Adatt Source Files

Sat Jun 5 18:44:00 2021

Contents

1 Files

1.1 Data/Collection/DistinctListSet.ad

Outline

Content

```
module Data.Collection.DistinctListSet (
  DistinctListSet(..)
) where

import qualified Data.List as List

datatype DistinctListSet 'a := DistinctListSet ['a]

instance Eq 'e => Semigroup (DistinctListSet 'e) where
  define (<>) (DistinctListSet l1) (DistinctListSet l2) :=
    DistinctListSet (l1 <> (List.filter (fn x -> not (List.elem x l1)) l2))
end-instance

instance Eq 'e => Monoid (DistinctListSet 'e) where
  define mempty := DistinctListSet []
end-instance

instance Eq 'e => Collection (DistinctListSet 'e) 'e where
  define insert e (DistinctListSet l) :=
    DistinctListSet (if elem e l then l else e:l)

  define fromList l := DistinctListSet (List.nub l)
  {-# inline fromList #-}

  define non-exec isEmptyP (DistinctListSet l) := bool2prop (null l)
  define non-exec memberP e (DistinctListSet l) := List.elemP e l
end-instance

instance Eq 'e => CollectionWithMembershipTest (DistinctListSet 'e) 'e where
  define occur e dl := if member e dl then 1 else 0
  define member e (DistinctListSet l):= elem e l
end-instance

instance Foldable (DistinctListSet 'e) 'e where
  define toList (DistinctListSet dl) := dl
  {-# inline toList #-}

  define foldl f acc (DistinctListSet dl):= List.foldl f acc dl
  define foldr f acc (DistinctListSet dl):= List.foldr f acc dl
  define size (DistinctListSet dl) := List.length dl
  define null (DistinctListSet dl) := List.null dl
  define elem e (DistinctListSet dl):= List.elem e dl
end-instance


instance Eq 'e => CollectionWithEmptynessTest (DistinctListSet 'e) 'e where
  define isEmpty := null
  {-#  inline isEmpty #-}
end-instance


instance Eq 'e => CollectionWithDelete (DistinctListSet 'e) 'e where
  define delete e (DistinctListSet l) := DistinctListSet (List.removeFirst ((==) e) l)

  define filter p (DistinctListSet l) := DistinctListSet (List.filter p l)

  define partition p (DistinctListSet l) :=
```

```
58      let
59        val (l1, l2) = List.partition p l
60      in
61        (DistinctListSet l1, DistinctListSet l2)
62      end
63
64    define difference (DistinctListSet l1) (DistinctListSet l2) :=
65      DistinctListSet (List.filter (fn x -> not (List.elem x l2)) l1)
66
67    define intersection (DistinctListSet l1) (DistinctListSet l2) :=
68      DistinctListSet (List.filter (fn x -> List.elem x l2) l1)
69
70  end-instance
71
72  instance Eq 'e => CollectionWithSubsetTest (DistinctListSet 'e) 'e where
73    define isSubsetOf (DistinctListSet l1) (DistinctListSet l2) :=
74      List.all (fn x -> List.elem x l2) l1
75
76    define isProperSubsetOf xs ys := isSubsetOf xs ys && (size xs < size ys)
77
78    define isSetEquiv xs ys := isSubsetOf xs ys && (size xs == size ys)
79
80    define disjoint (DistinctListSet l1) (DistinctListSet l2) :=
81      List.all (fn x -> not (List.elem x l2)) l1
82  end-instance
83
84  instance Eq 'e => FiniteCollection (DistinctListSet 'e) 'e where
85  end-instance
86
87  instance Eq 'e => SetLike (DistinctListSet 'e) 'e where
88  end-instance
89
90  instance Eq 'e => Set (DistinctListSet 'e) 'e where
91  end-instance
92
93  instance Eq 'e => FiniteSet (DistinctListSet 'e) 'e where
94  end-instance
95
96  end-module
```

1.2 Data/Collection/FunSetBool.ad

Outline

Content

```
module Data.Collection.FunSetBool (
  FunSetBool(..)
) where

-- A dataytpe of sets given by its characteristic function
datatype FunSetBool 'a := FunSetBool ('a -> Bool)

instance Eq 'e => Semigroup (FunSetBool 'e) where
  define (<>) (FunSetBool f1) (FunSetBool f2) :=
    FunSetBool (fn x -> f1 x || f2 x)
end-instance

instance Eq 'e => Monoid (FunSetBool 'e) where
  define mempty := FunSetBool (constant False)
end-instance

instance Eq 'e => Collection (FunSetBool 'e) 'e where
  define insert e (FunSetBool f) :=
    FunSetBool (fn x -> (x == e) || f x)

  define fromList l := FunSetBool (fn x -> elem x l)
  {-# inline fromList #-}

  define non-exec isEmptyP (FunSetBool f) :=
    (forallP e. bool2prop (not (f e)))
  define non-exec memberP e (FunSetBool f) := bool2prop (f e)
end-instance

instance Eq 'e => CollectionWithMembershipTest (FunSetBool 'e) 'e where
  define occur e dl := if member e dl then 1 else 0
  define member e (FunSetBool f):= f e
end-instance

instance Eq 'e => CollectionWithDelete (FunSetBool 'e) 'e where
  define delete e (FunSetBool f) := FunSetBool (fn x -> (x /= e) && f e)

  define filter p (FunSetBool f) := FunSetBool (fn x -> (p x) && (f x))

  define partition p s := (filter p s, filter (not . p) s)

  define difference (FunSetBool f1) (FunSetBool f2) :=
    FunSetBool (fn x -> f1 x && not (f2 x))

```

```
44    define intersection (FunSetBool f1) (FunSetBool f2) :=
45      FunSetBool (fn x -> (f1 x) && (f2 x))
46
47  end-instance
48
49  instance Eq 'e => SetLike (FunSetBool 'e) 'e where
50  end-instance
51
52  end-module
```

1.3 Data/Collection/ListBag.ad

Outline

Content

```
1  module Data.Collection.ListBag (
2    ListBag
3  ) where
4
5  import qualified Data.List as List
6
7
8  datatype ListBag 'a := ListBag ['a]
9
10 declare emptyListBag :: ListBag 'a
11 define emptyListBag := ListBag []
12
13 instance Functor ListBag where
14   define fmap f (ListBag l) := ListBag (map f l)
15 end-instance
16
17 instance Semigroup (ListBag 'a) where
18   define (<>) (ListBag xs) (ListBag ys) := ListBag (xs ++ ys)
19 end-instance
20
21 instance Monoid (ListBag 'a) where
22   define mempty := emptyListBag
23   {-# inline mempty #-}
24 end-instance
25
26 instance CollectionWithEmptynessTest (ListBag 'e) 'e where
27   define isEmpty (ListBag l) := (null l)
28 end-instance
29
30 instance Collection (ListBag 'e) 'e where
31   define insert e (ListBag l):= ListBag (e:l)
32   {-# inline insert #-}
33
34   define fromList l := ListBag l
35   {-# inline fromList #-}
36
37   define non-exec isEmptyP (ListBag l) := bool2prop (null l)
38   define non-exec memberP e (ListBag l) := List.elemP e l
39   {-# inline memberP #-}
40
41 end-instance
42
43 instance Foldable (ListBag 'e) 'e where
```

```
  define toList (ListBag l) := l
end-instance

instance CollectionWithMembershipTest (ListBag 'e) 'e where
  define occur e (ListBag l) := count e l
  {-# inline occur #-}

  define member := elem
  {-# inline member #-}

end-instance

instance Eq 'e => FiniteCollection (ListBag 'e) 'e where
end-instance



declare listIntersection :: Eq 'a => ['a] -> ['a] -> ['a]
define rec listIntersection [] _ := []
         | listIntersection _ [] := []
         | listIntersection (x:xs) l :=
             case (takeFirst ((==) x) l) of
               Nothing -> listIntersection xs l
             | Just (l', _) -> x:(listIntersection xs l')
             end

instance Eq 'e => CollectionWithDelete (ListBag 'e) 'e where
  define delete e (ListBag l) := ListBag (removeFirst ((==) e) l)
  {-# inline delete #-}

  define filter p (ListBag l) := ListBag (filter p l)
  {-# inline filter #-}

  define partition p (ListBag l) :=
    let
      val (lt, lf) = List.partition p l
    in
      (ListBag lt, ListBag lf)
    end
  {-# inline partition #-}

  define difference := foldl (fn l e -> delete e l)
  {-# inline difference #-}

  define intersection (ListBag l1) (ListBag l2) := ListBag (listIntersection l1 l2)
  {-# inline intersection #-}
end-instance


declare listSubsetOf :: Eq 'a => ['a] -> ['a] -> Bool
define rec listSubsetOf [] _ := True
         | listSubsetOf _ [] := False
         | listSubsetOf (x:xs) l :=
             case (takeFirst ((==) x) l) of
               Nothing -> False
             | Just (l', _) -> listSubsetOf xs l'
             end


```

```
declare listSetEquiv :: Eq 'a => ['a] -> ['a] -> Bool
define rec listSetEquiv [] [] := True
         | listSetEquiv [] _ := False
         | listSetEquiv _ [] := False
         | listSetEquiv (x:xs) l :=
             case (takeFirst ((==) x) l) of
               Nothing -> False
             | Just (l', _) -> listSetEquiv xs l'
             end

declare listDisjoint :: Eq 'a => ['a] -> ['a] -> Bool
define rec listDisjoint [] _ := True
         | listDisjoint _ [] := True
         | listDisjoint (x:xs) l :=
             case (takeFirst ((==) x) l) of
               Nothing -> listDisjoint xs l
             | Just _ -> False
             end


instance Eq 'e => CollectionWithSubsetTest (ListBag 'e) 'e where
  define isSubsetOf (ListBag xs) (ListBag ys) := listSubsetOf xs ys
  {-# inline isSubsetOf #-}

  define isProperSubsetOf (ListBag xs) (ListBag ys) := listSubsetOf xs ys && (length xs < length ys)
  {-# inline isProperSubsetOf #-}

  define isSetEquiv (ListBag xs) (ListBag ys) := listSetEquiv xs ys
  {-# inline isSetEquiv #-}

  define disjoint (ListBag xs) (ListBag ys) := listDisjoint xs ys
  {-# inline disjoint #-}
end-instance

instance Eq 'e => EqP (ListBag 'e) where
  define non-exec (===) q1 q2 := bool2prop (isSetEquiv q1 q2)
end-instance

instance Eq 'e => Eq (ListBag 'e) where
  define (==) := isSetEquiv
  {-# inline (==) #-}
end-instance

instance BagLike (ListBag 'e) 'e where
end-instance

instance Eq 'e => Bag (ListBag 'e) 'e where
end-instance

instance Eq 'e => FiniteBag (ListBag 'e) 'e where
end-instance

end-module
```

