2.12. pairs.ak

//// A module for working with associative lists (a.k.a `Pairs`).

////

//// While any function that works on `List` also work on `Pairs`, this module provides some extra helpers

//// that are specifically tailored to working with associative lists. Fundamentally, a `Pairs<k, v>` is

//// a type-alias to `List<Pair<k, v>>`.

////

//// ### Important

////

//// Unlike dictionnaries (a.k.a. `Dict`), associative lists make no assumption

//// about the ordering of elements within the list. As a result, lookup

//// functions do traverse the entire list when invoked. They are also not \_sets\_,

//// and thus allow for duplicate keys. This is reflected in the functions used

//// to interact with them.

/// Remove a single key-value pair from the `Pairs`. If the key is not found, no changes are made.

/// Duplicate keys are not removed. Only the \*\*first\*\* key found is removed.

///

/// ```aiken

/// pairs.remove\_first([], "a") == []

/// pairs.remove\_first([Pair("a", 1)], "a") == []

/// pairs.remove\_first([Pair("a", 1), Pair("b", 2)], "a") == [Pair("b", 2)]

/// pairs.remove\_first([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == [Pair("b", 2), Pair("a", 3)]

/// ```

pub fn remove\_first(self: Pairs<key, value>, key k: key) -> Pairs<key, value> {

when self is {

[] ->

[]

[Pair(k2, v2), ..rest] ->

if k == k2 {

rest

} else {

[Pair(k2, v2), ..remove\_first(rest, k)]

}

}

}

test remove\_first\_1() {

remove\_first([], "a") == []

}

test remove\_first\_2() {

remove\_first([Pair("a", 14)], "a") == []

}

test remove\_first\_3() {

let fixture =

[Pair("a", 14)]

remove\_first(fixture, "b") == fixture

}

test remove\_first\_4() {

let fixture =

[Pair("a", 1), Pair("b", 2), Pair("a", 3)]

remove\_first(fixture, "a") == [Pair("b", 2), Pair("a", 3)]

}

/// Remove a single key-value pair from the Pairs. If the key is not found, no changes are made.

/// Duplicate keys are not removed. Only the \*\*last\*\* key found is removed.

///

/// ```aiken

/// pairs.remove\_last([], "a") == []

/// pairs.remove\_last([Pair("a", 1)], "a") == []

/// pairs.remove\_last([Pair("a", 1), Pair("b", 2)], "a") == [Pair("b", 2)]

/// pairs.remove\_last([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == [Pair("a", 1), Pair("b", 2)]

/// ```

pub fn remove\_last(self: Pairs<key, value>, key k: key) -> Pairs<key, value> {

when self is {

[] ->

[]

[Pair(k2, v2), ..rest] ->

if k == k2 {

let tail = remove\_last(rest, k)

if tail == rest {

rest

} else {

[Pair(k2, v2), ..tail]

}

} else {

[Pair(k2, v2), ..remove\_last(rest, k)]

}

}

}

test remove\_last\_1() {

remove\_last([], "a") == []

}

test remove\_last\_2() {

remove\_last([Pair("a", 14)], "a") == []

}

test remove\_last\_3() {

let fixture =

[Pair("a", 14)]

remove\_last(fixture, "b") == fixture

}

test remove\_last\_4() {

let fixture =

[Pair("a", 1), Pair("b", 2), Pair("a", 3)]

remove\_last(fixture, "a") == [Pair("a", 1), Pair("b", 2)]

}

/// Remove all key-value pairs matching the key from the Pairs. If the key is not found, no changes are made.

///

/// ```aiken

/// pairs.remove\_all([], "a") == []

/// pairs.remove\_all([Pair("a", 1)], "a") == []

/// pairs.remove\_all([Pair("a", 1), Pair("b", 2)], "a") == [Pair("b", 2)]

/// pairs.remove\_all([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == [Pair("b", 2)]

/// ```

pub fn remove\_all(self: Pairs<key, value>, key k: key) -> Pairs<key, value> {

when self is {

[] ->

[]

[Pair(k2, v2), ..rest] ->

if k == k2 {

remove\_all(rest, k)

} else {

[Pair(k2, v2), ..remove\_all(rest, k)]

}

}

}

test remove\_all\_1() {

remove\_all([], "a") == []

}

test remove\_all\_2() {

remove\_all([Pair("a", 14)], "a") == []

}

test remove\_all\_3() {

let fixture =

[Pair("a", 14)]

remove\_all(fixture, "b") == fixture

}

test remove\_all\_4() {

let fixture =

[Pair("a", 1), Pair("b", 2), Pair("a", 3)]

remove\_all(fixture, "a") == [Pair("b", 2)]

}

/// Finds the first key in the alist associated with a given value, if any.

