1, 2)

intersection(iv1, iv2)

|> is\_empty

}

test intersection\_4() {

let iv1 = entirely\_between(0, 1)

let iv2 = entirely\_between(1, 2)

intersection(iv1, iv2)

|> is\_empty

}

test intersection\_5() {

let iv1 = between(0, 10)

let iv2 = before(4)

intersection(iv1, iv2) == between(0, 4)

}

test intersection\_6() {

let iv1 = entirely\_before(10)

let iv2 = entirely\_after(0)

intersection(iv1, iv2) == entirely\_between(0, 10)

}

/// Computes the smallest interval containing the two given intervals, if any

///

/// ```aiken

/// let iv1 = between(0, 10)

/// let iv2 = between(2, 14)

/// hull(iv1, iv2) == between(0, 14)

///

/// let iv1 = between(5, 10)

/// let iv2 = before(0)

/// hull(iv1, iv2) == before(10)

///

/// let iv1 = entirely\_after(0)

/// let iv2 = between(10, 42)

/// hull(iv1, iv2) = entirely\_after(0)

/// ```

pub fn hull(iv1: Interval<Int>, iv2: Interval<Int>) -> Interval<Int> {

Interval {

lower\_bound: min(iv1.lower\_bound, iv2.lower\_bound),

upper\_bound: max(iv1.upper\_bound, iv2.upper\_bound),

}

}

test hull\_1() {

let iv1 = between(0, 10)

let iv2 = between(2, 14)

hull(iv1, iv2) == between(0, 14)

}

test hull\_2() {

let iv1 = between(5, 10)

let iv2 = before(0)

hull(iv1, iv2) == before(10)

}

test hull\_3() {

let iv1 = entirely\_after(0)

let iv2 = between(10, 42)

hull(iv1, iv2) == entirely\_after(0)

}