1.4 Data/Collection/Queue.ad

Outline

Content

```
1 module Data.Collection.Queue (
2   Queue,
3   emptyQueue,
4   enqueue,
5   dequeue
6 ) where
```

```
7
8  import qualified Data.List as List
9
10 datatype Queue 'a := Queue ['a] ['a]
11
12 declare emptyQueue :: Queue 'a
13 define emptyQueue := Queue [] []
14
15 {-# allow-similar-names "enqueue", "dequeue" #-}
16
17 declare enqueue :: 'a -> Queue 'a -> Queue 'a
18 define enqueue e (Queue l1 l2) := Queue l1 (e:l2)
19
20 declare dequeue :: Queue 'a -> Maybe ('a, Queue 'a)
21 define rec dequeue (Queue (e:l1) l2) := Just (e, Queue l1 l2)
22          | dequeue (Queue [] []) := Nothing
23          | dequeue (Queue [] l2) := dequeue (Queue (reverse l2) [])
24
25 declare queueToList :: Queue 'a -> ['a]
26 define queueToList (Queue l1 l2) := l1 ++ reverse l2
27
28 declare queueFromList :: ['a] -> Queue 'a
29 define queueFromList l := Queue l []
30
31 instance Functor Queue where
32   define fmap f (Queue l1 l2) := Queue (map f l1) (map f l2)
33 end-instance
34
35 instance EqP 'a => EqP (Queue 'a) where
36   define non-exec (===) q1 q2 := (queueToList q1) === (queueToList q2)
37 end-instance
38
39 instance Eq 'a => Eq (Queue 'a) where
40   define (==) q1 q2 := (queueToList q1) == (queueToList q2)
41 end-instance
42
43 instance Ord 'a => Ord (Queue 'a) where
44   define compare q1 q2 := compare (queueToList q1) (queueToList q2)
45 end-instance
46
47 instance Semigroup (Queue 'a) where
48   define (<>) q1 q2 := queueFromList ((queueToList q1) ++ (queueToList q2))
49 end-instance
50
51 instance Monoid (Queue 'a) where
52   define mempty := emptyQueue
53   {-# inline mempty #-}
54 end-instance
55
56 instance CollectionWithEmptynessTest (Queue 'e) 'e where
57   define isEmpty (Queue l1 l2) := (null l1 && null l2)
58 end-instance
59
60 instance Collection (Queue 'e) 'e where
61   define insert := enqueue
62   {-# inline insert #-}
63
64   define fromList := queueFromList
65   {-# inline fromList #-}
```

```
66
67    define non-exec isEmptyP q := bool2prop (isEmpty q)
68    define non-exec memberP e (Queue l1 l2) := List.elemP e l1 \/ List.elemP e l2
69  end-instance
70
71  instance Foldable (Queue 'e) 'e where
72    define toList := queueToList
73    {-# inline toList #-}
74
75    define size (Queue l1 l2) := length l1 + length l2
76    define null := isEmpty
77    {-# inline null #-}
78
79    define elem e (Queue l1 l2):= elem e l1 || elem e l2
80  end-instance
81
82  instance Eq 'e => CollectionWithMembershipTest (Queue 'e) 'e where
83    define occur e (Queue l1 l2) := count e l1 + count e l2
84    define member := elem
85    {-# inline member #-}
86  end-instance
87
88  instance Eq 'e => FiniteCollection (Queue 'e) 'e where
89  end-instance
90
91  end-module
```

1.5 Data/Collection/Stack.ad

Outline

Content

```
module Data.Collection.Stack (
  Stack,
  emptyStack,
  push,
  pop
) where

import qualified Data.List as List

datatype Stack 'a := Stack ['a]

declare emptyStack :: Stack 'a
define emptyStack := Stack []

declare push :: 'a -> Stack 'a -> Stack 'a
define push e (Stack l) := Stack (e:l)

declare pop :: Stack 'a -> Maybe ('a, Stack 'a)
define pop (Stack (e:l)) := Just (e, Stack l)
     | pop (Stack []) := Nothing

declare stackToList :: Stack 'a -> ['a]
define stackToList (Stack l) := l

declare stackFromList :: ['a] -> Stack 'a
define stackFromList l := Stack l

instance Functor Stack where
  define fmap f (Stack l) := Stack (map f l)
end-instance

instance EqP 'a => EqP (Stack 'a) where
  define non-exec (===) q1 q2 := (stackToList q1) === (stackToList q2)
end-instance

instance Eq 'a => Eq (Stack 'a) where
  define (==) q1 q2 := (stackToList q1) == (stackToList q2)
end-instance

instance Ord 'a => Ord (Stack 'a) where
  define compare q1 q2 := compare (stackToList q1) (stackToList q2)
end-instance
```

```
43
44 instance Semigroup (Stack 'a) where
45   define (<>) q1 q2 := stackFromList ((stackToList q1) ++ (stackToList q2))
46 end-instance
47
48 instance Monoid (Stack 'a) where
49   define mempty := emptyStack
50   {-# inline mempty #-}
51 end-instance
52
53 instance CollectionWithEmptynessTest (Stack 'e) 'e where
54   define isEmpty (Stack l) := (null l)
55 end-instance
56
57 instance Collection (Stack 'e) 'e where
58   define insert := push
59   {-# inline insert #-}
60
61   define fromList := stackFromList
62   {-# inline fromList #-}
63
64   define non-exec isEmptyP q := bool2prop (isEmpty q)
65   define non-exec memberP e (Stack l) := List.elemP e l
66 end-instance
67
68 instance Foldable (Stack 'e) 'e where
69   define toList := stackToList
70   {-# inline toList #-}
71 end-instance
72
73 instance Eq 'e => CollectionWithMembershipTest (Stack 'e) 'e where
74   define occur e (Stack l) := count e l
75   define member := elem
76   {-# inline member #-}
77 end-instance
78
79 instance Eq 'e => FiniteCollection (Stack 'e) 'e where
80 end-instance
81
82 end-module
```