///

/// ```aiken

/// pairs.find\_first([], 1) == None

/// pairs.find\_first([Pair("a", 1)], 1) == Some("a")

/// pairs.find\_first([Pair("a", 1), Pair("b", 2)], 1) == Some("a")

/// pairs.find\_first([Pair("a", 1), Pair("b", 2), Pair("c", 1)], 1) == Some("a")

/// ```

pub fn find\_first(self: Pairs<key, value>, v: value) -> Option<key> {

when self is {

[] -> None

[Pair(k2, v2), ..rest] ->

if v == v2 {

Some(k2)

} else {

find\_first(rest, v)

}

}

}

test find\_first\_1() {

find\_first([], "a") == None

}

test find\_first\_2() {

find\_first([Pair("a", 14)], 14) == Some("a")

}

test find\_first\_3() {

find\_first([Pair("a", 14)], 42) == None

}

test find\_first\_4() {

find\_first([Pair("a", 14), Pair("b", 42), Pair("c", 14)], 14) == Some("a")

}

/// Finds the last key in the alist associated with a given value, if any.

///

/// ```aiken

/// pairs.find\_last([], 1) == None

/// pairs.find\_last([Pair("a", 1)], 1) == Some("a")

/// pairs.find\_last([Pair("a", 1), Pair("b", 2)], 1) == Some("a")

/// pairs.find\_last([Pair("a", 1), Pair("b", 2), Pair("c", 1)], 1) == Some("c")

/// ```

pub fn find\_last(self: Pairs<key, value>, v: value) -> Option<key> {

when self is {

[] -> None

[Pair(k2, v2), ..rest] ->

if v == v2 {

when find\_last(rest, v) is {

None -> Some(k2)

some -> some

}

} else {

find\_last(rest, v)

}

}

}

test find\_last\_1() {

find\_last([], "a") == None

}

test find\_last\_2() {

find\_last([Pair("a", 14)], 14) == Some("a")

}

test find\_last\_3() {

find\_last([Pair("a", 14)], 42) == None

}

test find\_last\_4() {

find\_last([Pair("a", 14), Pair("b", 42), Pair("c", 14)], 14) == Some("c")

}

/// Finds all keys in the alist associated with a given value.

///

/// ```aiken

/// pairs.find\_all([], 1) == []

/// pairs.find\_all([Pair("a", 1)], 1) == ["a"]

/// pairs.find\_all([Pair("a", 1), Pair("b", 2)], 1) == ["a"]

/// pairs.find\_all([Pair("a", 1), Pair("b", 2), Pair("c", 1)], 1) == ["a", "c"]

/// ```

pub fn find\_all(self: Pairs<key, value>, v: value) -> List<key> {

when self is {

[] ->

[]

[Pair(k2, v2), ..rest] ->

if v == v2 {

[k2, ..find\_all(rest, v)]

} else {

find\_all(rest, v)

}

}

}

test find\_all\_1() {

find\_all([], "a") == []

}

test find\_all\_2() {

find\_all([Pair("a", 14)], 14) == ["a"]

}

test find\_all\_3() {

find\_all([Pair("a", 14)], 42) == []

}

test find\_all\_4() {

find\_all([Pair("a", 14), Pair("b", 42), Pair("c", 14)], 14) == ["a", "c"]

}

/// Fold over the key-value pairs in a Pairs. The fold direction follows the

/// order of elements in the Pairs and is done from right-to-left.