1.6 Data/Lens.ad

Outline

Content

```
1  -- very poor man's lenses
2  module Data.Lens (
3    -- Basic Defintions for Lenses, Getters and Setters
4    Getter(..),
5    Setter(..),
6    Lens(..),
7    SimpleLens(..),
8    SimpleSetter(..),
9
10   IsGetter(..),
11   IsSetter(..),
12   to,
```

```
13    (^.),
14    simpleLensToGetter,
15
16    -- Tuple lenses
17    Elem1(..),
18    Elem2(..),
19    Elem3(..),
20    Elem4(..),
21    Elem5(..),
22    Elem6(..),
23    Elem7(..),
24    Elem8(..),
25    Elem9(..),
26
27    -- Templates for building lenses corresponding to record fields
28    BuildRecordFieldLens,
29    BuildRecordFieldLenses,
30    BuildRecordFieldLensesPrefix,
31    BuildRecordFieldLensesSuffix
32  ) where
33
34  {-# severity similar-names - #-}
35
36  -- | a very poor mans lens. Essentially it is a tuple of a get and set function. The getter-part can
37  -- get a certain part out of some bigger data. The setter sets this part to a new value. Setting might ch
38  -- output. For example, it is possible to set the second element of a tuple to a different type than its
39  -- This changes the type of the output so. ``Lens 's 't 'a 'b`` describes functions for getting a value o
40  -- type 's and a for updating a value of type 's with a value of type 'b to get a value of type 't.
41  recordtype Lens 's 't 'a 'b := Lens <|
42       lensGet :: 's -> 'a,
43       lensSet :: 's -> 'b -> 't
44    |>
45
46  -- | very often, the type is not changed by setting. This is then called a simple lens.
47  type SimpleLens 's 'a := Lens 's 's 'a 'a
48
49  -- | a datatype for just the getter part of lenses
50  recordtype Getter 'a 'b := Getter <|
51       getterGet :: 'a -> 'b
52    |>
53
54  -- | constructor for getters mimicking Haskell naming
55  declare to :: ('a -> 'b) -> Getter 'a 'b
56  define to := Getter
57  {-# inline to #-}
58
59  -- | the setter part of a lens, however, Setters without Getters are quite rare
60  recordtype Setter 's 't 'b := Setter <|
61       setterSet :: 's -> 'b -> 't
62    |>
63
64  type SimpleSetter 's 'b := Setter 's 's 'b
65
66
67  -- | type-class for accessing Lenses and Getters with a common interface
68  class IsGetter 'v 'a 'b | 'v -> 'a 'b where
69    declare view :: 'v -> 'a -> 'b
70
71    declare toGetter :: 'v -> Getter 'a 'b
```

```
72    define toGetter := Getter . view
73    {-# inline toGetter #-}
74  end-class
75
76  alias (^.) := view
77
78  -- | type-class for accessing Lenses and Setters with a common interface
79  class IsSetter 'v 's 't 'b | 'v -> 's 't 'b where
80    declare set :: 'v -> 's -> 'b -> 't
81
82    declare toSetter :: 'v -> Setter 's 't 'b
83    define toSetter := Setter . set
84    {-# inline toSetter #-}
85  end-class
86
87  instance IsGetter (Getter 'a 'b) 'a 'b where
88    define view := getterGet
89    {-# inline view #-}
90  end-instance
91
92  instance IsGetter (Lens 's 't 'a 'b) 's 'a where
93    define view := lensGet
94    {-# inline view #-}
95  end-instance
96
97  -- | a version of toGetter for Lenses with extra type constraints
98  declare simpleLensToGetter :: SimpleLens 'a 'b -> Getter 'a 'b
99  define simpleLensToGetter sl := toGetter sl
100 {-# inline simpleLensToGetter #-}
101
102 instance IsSetter (Setter 's 't 'b) 's 't 'b where
103   define set := setterSet
104   {-# inline set #-}
105 end-instance
106
107 instance IsSetter (Lens 's 't 'a 'b) 's 't 'b where
108   define set := lensSet
109   {-# inline set #-}
110 end-instance
111
112 -- ---------------------------------------
113 -- combine lenses, getters and setters
114 -- ---------------------------------------
115
116 declare combineGetter :: (IsGetter 'g1 'a 'b, IsGetter 'g2 'b 'c) => 'g1 -> 'g2 -> 'a -> 'c
117 define combineGetter g1 g2 := view g2 . view g1
118 {-# inline combineGetter #-}
119
120 instance Dot (Getter 'a 'b) (Getter 'b2 'c) (Getter 'a 'c) | 'b2 -> 'b  where
121   define (.) g1 g2 := Getter $ combineGetter g1 g2
122   {-# inline (.) #-}
123 end-instance
124
125 instance Dot (Getter 'a 'b) (Lens 'b2 'x 'c 'y) (Getter 'a 'c) | 'b2 -> 'b where
126   define (.) g1 g2 := Getter $ combineGetter g1 g2
127   {-# inline (.) #-}
128 end-instance
129
130 instance Dot (Lens 'a 'x 'b 'y) (Getter 'b2 'c) (Getter 'a 'c) | 'b2 -> 'b where
```

```
  define (.) g1 g2 := Getter $ combineGetter g1 g2
  {-# inline (.) #-}
end-instance


declare combineSetter :: (IsGetter 'g1 's1 'a, IsSetter 'g1 's1 't1 'b, IsSetter 'g2 'a 'b 'a2) => 'g1 ->
define combineSetter g1 g2 v1 x2 :=
  let val v2 = view g1 v1
      val v2' = set g2 v2 x2
      val v1' = set g1 v1 v2'
  in v1' end
{-# inline combineSetter #-}

instance Dot (Lens 's1 't1 'a1 'b1) (Setter 's2 't2 'b2) (Setter 's1 't1 'b2) | 'a1 -> 's2, 'b1 -> 't2 wh
  define (.) g1 g2 := Setter $ combineSetter g1 g2
  {-# inline (.) #-}
end-instance

instance Dot (Lens 's1 't1 'a1 'b1) (Lens 's2 't2 'a2 'b2) (Lens 's1 't1 'a2 'b2) | 'a1 -> 's2, 'b1 -> 't
  define (.) g1 g2 := Lens (combineGetter g1 g2) (combineSetter g1 g2)
  {-# inline (.) #-}
end-instance




-- ----------------------------------------
-- Lenses for tuples
-- ----------------------------------------

-- Tuples with distinct number of elements are different types in ADATT. Nevertheless
-- using the same syntax for accessing elements is useful. For this purpose type-classes
-- for accessing the i-th element of any tuple with at least i elements are introduced
-- and instantiated using code-generation


template BuildTupleElemLensClass (m :: Nat) := '''
class Elem{{=m=}} \'s \'t \'a \'b | \'s -> \'a, \'s \'b -> \'t where
  declare elem{{=m=}} :: Lens \'s \'t \'a \'b
end-class
'''

template BuildTupleElemLenses (n :: Nat) := '''
{{ name-scope {
     var tyVars := !GetTypeVarsNumbered("a", n);
     var xs := generateNames("x", n);
   }
}}
{{% for var m := 1 to n %}}
{{ var xs' := xs; xs'[m-1] := "_";
   var xs'' := xs; xs''[m-1] := "y";
   var tyVars' := tyVars; tyVars'[m-1] := "'y"; }}
instance Elem{{=m=}} {{= !BuildTuple(tyVars) =}} \'x \'z \'y | \'x -> {{= !BuildTuple(tyVars') =}}, \'z
  define elem{{=m=}} := Lens #{{= m =}} (fn {{= !BuildTuple(xs') =}} y -> {{= !BuildTuple(xs'') =}})
  \{-# inline elem{{=m=}} #-\}
end-instance

{{% end-for %}}
'''
```

```
190
191 generate-code '''
192   {{ -- max number of tuple elements for which to generate classes and instances
193      var max := 9  }}
194   {{% for var m := 1 to max %}}
195   {{! BuildTupleElemLensClass(m) !}}
196
197   {{% end-for %}}
198   {{% for var m := 2 to max %}}
199   {{! BuildTupleElemLenses(m) !}}
200   {{% end-for %}}
201 '''
202
203
204 -- ---------------------------------------
205 -- Lenses for tuples
206 -- ---------------------------------------
207
208 -- It is useful to have lenses corresponding to record fields.
209 -- There are templates to automatically generate such lenses.
210
211 -- - BuildRecordFieldLens (field, lensName)
212 -- generate a lens with name "lensName" for the record field field
213
214 -- - BuildRecordFieldLensesSuffix (suffix, type)
215 -- generate lenses for all fields of record-type "type", whose name ends with given suffix. The lens nam
216
217 -- - BuildRecordFieldLensesPrefix (prefix, type)
218 -- generate lenses for all fields of record-type "type", whose name starts with given prefix. The lens na
219
220 -- - BuildRecordFieldLenses (type)
221 -- short for "BuildRecordFieldLensesSuffix("_", type)
222
223
224 template BuildRecordFieldLens (field :: ConstID, lensName :: String ) := '''
225 {{ var tyID := field.baseType;
226    var (_, tyWithArgs) := !TypeWithArgs(True, tyID);
227    var fs := tyID.fields;
228 }}
229 declare {{= lensName =}} :: Lens {{= tyWithArgs =}} _ _ _
230 define {{= lensName =}} := <|
231     {{= const lensGet =}} := {{= field =}},
232     {{= const lensSet =}} := fn r v -> <|
233         {{% foreach var fi in fs.indices %}}
234         {{= fs[fi] =}} := {{% if (fs[fi] == field) %}}v{{% else %}}{{=fs[fi]=}} r{{% end-if %}}{{% if (fi
235         {{% end-foreach %}}
236       |>
237   |>
238 '''
239
240
241 template BuildRecordFieldLensesSuffix (suffix :: String, tyID :: TypeID) := '''
242 {{% foreach var f in tyID.fields %}}
243 {{% if (endsWith(suffix, f.name)) %}}
244 {{! BuildRecordFieldLens(f, dropEnd(suffix.length, f.name)) !}}
245
246 {{% end-if %}}
247 {{% end-foreach %}}'''
248
```

```
249 template BuildRecordFieldLensesPrefix (pre :: String, tyID :: TypeID) := '''
250 {{% foreach var f in tyID.fields %}}
251 {{% if (startsWith(pre, f.name)) %}}
252 {{! BuildRecordFieldLens(f, drop(pre.length, f.name)) !}}
253
254 {{% end-if %}}
255 {{% end-foreach %}}'''
256
257 template BuildRecordFieldLenses (tyID :: TypeID) := '''{{! BuildRecordFieldLensesSuffix("_", tyID) !}}'''
258
259 end-module
260
261
262
```

1.7 Data/List.ad

Outline

Content

```
1  module Data.List (
2    module Prelude.List,
3    replicate,
4    take,
5    drop,
6    splitAt,
7    takeWhile,
8    dropWhile,
9    break,
10   span,
11   lookup,
12   zip,
13   unzip
14 ) where
15
16 declare replicate :: Natural -> 'a -> ['a]
17 define rec replicate 0 _ := []
18           | replicate n a := a : (replicate (pred n) a)
```

```
19
20 declare take :: Natural -> ['a] -> ['a]
21 define rec take 0 _ := []
22          | take _ [] := []
23          | take n (x:xs) := x : (take (pred n) xs)
24
25 declare drop :: Natural -> ['a] -> ['a]
26 define rec drop 0 l := l
27          | drop _ [] := []
28          | drop n (_:xs) := drop (pred n) xs
29
30 declare splitAt :: Natural -> ['a] -> (['a], ['a])
31 define rec splitAt 0 l := ([], l)
32          | splitAt _ [] := ([], [])
33          | splitAt n (x:xs) :=
34              let
35                val (lt, ld) = (splitAt (pred n) xs)
36              in
37                (x:lt, ld)
38              end
39
40 property EqP 'a => splitAt_take_drop n (l :: ['a]) := ((splitAt n l) === (take n l, drop n l))
41
42
43 declare takeWhile :: ('a -> Bool) -> ['a] -> ['a]
44 define rec takeWhile _ [] := []
45          | takeWhile p (x:xs) :=
46              if (p x) then x : (takeWhile p xs) else []
47
48 declare dropWhile :: ('a -> Bool) -> ['a] -> ['a]
49 define rec dropWhile _ [] := []
50          | dropWhile p (x:xs) :=
51              if (p x) then (dropWhile p xs) else (x:xs)
52
53 declare span :: ('a -> Bool) -> ['a] -> (['a], ['a])
54 define rec span _ [] := ([], [])
55          | span p (x:xs) :=
56              if (p x) then
57                let val (ys, zs) = span p xs in
58                   (x:ys, zs)
59                end
60              else ([], x:xs)
61
62 declare break :: ('a -> Bool) -> ['a] -> (['a], ['a])
63 define break p := span (not . p)
64
65
66 declare lookup :: Eq 'k => 'k -> [('k, 'v)] -> Maybe 'v
67 define rec lookup _key [] := Nothing
68          | lookup key ((k, v):kvs) :=
69              if (key == k) then Just v else lookup key kvs
70
71
72 declare zipWith :: ('a -> 'b -> 'c) -> ['a] -> ['b] -> ['c]
73 define rec zipWith f (x:xs) (y:ys) := (f x y):(zipWith f xs ys)
74          | zipWith _ _ _ := undefined
75
76 declare zip :: ['a] -> ['b] -> [('a, 'b)]
77 define zip := zipWith (fn x y -> (x, y))
```

```
78
79 declare unzip :: [('a,'b)] -> (['a],['b])
80 define unzip :=  foldr (fn (x,y) (xs,ys) -> (x:xs,y:ys)) ([],[])
81
82 end-module
```

1.8 Data/Maybe.ad

Outline

Content

```
1  module Data.Maybe (
2    module Prelude.Maybe,
3    listToMaybe,
4    maybeToList,
5    catMaybes,
6    mapMaybe
7  ) where
8
9  declare listToMaybe :: ['a] -> Maybe 'a
10 define listToMaybe [] := Nothing
11      | listToMaybe (x:_) := Just x
12
13 declare maybeToList :: Maybe 'a -> ['a]
14 define maybeToList Nothing := []
15      | maybeToList (Just v) := [v]
16
17 declare catMaybes :: [Maybe 'a] -> ['a]
18 define rec catMaybes [] := []
19          | catMaybes (Nothing:vs) := catMaybes vs
20          | catMaybes ((Just v):vs) := v : (catMaybes vs)
21
22 declare mapMaybe :: ('a -> Maybe 'b) -> ['a] -> ['b]
23 define rec mapMaybe _ [] := []
24          | mapMaybe mf (v : vs) :=
25              case (mf v) of
26                  Nothing -> mapMaybe mf vs
27                | (Just x) -> x:(mapMaybe mf vs)
28              end
29
30 end-module
```

1.9 Prelude.ad

Outline

Content

```
module Prelude (
  module Prelude.Unit,
  module Prelude.Bool,
  module Prelude.Literal,
  module Prelude.Default,
  module Prelude.Function,
  module Prelude.Either,
  module Prelude.Maybe hiding (map, bind),
  module Prelude.Eq,
  module Prelude.List hiding (foldr, foldl, partition, elem, elemP, notElemP, null, filter, all, any, f
  module Prelude.Num,
  module Prelude.Ord,
  module Prelude.Tuple,
  module Prelude.Char,
  module Prelude.String,
  module Prelude.Text,
  module Prelude.BasicClasses,
  module Prelude.Foldable,
  module Prelude.Functor,
  module Prelude.Monad,
  module Prelude.Collection,
  module Prelude.Template
) where

{-# NoImplicitPrelude #-}

-- Just include instances, no need to explicitly export
import Prelude.ListCollection


end-module
```

1.10 Prelude/BasicClasses.ad

Outline

Content

```
1  module Prelude.BasicClasses (
2    Enum(..),
3    Bounded(..),
4    Finite(..),
5
6    Monoid(..),
7    mappend,
8    Semigroup(..)
9  ) where
10
11 {-# NoImplicitPrelude #-}
12
13 import Prelude.Bool
14 import Prelude.Function
15 import Prelude.Num
16 import Prelude.Eq
17 import Prelude.Ord
18 import Prelude.Maybe
19 import Prelude.Either
20 import Prelude.Template
21 import Prelude.Unit ()
22 import Prelude.List hiding (map)
23
24
25 class Ord 'a => Enum 'a where
26   declare succ :: 'a -> 'a
27   declare pred :: 'a -> 'a
```

```
28
29    declare fromEnum :: 'a -> Natural
30    declare toEnum :: Natural -> Maybe 'a
31
32    define succ a := fromMaybe a (toEnum (fromEnum a + 1))
33    define pred a := fromMaybe a (toEnum (fromEnum a - 1))
34
35    declare enumFromTo :: 'a -> 'a -> ['a]
36    define rec enumFromTo n m :=
37      if (n > m) then [] else n : (enumFromTo (succ n) m)
38
39    test toFromEnumInv (a :: 'a) :=
40      toEnum (fromEnum a) == Just a
41
42    test fromEnum11 a1 a2 :=
43           (fromEnum a1 == fromEnum a2) <-> (a1 == a2)
44
45    test toEnumBounds0 := isJust ((toEnum 0) :: Maybe 'a)
46    test toEnumBoundsSmaller i j :=
47      (i > j) -->
48      isJust ((toEnum i) :: Maybe 'a) -->
49      isJust ((toEnum j) :: Maybe 'a)
50
51    test compareOK a1 a2 :=
52      compare a1 a2 == compare (fromEnum a1) (fromEnum a2)
53
54    test succ_greater_ok (x :: 'a) := (succ x >= x)
55    test pred_smaller_ok (x :: 'a) := (pred x <= x)
56
57    test Bounded 'a => succ_bound_ok (x :: 'a) :=
58            (x < maxBound) --> ((succ x > x) && (pred (succ x) == x))
59    test Bounded 'a => pred_bound_ok (x :: 'a) :=
60            (x > minBound) --> ((pred x < x) && (succ (pred x) == x))
61    test Bounded 'a => pred_minbound := pred (minBound :: 'a) == minBound
62    test Bounded 'a => succ_maxbound := succ (maxBound :: 'a) == maxBound
63
64    {-# minimal fromEnum, toEnum #-}
65  end-class
66
67  instance Enum Natural where
68    define succ := (+) 1
69    define pred := (-) 1
70    define toEnum n := Just n
71    define fromEnum n := n
72  end-instance
73
74  {-# allow-similar-names "minBound", "maxBound",  "minBound_ok", "maxBound_ok", "minBoundEnum", "maxBoundE
75
76  class Ord 'a => Bounded 'a where
77    declare minBound :: 'a
78    declare maxBound :: 'a
79
80    test minBound_ok (x :: 'a) := (minBound :: 'a) <= x
81    test maxBound_ok (x :: 'a) := (maxBound :: 'a) >= x
82
83    test Enum 'a => minBoundEnum := fromEnum (minBound :: 'a) == 0
84
85    test Enum 'a => maxBoundEnum i := (i > fromEnum (maxBound :: 'a)) --> isNothing ((toEnum i) :: Maybe '
86  end-class
```

```
87
88
89  -- finite, type
90  class Finite 'a where
91    declare universeList :: ['a]
92
93    declare universeFoldWithAbort :: ('acc -> Bool) -> ('acc -> 'a -> 'acc) -> 'acc -> 'acc
94    {-# static resolve universeFoldWithAbort #-}
95
96    declare forallB_fun :: ('a -> Bool) -> Bool
97    declare existsB_fun :: ('a -> Bool) -> Bool
98    declare uexistsB_fun :: ('a -> Bool) -> Bool
99
100   define universeFoldWithAbort ab f acc := foldlWithAbort ab f acc universeList
101   define universeList := universeFoldWithAbort (constant False) (fn l e -> e:l) []
102
103   define forallB_fun p := universeFoldWithAbort not (constant p) True
104   define existsB_fun p := universeFoldWithAbort id (constant p) False
105   define uexistsB_fun p :=
106     (universeFoldWithAbort (fn c -> (c >= 2)) (fn c e -> if p e then c + 1 else c) 0) == (1 :: Natural)
107
108   {-# minimal universeList | universeFoldWithAbort #-}
109 end-class
110
111
112 class Semigroup 'a where
113   declare (<>) :: 'a -> 'a -> 'a
114 end-class
115
116
117 class Semigroup 'a => Monoid 'a where
118   declare mempty :: 'a
119
120   property EqP 'a => mempty_neutral_1 (x :: 'a) := (mempty <> x) === x
121   property EqP 'a => mempty_neutral_2 (x :: 'a) := (x <> mempty) === x
122 end-class
123
124 declare mappend :: Monoid 'a => 'a -> 'a -> 'a
125 define mappend := (<>)
126 {-# inline mappend #-}
127
128
129 instance Semigroup () where
130   define (<>) := constant (constant ())
131   {-# inline (<>) #-}
132 end-instance
133
134 instance Monoid () where
135   define mempty := ()
136   {-# inline mempty #-}
137 end-instance
138
139 instance Semigroup ['a] where
140   define (<>) := (++)
141   {-# inline (<>) #-}
142 end-instance
143
144 instance Monoid ['a] where
145   define mempty := []
```

```
  {-# inline mempty #-}
end-instance

instance Semigroup Ordering where
  define (<>) LT _ := LT
       | (<>) EQ o := o
       | (<>) GT _ := GT
end-instance

instance Monoid Ordering where
  define mempty := EQ
end-instance


instance (!Finite 'a, !Finite 'b) => Finite ('a, 'b) where
  define universeFoldWithAbort ab f :=
      universeFoldWithAbort ab (fn acc a -> universeFoldWithAbort ab (fn acc' b -> f acc' (a, b)) acc)
  {-# inline universeFoldWithAbort #-}
end-instance

template BuildFiniteTupleInstance (n :: Nat) := '''
{{ name-scope {
     var tyVars := !GetTypeVars(n);
     var xs := generateNames("x", n);
   }
   var super := [] :: [String];
   foreach var tyV in tyVars {
     super := insertAtEnd("!Finite " ++ tyV, super);
   }

   var inst := "Finite " ++ !BuildTuple(tyVars);
   var m := n / 2;
}}
{{! BuildInstanceHeader(super, inst) !}}
  define universeFoldWithAbort ab f :=
    universeFoldWithAbort ab (fn acc ({{= !BuildTuple(take(m, xs)) =}}, {{= !BuildTuple(drop(m, xs)) =}})
  \{-# INLINE universeFoldWithAbort #-\}
end-instance
'''

generate-code '''
  {{% for var i := 3 to 10 %}}
  {{! BuildFiniteTupleInstance(i) !}}

  {{% end-for %}}'''


instance Finite () where
  define universeList := [()]
  define universeFoldWithAbort ab f a :=
    if ab a then a else f a ()
end-instance

instance !Finite 'a => Finite (Maybe 'a) where
  define universeFoldWithAbort ab f a :=
    if ab a then a else
      universeFoldWithAbort ab (fn acc v -> f acc (Just v)) (f a Nothing)
end-instance

```

```
instance (!Finite 'a, !Finite 'b)  => Finite (Either 'a 'b) where
  define universeFoldWithAbort ab f a :=
    universeFoldWithAbort ab (fn acc v -> f acc (Left v)) $
      universeFoldWithAbort ab (fn acc v -> f acc (Right v)) a
end-instance



template DeriveInstanceEnumSimpleEnumDatatypeBody (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
{{ if (not (!IsSimpleEnumDatatype(ty))) {
      error("This template only works for simple enumeration types");
   }
   var constrs := ty.constructors;
   var maxInd := constrs.length - 1;
}}
define fromEnum {{
  for var pos := 0 to maxInd {
    if (pos > 0) print("     | fromEnum ");
    println(constrs[pos] ++ " := " ++ pos);
  }
}}
define toEnum {{
  for var pos := 0 to maxInd {
    if (pos > 0) print("     | toEnum ");
    println(pos ++ " := Just " ++ constrs[pos]);
  }
  println("     | toEnum _ := Nothing")
}}
define succ {{
  for var pos := 0 to maxInd {
    if (pos > 0) print("     | succ ");
    var nextPos := pos+1;
    if (nextPos > maxInd) nextPos := pos;
    println(constrs[pos] ++ " := " ++ constrs[nextPos]);
  }
}}
define pred {{
  for var pos := 0 to maxInd {
    if (pos > 0) print("     | pred ");
    println(constrs[pos] ++ " := " ++ constrs[pos-1]);
  }
}}
'''

register-derive-template Enum DeriveInstanceEnumSimpleEnumDatatypeBody

register-derive-template Enum DeriveInstanceEnumSimpleEnumDatatypeBody as "simpleEnum"

template DeriveInstanceFiniteSimpleEnumDatatypeBody (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
{{ if (not (!IsSimpleEnumDatatype(ty))) {
      error("This template only works for simple enumeration types");
   }
}}
define universeList := [{{= !Commafy(ty.constructors) =}}]
'''

register-derive-template Finite DeriveInstanceFiniteSimpleEnumDatatypeBody as "simpleEnum"