///

/// ```aiken

/// let fixture = [

/// Pair(1, 100),

/// Pair(2, 200),

/// Pair(3, 300),

/// ]

///

/// pairs.foldr(fixture, 0, fn(k, v, result) { k \* v + result }) == 1400

/// ```

pub fn foldr(

self: Pairs<key, value>,

zero: result,

with: fn(key, value, result) -> result,

) -> result {

when self is {

[] -> zero

[Pair(k, v), ..rest] -> with(k, v, foldr(rest, zero, with))

}

}

test foldr\_1() {

foldr([], 14, fn(\_, \_, \_) { 42 }) == 14

}

test foldr\_2() {

foldr(

[Pair("a", 42), Pair("b", 14)],

zero: 0,

with: fn(\_, v, total) { v + total },

) == 56

}

test foldr\_3() {

let fixture =

[Pair(1, 100), Pair(2, 200), Pair(3, 300)]

foldr(fixture, 0, fn(k, v, result) { k \* v + result }) == 1400

}

/// Fold over the key-value pairs in a pairs. The fold direction follows keys

/// in ascending order and is done from left-to-right.

///

/// ```aiken

/// let fixture = [

/// Pair(1, 100),

/// Pair(2, 200),

/// Pair(3, 300),

/// ]

///

/// pairs.foldl(fixture, 0, fn(k, v, result) { k \* v + result }) == 1400

/// ```

pub fn foldl(

self: Pairs<key, value>,

zero: result,

with: fn(key, value, result) -> result,

) -> result {

when self is {

[] -> zero

[Pair(k, v), ..rest] -> foldl(rest, with(k, v, zero), with)

}

}

test foldl\_1() {

foldl([], 14, fn(\_, \_, \_) { 42 }) == 14

}

test foldl\_2() {

foldl(

[Pair("a", 42), Pair("b", 14)],

zero: 0,

with: fn(\_, v, total) { v + total },

) == 56

}

/// Get the value in the alist by its key.

/// If multiple values with the same key exist, only the first one is returned.

///

/// ```aiken

/// pairs.get\_first([], "a") == None

/// pairs.get\_first([Pair("a", 1)], "a") == Some(1)

/// pairs.get\_first([Pair("a", 1), Pair("b", 2)], "a") == Some(1)

/// pairs.get\_first([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == Some(1)

/// ```

pub fn get\_first(self: Pairs<key, value>, key k: key) -> Option<value> {

when self is {

[] -> None

[Pair(k2, v), ..rest] ->

if k == k2 {

Some(v)

} else {

get\_first(rest, k)

}

}

}

test get\_first\_1() {

get\_first([], "a") == None

}

test get\_first\_2() {

get\_first([Pair("a", 1)], "a") == Some(1)

}

test get\_first\_3() {

get\_first([Pair("a", 1), Pair("b", 2)], "a") == Some(1)

}

test get\_first\_4() {

get\_first([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == Some(1)

}

test get\_first\_5() {

get\_first([Pair("a", 1), Pair("b", 2), Pair("c", 3)], "d") == None

}

/// Get the value in the alist by its key.

/// If multiple values with the same key exist, only the last one is returned.

///

/// ```aiken

/// pairs.get\_last([], "a") == None

/// pairs.get\_last([Pair("a", 1)], "a") == Some(1)

/// pairs.get\_last([Pair("a", 1), Pair("b", 2)], "a") == Some(1)

/// pairs.get\_last([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == Some(3)

/// ```

pub fn get\_last(self: Pairs<key, value>, key k: key) -> Option<value> {

when self is {

[] -> None

[Pair(k2, v), ..rest] ->

if k == k2 {

when get\_last(rest, k) is {

None -> Some(v)

some -> some

}

} else {

get\_last(rest, k)

}

}

}

test get\_last\_1() {

get\_last([], "a") == None

}

test get\_last\_2() {

get\_last([Pair("a", 1)], "a") == Some(1)

}

test get\_last\_3() {

get\_last([Pair("a", 1), Pair("b", 2)], "a") == Some(1)

}

test get\_last\_4() {

get\_last([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == Some(3)

}

test get\_last\_5() {

get\_last([Pair("a", 1), Pair("b", 2), Pair("c", 3)], "d") == None

}

/// Get all values in the alist associated with a given key.

///

/// ```aiken

/// pairs.get\_all([], "a") == []

/// pairs.get\_all([Pair("a", 1)], "a") == [1]

/// pairs.get\_all([Pair("a", 1), Pair("b", 2)], "a") == [1]

/// pairs.get\_all([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == [1, 3]

/// ```

pub fn get\_all(self: Pairs<key, value>, key k: key) -> List<value> {

when self is {

[] ->

[]

[Pair(k2, v), ..rest] ->

if k == k2 {

[v, ..get\_all(rest, k)]

} else {

get\_all(rest, k)

}

}

}

test get\_all\_1() {

get\_all([], "a") == []

}

test get\_all\_2() {

get\_all([Pair("a", 1)], "a") == [1]

}

test get\_all\_3() {

get\_all([Pair("a", 1), Pair("b", 2)], "a") == [1]

}

test get\_all\_4() {

get\_all([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == [1, 3]

}

test get\_all\_5() {

get\_all([Pair("a", 1), Pair("b", 2), Pair("c", 3)], "d") == []

}

/// Check if a key exists in the pairs.