```

```
template DeriveInstanceBoundedSimpleEnumDatatypeBody (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
{{ if (not (!IsSimpleEnumDatatype(ty))) {
      error("This template only works for simple enumeration types");
   }
   var constrs := ty.constructors;
}}
define minBound := {{= constrs[0] =}}
define maxBound := {{= constrs[constrs.length - 1] =}}
'''

register-derive-template Bounded DeriveInstanceBoundedSimpleEnumDatatypeBody as "simpleEnum"


derive Ordering instances (Bounded, Finite, Enum) via "simpleEnum"
derive Bool instances (Bounded, Finite, Enum) via "simpleEnum"

end-module
```

1.11 Prelude/Bool.ad

Outline

Content

```
module Prelude.Bool (
  type Bool(..),
  boolCase,
  not, (&&), (||),
  (-->), (<->), (<-/->),

  type Prop,
  true, false,
  notP, (/\), (\/), (==>), (<=>), (<=/=>),
  bool2prop,

  forallP_fun,
  existsP_fun,
  uexistsP_fun
) where

{-# NoImplicitPrelude #-}

import Prelude.Default
{-# allow-similar-names "False", "True", "true", "false" #-}

datatype Bool := False | True
declare not :: Bool -> Bool

instance Default Bool where
end-instance

declare boolCase :: Bool -> 'a -> 'a -> 'a
define boolCase True ct _ := ct
     | boolCase False _ cf := cf

define not True := False
     | not False := True

declare (&&) :: Bool -> Bool -> Bool
define (&&) True b := b
     | (&&) False _ := False

declare (||) :: Bool -> Bool -> Bool
define (||) True _ := True
     | (||) False b := b

declare (-->) :: Bool -> Bool -> Bool
define (-->) b1 b2  := not b1 || b2

declare (<->) :: Bool -> Bool -> Bool
define (<->) b1 b2 :=
  (b1 --> b2) && (b2 --> b1)

declare (<-/->) :: Bool -> Bool -> Bool
define (<-/->) b1 b2 := not (b1 <-> b2)

type non-exec Prop

declare non-exec true  :: Prop
declare non-exec false :: Prop

```

```
58 declare non-exec bool2prop :: Bool -> Prop
59 define non-exec bool2prop b := if b then true else false
60
61
62 declare non-exec notP   :: Prop -> Prop
63 declare non-exec (/\)   :: Prop -> Prop -> Prop
64 declare non-exec (\/)   :: Prop -> Prop -> Prop
65 declare non-exec (==>)  :: Prop -> Prop -> Prop
66 declare non-exec (<=>)  :: Prop -> Prop -> Prop
67 declare non-exec (<=/=>)  :: Prop -> Prop -> Prop
68
69 {-# allow-similar-names "forallP_fun", "existsP_fun", "uexistsP_fun" #-}
70 declare non-exec forallP_fun :: ('a -> Prop) -> Prop
71 declare non-exec existsP_fun :: ('a -> Prop) -> Prop
72 declare non-exec uexistsP_fun :: ('a -> Prop) -> Prop
73
74 end-module
```

1.12 Prelude/Char.ad

Outline

Content

```
1 module Prelude.Char (
2   Char,
3   mkChar,
4   chr,
5   ord
6 ) where
7
8 {-# NoImplicitPrelude #-}
9
10 import Prelude.Eq
11 import Prelude.Bool
```

```
12 import Prelude.Ord
13 import Prelude.Num
14 import Prelude.Maybe
15 import Prelude.Literal
16 import Prelude.Default
17 import Prelude.BasicClasses
18
19 -- Characters are represented by their unicode-value
20 datatype Char := Char Natural
21
22 derive Char instances (Eq, EqP, Ord)
23
24 instance Default Char where
25   define default := '-'
26   {-# inline default #-}
27 end-instance
28
29 declare chr :: Natural -> Char
30 define chr := Char
31
32 declare ord :: Char -> Natural
33 define ord (Char c) := c
34
35 declare mkChar :: BuiltInChar -> Char
36
37 instance LiteralChar Char where
38   define mkLiteralChar := mkChar
39   {-# inline mkLiteralChar #-}
40 end-instance
41
42
43 instance Enum Char where
44   define succ (Char c) := Char (succ c)
45   define pred (Char c) := Char (pred c)
46   define fromEnum (Char c) := c
47   define toEnum c := Just (Char c)
48 end-instance
49
50 end-module
```

1.13 Prelude/Collection.ad

Outline

Content

```
1  module Prelude.Collection (
2    Collection(..),
3    empty,
4    union,
5    image,
6
7    setEmpty,
8    setInsert,
9    setFromList,
10
11   CollectionWithEmptynessTest(..),
12   CollectionWithMembershipTest(..),
13   CollectionWithDelete(..),
14   FiniteCollection(..),
15   CollectionWithSubsetTest(..),
16   SetLike(..),
17   BagLike(..),
18   Set(..),
19   FiniteSet(..),
20   Bag(..),
21   FiniteBag(..)
22 ) where
23
24 {-# NoImplicitPrelude #-}
25
26 import Prelude.Bool
27 import Prelude.BasicClasses
28 import Prelude.Eq
29 import Prelude.Ord
30 import Prelude.Function
31 import Prelude.Tuple
32 import Prelude.Functor
33 import Prelude.Foldable
34 import Prelude.Num
35 import qualified Prelude.List as List
36
37
38 -- {-# allow-similar-names "isEmptyP_ok", "isEmpty_ok" #-}
39
40 -- | A 'Collection' is a container for some elements. Not much is known
41 -- about this element or the container. There might be a finite or infinite number of elements, elements
42 -- might occour multiple times, their order might matter or not, there might not be a (decidable) equali
43 -- elements and membership might not be decidable.
44 -- Via the 'Monoid' type class an empty collection and combining of collections is available.
45 class Monoid 'c => Collection 'c 'e | 'c -> 'e where
46   declare insert :: 'e -> 'c -> 'c
```

```
  define insert e c := (singleton e) <> c

  declare fromList :: ['e] -> 'c
  define fromList := foldl (fn c e -> insert e c) mempty

  declare singleton :: 'e -> 'c
  define singleton e := insert e mempty

  declare non-exec memberP :: EqP 'e => 'e -> 'c -> Prop

  declare non-exec isEmptyP :: 'c -> Prop
  property EqP 'e => isEmptyP_ok (c :: 'c) := isEmptyP c <=> (forallP e. notP (memberP e c))
  property isEmptyP_empty := isEmptyP (mempty :: 'c)

  property EqP 'e => memberP_insert (e1 :: 'e) e2 (c :: 'c) := (memberP e1 (insert e2 c) <=> ((e1 === e2
  property EqP 'e => memberP_fromList (e :: 'e) (l :: ['e]) := (memberP e ((fromList l)::'c) <=> List.el
end-class

declare empty :: Collection 'c 'e => 'c
define empty := mempty
{-# inline empty #-}

declare union :: Collection 'c 'e => 'c -> 'c -> 'c
define union := (<>)
{-# inline union #-}

{-# severity class-constr-unused ignore #-}
declare image :: (!Functor 'C, Collection ('C 'e1) 'e1, Collection ('C 'e2) 'e2) => ('e1 -> 'e2) -> 'C '
define image := fmap
{-# inline image #-}
{-# severity class-constr-unused reset #-}

class Collection 'c 'e => CollectionWithEmptynessTest 'c 'e | 'c -> 'e where
  declare isEmpty :: 'c -> Bool

  property isEmptyP_eq_isEmpty (c :: 'c) := (isEmptyP c <=> (isEmpty c === True))
end-class


-- | For most collections, we want a decidable membership test. Since elements might occur multiple times
-- there is also a function that returns that number.
class Collection 'c 'e => CollectionWithMembershipTest 'c 'e | 'c -> 'e where
  declare occur :: Eq 'e => 'e -> 'c -> Natural

  declare member :: Eq 'e => 'e -> 'c -> Bool
  define member e c := (occur e c > 0)

  test Eq 'e => occur_insert_1 (e :: 'e) (c :: 'c) :=
                  (occur e (insert e c) == occur e c + 1) ||
                  ((occur e (insert e c) == 1) && (occur e c == 1))
  test Eq 'e => occur_insert_2 (e1 :: 'e) (e2 :: 'e) (c :: 'c) :=
                  (e1 /= e2) -->
                  ((occur e1 (insert e2 c) == occur e1 c))

  property Eq 'e => memberP_member (e :: 'e) (c :: 'c) :=
    (memberP e c <=> (member e c === True))
end-class

-- | Removing elements is also common. Remember that collections can contain elements multiple times and
```

```
106 -- order of elements might matter or not,
107 class (Eq 'e, CollectionWithMembershipTest 'c 'e) => CollectionWithDelete 'c 'e | 'c -> 'e where
108   -- remove the first occurence of an element in a container
109   declare delete :: 'e -> 'c -> 'c
110
111   -- remove all elements not satisfying a predicate from a container, order is preserved
112   declare filter :: ('e -> Bool) -> 'c -> 'c
113   define filter p s := fst (partition p s)
114
115   declare difference :: 'c -> 'c -> 'c
116   declare intersection :: 'c -> 'c -> 'c
117
118   -- split a container in all elements not satisfying a predicate and the ones that don't, order is pres
119   declare partition :: ('e -> Bool) -> 'c -> ('c, 'c)
120   define partition p s := (filter p s, filter (not . p) s)
121
122
123   test member_filter (e :: 'e) (c :: 'c) p :=
124     member e (filter p c) <-> (member e c && p e)
125
126   test occur_filter (e :: 'e) (c :: 'c) p :=
127     occur e (filter p c) == (if p e then occur e c else 0)
128
129   test occur_delete (e :: 'e) (c :: 'c) :=
130     occur e (delete e c) == max 0 (occur e c - 1)
131
132   test member_delete (e1 :: 'e) (e2 :: 'e) (c :: 'c) :=
133     (e1 /= e2) --> (occur e1 (delete e2 c) == occur e1 c)
134
135   test intersection_ok (e :: 'e) (c1 :: 'c) (c2 :: 'c) :=
136     occur e (intersection c1 c2) == min (occur e c1) (occur e c2)
137   test difference_ok (e :: 'e) (c1 :: 'c) (c2 :: 'c) :=
138     occur e (difference c1 c2) == max 0 (occur e c1 - occur e c2)
139
140 end-class
141
142 -- | A finite collection. One can fold over a finite collection. This means being able to get the size a
143 -- having decidable membership tests. Moreover, one can now also convert to a list.
144 class (CollectionWithMembershipTest 'c 'e, CollectionWithEmptynessTest 'c 'e, Foldable 'c 'e) => FiniteC
145   test empty_size := (size (empty :: 'c) == 0)
146   test null_ok (c :: 'c) := (isEmpty c <-> null c)
147
148   test size_fromToList (c :: 'c) := size ((fromList ((toList c) :: ['e])) :: 'c) == size c
149   test size_toFromList (l :: ['e]) := size ((fromList l) :: 'c) <= List.length l
150
151   test Eq 'e => elem_ok (e :: 'e) (c :: 'c) := member e c <-> elem e c
152
153   property EqP 'e => memberP_toList (e :: 'e) (c :: 'c) :=
154     (List.elemP e (toList (c::'c)) <=> memberP e c)
155
156   test Eq 'e => member_elem (e :: 'e) (c :: 'c) := (elem e c <-> member e c)
157 end-class
158
159
160 -- | collections might consider the order of elements. If we ignore the order,
161 -- we can define datatypes like sets and bags (multisets)
162 class (CollectionWithDelete 'c 'e, CollectionWithEmptynessTest 'c 'e) => CollectionWithSubsetTest 'c 'e
163   declare disjoint :: 'c -> 'c -> Bool
164
```

```
165   declare isSubsetOf :: 'c -> 'c -> Bool
166   declare isProperSubsetOf :: 'c -> 'c -> Bool
167
168   declare isSetEquiv :: 'c -> 'c -> Bool
169   define isSetEquiv xs ys := isSubsetOf xs ys && isSubsetOf ys xs
170
171   define disjoint xs ys := isEmpty (intersection xs ys)
172
173   test disjoint_ok (e :: 'e) (c1 :: 'c) (c2 :: 'c) := not (member e c1 && member e c2)
174   test isSubsetOf_ok (e :: 'e) (c1 :: 'c) (c2 :: 'c) := (occur e c1 <= occur e c2)
175   test isProperSubsetOf_ok (c1 :: 'c) (c2 :: 'c) :=  (isSubsetOf c1 c2) && not (isSubsetOf c2 c1)
176
177 end-class
178
179 class CollectionWithMembershipTest 'c 'e => SetLike 'c 'e | 'c -> 'e where
180   test Eq 'e => noDups_occur (e :: 'e) (c :: 'c) := occur e c <= 1
181 end-class
182
183 declare setEmpty :: SetLike 'c 'e => 'c
184 define setEmpty := mempty
185 {-# inline setEmpty #-}
186
187 declare setInsert :: SetLike 'c 'e => 'e -> 'c -> 'c
188 define setInsert := insert
189 {-# inline setInsert #-}
190
191 declare setFromList :: SetLike 'c 'e => ['e] -> 'c
192 define setFromList := fromList
193 {-# inline setFromList #-}
194
195 class CollectionWithMembershipTest 'c 'e => BagLike 'c 'e | 'c -> 'e where
196   test Eq 'e => occur_insert_bag (e :: 'e) (c :: 'c) := (occur e (insert e c) == occur e c + 1)
197 end-class
198
199 class (SetLike 'c 'e, CollectionWithDelete 'c 'e, CollectionWithEmptynessTest 'c 'e) => Set 'c 'e | 'c ->
200 end-class
201
202 class (Set 'c 'e, FiniteCollection 'c 'e) => FiniteSet 'c 'e | 'c -> 'e where
203 end-class
204
205 class (BagLike 'c 'e, CollectionWithDelete 'c 'e, CollectionWithEmptynessTest 'c 'e) => Bag 'c 'e | 'c ->
206 end-class
207
208 class (Bag 'c 'e, FiniteCollection 'c 'e) => FiniteBag 'c 'e | 'c -> 'e where
209 end-class
210
211 end-module
```

1.14 Prelude/Default.ad

Outline

Content

```
module Prelude.Default (
  Default(..),
  unreachable
) where

{-# NoImplicitPrelude #-}

class Default 'a where
  declare default :: 'a
  define default := ???
  {-# inline default #-}

  declare undefined :: 'a
  define undefined := default
  {-# inline undefined #-}
end-class

declare unreachable :: Default 'a => 'a
define unreachable := default
{-# inline unreachable #-}

template DeriveInstanceDefaultBody (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
define default := ???
\{-# INLINE default #-\}
'''

register-derive-template Default DeriveInstanceDefaultBody

end-module
```

1.15 Prelude/Either.ad

Outline

Content

```
module Prelude.Either (
  Either(..),
  destEither
) where

{-# NoImplicitPrelude #-}
import Prelude.Bool
import Prelude.Eq
import Prelude.Ord

```

```
datatype Either 'a 'b :=
    Left 'a
  | Right 'b


declare destEither :: ('a -> 'c) -> ('b -> 'c) -> Either 'a 'b -> 'c
define destEither lf _ (Left v)  := lf v
     | destEither _ rf (Right v) := rf v

derive Either instances (EqP, Eq, Ord)

end-module
```

1.16 Prelude/Eq.ad

Outline

Content

```
module Prelude.Eq (
  (====),
  EqP(..),
  Eq(..)
) where

{-# NoImplicitPrelude #-}

import Prelude.Bool
import Prelude.Unit ()
import Prelude.Template

-- | build-in equality, it checks whether there is exactly the same representation of both values
declare non-exec (====) :: 'a -> 'a -> Prop


-- | sometimes, we do not want absolute identity, but define a congruence relation instead. If we
-- for example define a set-datastructure via distinct lists, (====) will check whether the lists
-- are exactly the same, while (===) checks whether the lists are permutations of each other
class EqP 'a where
  declare non-exec (===) :: 'a -> 'a -> Prop
  define non-exec (===) x y := notP (x =/= y)

  declare non-exec (=/=) :: 'a -> 'a -> Prop
  define non-exec (=/=) x y := notP (x === y)

  property negEqP   (x :: 'a) y   := (x === y) <=/=> (x =/= y)
  property reflEqP  (x :: 'a)     := (x === x)
  property symEqP   (x :: 'a) y   := (x === y) <=> (y === x)
  property transEqP (x :: 'a) y z := (x === y) ==> (y === z) ==> (x === z)

  property congEqP f x y := ((((x :: 'a) === y) ==> ((f x :: 'a) === f y)))

  {-# minimal (===) | (=/=) #-}
end-class

template DeriveInstanceEqPBodyFull (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
  {{ var constrs := ty.constructors;
     var clauses := [] :: [String];
     foreach var i in constrs.indices {
       name-scope {
         var (args1, constr1) := !ConstWithArgs(True, "x", constrs[i]);
         var (args2, constr2) := !ConstWithArgs(True, "y", constrs[i]);
         var cl := "(===) " ++ constr1 ++ " " ++ constr2 ++ " := ";
         if (args1.length > 0) {
           var rhs := [] :: [String];
           foreach var xy in zip(args1, args2) {
             rhs := insertAtEnd("(" ++ #1(xy) ++ " === " ++ #2(xy) ++ ")", rhs);
           }
           cl := cl ++ !Intercalate(" /\\ ", rhs);
         } else {
           cl := cl ++ "true"
         }
         clauses := insertAtEnd(cl, clauses);
       }
     }
     if (constrs.length > 1) {
```

```
        clauses := insertAtEnd("(===) _ _ := false", clauses);
      }
      var recDecl, indentRec := "";
      if (ty.isRec) { recDecl := "rec "; indentRec := "    "; }
  }}
{{= "define non-exec " ++ recDecl ++ !Intercalate("\n              "++indentRec++"| ", clauses) =}}
'''

register-derive-template EqP DeriveInstanceEqPBodyFull as "full"

template DeriveInstanceEqPBodyEq (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
define non-exec (===) x y := bool2prop (x == y)
'''

register-derive-template EqP DeriveInstanceEqPBodyEq - as "eq"

template DeriveInstanceEqPBody (cl :: ClassID, ty :: TypeID, args :: [String]) := '''
{{% if (args.length == 0) %}}
{{! DeriveInstanceEqPBodyEq(cl, ty, args) !}}
{{% else %}}
{{! DeriveInstanceEqPBodyFull(cl, ty, args) !}}
{{% end-if %}}
'''

register-derive-template EqP DeriveInstanceEqPBody

instance EqP 'b => EqP ('a -> 'b) where
  define non-exec (===) f1 f2 := forallP a. f1 a === f2 a
end-instance

-- | (===) is not computable. An example is an equality check for functions, which might not be
-- computable. For computing computability (==) is provided.
class EqP 'a => Eq 'a where
  declare (==) :: 'a -> 'a -> Bool
  define (==) x y := not (x /= y)

  declare (/=) :: 'a -> 'a -> Bool
  define (/=) x y := not (x == y)

  test negEq   (x :: 'a) y   := (x == y) <-> (not (x /= y))
  test reflEq  (x :: 'a)     := (x == x)
  test symEq   (x :: 'a) y   := (x == y) <-> (y == x)
  test transEq (x :: 'a) y z := (x == y) --> (y == z) --> (x == z)

  -- property eqProp x y := (((x :: 'a) == y) === True) <=> (x === y)
  {-# minimal (==) | (/=) #-}
end-class

template DeriveInstanceEqBody (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
  {{ var constrs := ty.constructors;
     var clauses := [] :: [String];
     foreach var i in constrs.indices {
       name-scope {
         var (args1, constr1) := !ConstWithArgs(True, "x", constrs[i]);
         var (args2, constr2) := !ConstWithArgs(True, "y", constrs[i]);
         var cl := "(==) " ++ constr1 ++ " " ++ constr2 ++ " := ";
         if (args1.length > 0) {
           var rhs := [] :: [String];
           foreach var xy in zip(args1, args2) {
```

```
117               rhs := insertAtEnd("(" ++ #1(xy) ++ " == " ++ #2(xy) ++ ")", rhs);
118             }
119             cl := cl ++ !Intercalate(" && ", rhs);
120           } else {
121             cl := cl ++ "True"
122           }
123           clauses := insertAtEnd(cl, clauses);
124         }
125       }
126       if (constrs.length > 1) {
127         clauses := insertAtEnd("(==) _ _ := False", clauses);
128       }
129       var recDecl, indentRec := "";
130       if (ty.isRec) { recDecl := "rec "; indentRec := "    "; }
131   }}
132 {{= "define " ++ recDecl ++ !Intercalate("\n     "++indentRec++"| ", clauses) =}}
133 '''
134
135 register-derive-template Eq DeriveInstanceEqBody
136
137
138 instance Eq () where
139   define (==) _ _ := True
140 end-instance
141
142 instance EqP () where
143   define non-exec (===) _ _ := true
144 end-instance
145
146 derive Bool instances (EqP, Eq)
147
148 end-module
```