///

/// ```aiken

/// pairs.has\_key([], "a") == False

/// pairs.has\_key([Pair("a", 1)], "a") == True

/// pairs.has\_key([Pair("a", 1), Pair("b", 2)], "a") == True

/// pairs.has\_key([Pair("a", 1), Pair("b", 2), Pair("a", 3)], "a") == True

/// ```

pub fn has\_key(self: Pairs<key, value>, k: key) -> Bool {

when self is {

[] -> False

// || is lazy so this is fine

[Pair(k2, \_), ..rest] -> k == k2 || has\_key(rest, k)

}

}

test has\_key\_1() {

!has\_key([], "a")

}

test has\_key\_2() {

has\_key([Pair("a", 14)], "a")

}

test has\_key\_3() {

!has\_key([Pair("a", 14)], "b")

}

test has\_key\_4() {

has\_key([Pair("a", 14), Pair("b", 42)], "b")

}

test has\_key\_5() {

has\_key([Pair("a", 14), Pair("b", 42), Pair("a", 42)], "a")

}

/// Extract all the keys present in a given `Pairs`.

///

/// ```aiken

/// pairs.keys([]) == []

/// pairs.keys([Pair("a", 1)]) == ["a"]

/// pairs.keys([Pair("a", 1), Pair("b", 2)]) == ["a", "b"]

/// pairs.keys([Pair("a", 1), Pair("b", 2), Pair("a", 3)]) == ["a", "b", "a"]

/// ```

pub fn keys(self: Pairs<key, value>) -> List<key> {

when self is {

[] ->

[]

[Pair(k, \_), ..rest] ->

[k, ..keys(rest)]

}

}

test keys\_1() {

keys([]) == []

}

test keys\_2() {

keys([Pair("a", 0)]) == ["a"]

}

test keys\_3() {

keys([Pair("a", 0), Pair("b", 0)]) == ["a", "b"]

}

/// Apply a function to all key-value pairs in a alist, replacing the values.

///

/// ```aiken

/// let fixture = [Pair("a", 100), Pair("b", 200)]

///

/// pairs.map(fixture, fn(\_k, v) { v \* 2 }) == [Pair("a", 200), Pair("b", 400)]

/// ```

pub fn map(

self: Pairs<key, value>,

with: fn(key, value) -> result,

) -> Pairs<key, result> {

when self is {

[] ->

[]

[Pair(k, v), ..rest] ->

[Pair(k, with(k, v)), ..map(rest, with)]

}

}

test map\_1() {

let fixture =

[Pair("a", 1), Pair("b", 2)]

map(fixture, with: fn(k, \_) { k }) == [Pair("a", "a"), Pair("b", "b")]

}

test map\_2() {

let fixture =

[Pair("a", 1), Pair("b", 2)]

map(fixture, with: fn(\_, v) { v + 1 }) == [Pair("a", 2), Pair("b", 3)]

}

/// Extract all the values present in a given `Pairs`.

///

/// ```aiken

/// pairs.values([]) == []

/// pairs.values([Pair("a", 1)]) == [1]

/// pairs.values([Pair("a", 1), Pair("b", 2)]) == [1, 2]

/// pairs.values([Pair("a", 1), Pair("b", 2), Pair("a", 3)]) == [1, 2, 3]

/// ```

pub fn values(self: Pairs<key, value>) -> List<value> {

when self is {

[] ->

[]

[Pair(\_, v), ..rest] ->

[v, ..values(rest)]

}

}

test values\_1() {

values([]) == []

}

test values\_2() {

values([Pair("a", 1)]) == [1]

}

test values\_3() {

values([Pair("a", 1), Pair("b", 2)]) == [1, 2]

}

test values\_4() {

values([Pair("a", 1), Pair("b", 2), Pair("a", 3)]) == [1, 2, 3]

}