1.17 Prelude/Foldable.ad

Outline

Content

```
module Prelude.Foldable (
  Foldable(..)
) where

{-# NoImplicitPrelude #-}

import Prelude.Function
import Prelude.Bool
import Prelude.Num
import Prelude.Eq
import Prelude.BasicClasses
import Prelude.Unit ()
import           Prelude.List hiding (foldl, foldr, foldlWithAbort, elem, length, null, all, any)
import qualified Prelude.List as List
import           Prelude.Maybe hiding (map, bind)

{-# allow-similar-names "foldl", "foldr", "foldlWithAbort", "foldrWithAbort" #-}
class Foldable 't 'e | 't -> 'e where

  declare toList :: 't -> ['e]

  declare foldr :: ('e -> 'acc -> 'acc) -> 'acc -> 't -> 'acc
  {-# static resolve foldr #-}

  declare foldl :: ('acc -> 'e -> 'acc) -> 'acc -> 't -> 'acc
  {-# static resolve foldl #-}

  declare foldlWithAbort :: ('acc -> Bool) -> ('acc -> 'e -> 'acc) -> 'acc -> 't -> 'acc
  {-# static resolve foldlWithAbort #-}

  declare elem :: Eq 'e => 'e -> 't  -> Bool
  declare all :: ('e -> Bool) -> 't -> Bool
  declare any :: ('e -> Bool) -> 't -> Bool

  declare size :: 't -> Natural
  declare null :: 't -> Bool

  define foldl f a t := List.foldl f a (toList t)
  define foldr f a t := List.foldr f a (toList t)
  define foldlWithAbort abort f a t := List.foldlWithAbort abort f a (toList t)
  define toList := foldr (:) []
  define size := foldl (fn a (_::'e) -> succ a) 0
  define null t := foldlWithAbort not (constant (constant False)) True t
  define elem e := any ((==) e)
  define any p t := foldlWithAbort id (constant p) False t
  define all p t := foldlWithAbort not (constant p) True t

  {-# minimal toList | foldr #-}

end-class
```

```
51
52  instance Foldable ['a] 'a where
53    define toList := id
54    {-# inline toList #-}
55
56    define foldl := List.foldl
57    {-# inline foldl #-}
58
59    define foldr := List.foldr
60    {-# inline foldr #-}
61
62    define foldlWithAbort := List.foldlWithAbort
63    {-# inline foldlWithAbort #-}
64
65    define size := List.length
66    {-# inline size #-}
67
68    define null := List.null
69    {-# inline null #-}
70
71    define any := List.any
72    {-# inline any #-}
73
74    define all := List.all
75    {-# inline all #-}
76
77    declare elem :: Eq 'a => 'a -> ['a] -> Bool
78    define elem := List.elem
79    {-# inline elem #-}
80  end-instance
81
82
83  instance Foldable (Maybe 'a) 'a where
84
85    define foldr _ z Nothing := z
86         | foldr f z (Just x) := f x z
87
88    define foldl _ z Nothing := z
89         | foldl f z (Just x) := f z x
90
91    define toList Nothing := []
92         | toList (Just x) := [x]
93
94    define any _ Nothing := False
95         | any p (Just x) := p x
96
97    define all _ Nothing := True
98         | all p (Just x) := p x
99
100   define foldlWithAbort _ _ z Nothing := z
101        | foldlWithAbort ab f z (Just x) :=
102            if (ab z) then z else f z x
103
104 end-instance
105
106
107 instance Foldable ('a, 'a) 'a where
108   define toList (x1, x2) := [x1,x2]
109   define null := constant False
```

```
110   {-# inline null #-}
111
112   define size := constant 2
113   {-# inline size #-}
114 end-instance
115
116 instance Foldable ('a, 'a, 'a) 'a where
117   define toList (x1, x2, x3) := [x1,x2,x3]
118   define null := constant False
119   {-# inline null #-}
120
121   define size := constant 3
122   {-# inline size #-}
123 end-instance
124
125 instance Foldable ('a, 'a, 'a, 'a) 'a where
126   define toList (x1, x2, x3, x4) := [x1,x2,x3,x4]
127   define null := constant False
128   {-# inline null #-}
129
130   define size := constant 4
131   {-# inline size #-}
132 end-instance
133
134 instance Foldable ('a, 'a, 'a, 'a, 'a) 'a where
135   define toList (x1, x2, x3, x4, x5) := [x1,x2, x3, x4, x5]
136   define null := constant False
137   {-# inline null #-}
138
139   define size := constant 5
140   {-# inline size #-}
141 end-instance
142
143 instance Foldable ('a, 'a, 'a, 'a, 'a, 'a) 'a where
144   define toList (x1, x2, x3, x4, x5, x6) := [x1,x2, x3, x4, x5, x6]
145   define null := constant False
146   {-# inline null #-}
147
148   define size := constant 6
149   {-# inline size #-}
150 end-instance
151
152 instance Foldable ('a, 'a, 'a, 'a, 'a, 'a, 'a) 'a where
153   define toList (x1, x2, x3, x4, x5, x6, x7) := [x1,x2, x3, x4, x5, x6, x7]
154
155   define null := constant False
156   {-# inline null #-}
157
158   define size := constant 7
159   {-# inline size #-}
160 end-instance
161
162 instance Foldable ('a, 'a, 'a, 'a, 'a, 'a, 'a, 'a) 'a where
163   define toList (x1, x2, x3, x4, x5, x6, x7, x8) := [x1,x2, x3, x4, x5, x6, x7, x8]
164   define null := constant False
165   {-# inline null #-}
166
167   define size := constant 8
168   {-# inline size #-}
```

```
end-instance


end-module
```

1.18 Prelude/Function.ad

Outline

Content

```
module Prelude.Function (
  Dot(..),
  flip,
  constant,
  id,
  ($)
) where

{-# NoImplicitPrelude #-}

class Dot 'a 'b 'c | 'a 'b -> 'c where
  declare (.) :: 'a -> 'b -> 'c
end-class

instance Dot ('b -> 'c) ('a -> 'b2) ('a -> 'c) | 'b2 -> 'b where
  define (.) f1 f2 x := f1 (f2 x)
  {-# inline (.) #-}
end-instance

declare id :: 'a -> 'a
define id x := x
{-# inline id #-}

declare constant :: 'a -> 'b -> 'a
define constant x := fn _ -> x
{-# inline constant #-}

declare flip :: ('a -> 'b -> 'c) -> ('b -> 'a -> 'c)
define flip f b a := f a b

declare ($) :: ('a -> 'b) -> 'a -> 'b
define ($) f x :=  f x
{-# inline ($) #-}

end-module
```

1.19 Prelude/Functor.ad

Outline

Content

```
1  module Prelude.Functor (
2    Functor(..),
3    (<$>)
4  ) where
5
6  {-# NoImplicitPrelude #-}
7
8  import Prelude.Eq
9  import Prelude.Function
10
11 {-# static resolve Functor #-}
12 class Functor ('F :: * -> *) where
13   declare fmap :: ('a -> 'b) -> 'F 'a -> 'F 'b
14
15   property EqP ('F 'c) => fmap_composition (f :: 'b -> 'c) (g :: 'a -> 'b) (x :: 'F 'a) := fmap (f . g)
16   property EqP ('F 'a) => fmap_identity (x :: 'F 'a) := fmap id x === x
17 end-class
18
19 alias (<$>) := fmap
20
21 end-module
```

1.20 Prelude/List.ad

Outline

Content

```
1  module Prelude.List (
2    List(..),
3    Cons,
4    snoc,
5    last,
6    map,
7    (++),
8    filter,
9    partition,
10   removeFirst,
11   takeFirst,
```

```
12    concat,
13    concatMap,
14    null,
15    length,
16    count,
17    foldl, foldr, foldlWithAbort,
18    scanl, scanr,
19    reverse,
20    revAppend,
21    and,
22    or,
23    all,
24    any,
25    elem,
26    notElem,
27    andP,
28    orP,
29    allP,
30    anyP,
31    elemP,
32    notElemP,
33    nub,
34    nubBy
35  ) where
36
37  {-# NoImplicitPrelude #-}
38  {-# allow-similar-names "foldl", "foldr", "scanl", "scanr" #-}
39
40  import Prelude.Bool
41  import Prelude.Num
42  import Prelude.Function
43  import Prelude.Default
44  import Prelude.Eq
45  import Prelude.Ord
46  import Prelude.Functor
47  import Prelude.Maybe hiding (map)
48
49  {-# naming-convention constructor - #-}
50  datatype rec List 'a := Nil | (:) 'a (List 'a)
51  {-# naming-convention constructor reset #-}
52
53  alias Cons := (:)
54
55  instance Default ['a] where
56    define default := []
57  end-instance
58
59
60  declare map :: ('a -> 'b) -> ['a] -> ['b]
61  define rec map _ [] := []
62           | map f (x : xs) := (f x) : (map f xs)
63
64  instance Functor List where
65    define fmap := map
66    {-# inline fmap #-}
67  end-instance
68
69
70  declare (++) :: ['a] -> ['a] -> ['a]
```

```
71  define rec (++) [] ys := ys
72          | (++) (x:xs) ys := x : (xs ++ ys)
73
74  declare snoc :: ['a] -> 'a -> ['a]
75  define rec snoc [] y := [y]
76          | snoc (x:xs) y := x : (snoc xs y)
77
78  declare last :: ['a] -> Maybe 'a
79  define rec last [] := Nothing
80          | last [x] := Just x
81          | last (_:x:xs) := last (x:xs)
82
83  declare destSnoc :: ['a] -> Maybe (['a], 'a)
84  define rec destSnoc [] := Nothing
85          | destSnoc (x:xs) :=
86              case destSnoc xs of
87                  Nothing -> Just ([], x)
88                | Just (ys, y) -> Just (x:ys, y)
89              end
90
91  declare caseListSnoc :: 'a -> (['c] -> 'c -> 'a) -> ['c] -> 'a
92  define caseListSnoc nilV snocV l :=
93    case (destSnoc l) of
94        Nothing -> nilV
95      | Just (xs, x) -> snocV xs x
96    end
97
98  constructor-family caseListSnoc (Nil, snoc)
99
100 declare filter :: ('a -> Bool) -> List 'a -> List 'a
101 define rec filter _ [] := []
102         | filter p (x:xs) :=
103             if (p x) then (x : (filter p xs)) else filter p xs
104
105 declare partition :: ('a -> Bool) -> List 'a -> (List 'a, List 'a)
106 define rec partition _ [] := ([], [])
107         | partition p (x:xs) :=
108             let
109               val (lt, lf) = partition p xs
110             in
111               if p x then (x : lt, lf) else (lt, x : lf)
112             end
113
114 declare takeFirst :: ('a -> Bool) -> List 'a -> Maybe (List 'a, 'a)
115 define takeFirst p := takeFirstAcc p []
116
117 declare takeFirstAcc :: ('a -> Bool) -> List 'a -> List 'a -> Maybe (List 'a, 'a)
118 define rec takeFirstAcc _ _ [] := Nothing
119         | takeFirstAcc p acc (x:xs) :=
120             if p x then Just (reverse acc ++ xs, x) else
121               takeFirstAcc p (x:acc) xs
122
123 declare removeFirst :: ('a -> Bool) -> List 'a -> List 'a
124 define removeFirst p l := case takeFirst p l of
125                             Nothing -> l
126                           | Just (l', _) -> l'
127                         end
128
129 declare concat :: [['a]] -> ['a]
```

```
130 define concat xss := foldr (++) [] xss
131
132 declare concatMap :: ('a -> ['b]) -> ['a] -> ['b]
133 define concatMap f := concat . (map f)
134
135 declare null :: ['a] -> Bool
136 define null [] := True
137      | null (_:_) :=  False
138
139 declare foldl :: ('a -> 'b -> 'a) -> 'a -> ['b] -> 'a
140 define rec foldl _ z [] := z
141          | foldl f z (x:xs) := foldl f (f z x) xs
142
143 declare foldlWithAbort :: ('a -> Bool) -> ('a -> 'b -> 'a) -> 'a -> ['b] -> 'a
144 define rec foldlWithAbort _ _ z [] := z
145          | foldlWithAbort ab f z (x:xs) :=
146              if (ab z) then z else foldlWithAbort ab f (f z x) xs
147
148 declare scanl :: ('a -> 'b -> 'a) -> 'a -> ['b] -> ['a]
149 define rec scanl _ q [] :=  [q]
150          | scanl f q (x:xs) := q : (scanl f (f q x) xs)
151
152
153 declare foldr :: ('a -> 'b -> 'b) -> 'b -> ['a] -> 'b
154 define rec foldr _ z []     := z
155          | foldr f z (x:xs) := f x (foldr f z xs)
156
157 declare scanr :: ('a -> 'b -> 'b) -> 'b -> ['a] -> ['b]
158 define rec scanr _ q0 [] := [q0]
159          | scanr f q0 (x:xs) :=
160              case (scanr f q0 xs) of
161                  [] -> unreachable
162                | q : qs ->  (f x q) : (q : qs)
163              end
164
165 declare length :: ['a] -> Natural
166 define length := foldl (fn c _ -> c + 1) 0
167
168 declare count :: Eq 'a => 'a -> ['a] -> Natural
169 define count e := foldl (fn c e' -> if (e == e') then c + 1 else c) 0
170
171 declare revAppend :: ['a] -> ['a] -> ['a]
172 define rec revAppend acc [] := acc
173          | revAppend acc (x:xs) := revAppend (x:acc) xs
174
175 declare reverse :: ['a] -> ['a]
176 define reverse := revAppend []
177
178
179 declare and :: [Bool] -> Bool
180 define rec and [] := True
181          | and (False:_) := False
182          | and (True:bs) := and bs
183
184 declare non-exec andP :: [Prop] -> Prop
185 define non-exec rec andP [] := true
186                   | andP (p:ps) := p /\ andP ps
187
188 declare or :: [Bool] -> Bool
```

```
189 define rec or [] := False
190          | or (True:_) := True
191          | or (False:bs) := or bs
192
193 declare non-exec orP :: [Prop] -> Prop
194 define non-exec rec orP [] := false
195                   | orP (p:ps) := p \/ orP ps
196
197 declare all :: ('a -> Bool) -> ['a] -> Bool
198 define rec all _ [] := True
199          | all f (x:xs) := if (f x) then all f xs else False
200
201 declare any :: ('a -> Bool) -> ['a] -> Bool
202 define rec any _ [] := False
203          | any f (x:xs) := if (f x) then True else any f xs
204
205 declare non-exec allP :: ('a -> Prop) -> ['a] -> Prop
206 define non-exec rec allP _ [] := true
207                   | allP f (x:xs) := (f x) /\ allP f xs
208
209 declare non-exec anyP :: ('a -> Prop) -> ['a] -> Prop
210 define non-exec rec anyP _ [] := false
211                   | anyP f (x:xs) := (f x) \/ anyP f xs
212
213
214 declare elem :: Eq 'a => 'a -> ['a] -> Bool
215 define elem x := any ((==) x)
216
217
218 declare notElem :: Eq 'a => 'a -> ['a] -> Bool
219 define notElem x xs := not (elem x xs)
220 {-# inline notElem #-}
221
222
223 declare non-exec elemP :: EqP 'a => 'a -> ['a] -> Prop
224 define non-exec elemP x := anyP ((===) x)
225
226
227 declare non-exec notElemP :: EqP 'a => 'a -> ['a] -> Prop
228 define non-exec notElemP x xs := notP (elemP x xs)
229 {-# inline notElemP #-}
230
231
232 derive List instances (Eq, EqP, Ord)
233
234 declare nubBy :: ('a -> 'a -> Bool) -> ['a] -> ['a]
235 define rec nubBy _ [] := []
236          | nubBy p (x:xs) := x : nubBy p (filter (fn y -> not (p x y)) xs)
237
238 declare nub :: Eq 'a => ['a] -> ['a]
239 define nub := nubBy (==)
240 {-# inline nub #-}
241
242
243 end-module
```

1.21 Prelude/ListCollection.ad

Outline

Content

```
1  module Prelude.ListCollection (
2  ) where
3
4  {-# NoImplicitPrelude #-}
5
6  import Prelude.Eq
7  import Prelude.Bool
8  import Prelude.Maybe hiding (map)
9  import Prelude.Collection
10 import Prelude.List hiding (filter, partition)
11 import qualified Prelude.List as List
12
13 instance Collection ['e] 'e where
14   define insert := (:)
15   {-# inline insert #-}
16
17   define fromList l := l
18   {-# inline fromList #-}
19
20   define non-exec isEmptyP c := bool2prop (null c)
21   define non-exec memberP := elemP
22   {-# inline memberP #-}
23
24 end-instance
25
26 instance CollectionWithMembershipTest ['e] 'e where
27   define occur := count
```

```
28    {-# inline occur #-}
29
30    define member := elem
31    {-# inline member #-}
32
33  end-instance
34
35  instance CollectionWithEmptynessTest ['e] 'e where
36    define isEmpty := null
37    {-#  inline isEmpty #-}
38  end-instance
39
40
41  declare listIntersection :: Eq 'a => ['a] -> ['a] -> ['a]
42  define rec listIntersection [] _ := []
43           | listIntersection _ [] := []
44           | listIntersection (x:xs) l :=
45                case (takeFirst ((==) x) l) of
46                  Nothing -> listIntersection xs l
47                | Just (l', _) -> x:(listIntersection xs l')
48                end
49
50  instance Eq 'e => CollectionWithDelete ['e] 'e where
51    define delete e := removeFirst ((==) e)
52    {-# inline delete #-}
53
54    define filter := List.filter
55    {-# inline filter #-}
56
57    define partition := List.partition
58    {-# inline partition #-}
59
60    define difference := foldl (fn l e -> List.removeFirst ((==) e) l)
61    {-# inline difference #-}
62
63    define intersection := listIntersection
64    {-# inline intersection #-}
65  end-instance
66
67  instance FiniteCollection ['e] 'e where
68  end-instance
69
70  end-module
```

1.22 Prelude/Literal.ad

Outline

Content

```
module Prelude.Literal (
  BuiltInChar,
  BuiltInNum,
  BuiltInDecimal,
  BuiltInString,
  LiteralNum(..),
  LiteralDecimal(..),
  LiteralChar(..),
  LiteralString(..)
) where

{-# NoImplicitPrelude #-}

import Prelude.Bool
import Prelude.Eq

-- | internal, build-in number represenation
{-# allow-similar-names "builtInNumEq", "builtInNumEqP", "BuiltInNumRep" #-}
type BuiltInNum
type BuiltInNumRep

declare builtInNumEq :: BuiltInNum -> BuiltInNum -> Bool
declare non-exec builtInNumEqP :: BuiltInNum -> BuiltInNum -> Prop

instance EqP BuiltInNum where
  define non-exec (===) := builtInNumEqP
  {-# inline (===) #-}
end-instance

instance Eq BuiltInNum where
  define (==) := builtInNumEq
  {-# inline (==) #-}
end-instance

-- | type class for making number literals
class Eq 'a => LiteralNum 'a where
  declare mkLiteralNum :: (BuiltInNumRep, BuiltInNum) -> 'a
end-class
{-# static resolve LiteralNum #-}

instance LiteralNum BuiltInNum where
  define mkLiteralNum (_, bi) := bi
  {-# inline mkLiteralNum #-}
end-instance


-- | internal, build-in number represenation
{-# allow-similar-names "builtInNumEq", "builtInNumEqP", "BuiltInNumRep" #-}
type BuiltInChar

declare builtInCharEq :: BuiltInChar -> BuiltInChar -> Bool
declare non-exec builtInCharEqP :: BuiltInChar -> BuiltInChar -> Prop

-- | type class for making number literals
class Eq 'a => LiteralChar 'a where
  declare mkLiteralChar :: BuiltInChar -> 'a
end-class
```

```
{-# static resolve LiteralChar #-}

instance EqP BuiltInChar where
  define non-exec (===) := builtInCharEqP
  {-# inline (===) #-}
end-instance

instance Eq BuiltInChar where
  define (==) := builtInCharEq
  {-# inline (==) #-}
end-instance

instance LiteralChar BuiltInChar where
  define mkLiteralChar c := c
  {-# inline mkLiteralChar #-}
end-instance


-- | internal, build-in number represenation
type BuiltInString

declare builtInStringEq :: BuiltInString -> BuiltInString -> Bool
declare non-exec builtInStringEqP :: BuiltInString -> BuiltInString -> Prop

instance EqP BuiltInString where
  define non-exec (===) := builtInStringEqP
  {-# inline (===) #-}
end-instance

instance Eq BuiltInString where
  define (==) := builtInStringEq
  {-# inline (==) #-}
end-instance

-- | type class for making number literals
class Eq 'a => LiteralString 'a where
  declare mkLiteralString :: BuiltInString -> 'a
end-class
{-# static resolve LiteralString #-}

instance LiteralString BuiltInString where
  define mkLiteralString s := s
  {-# inline mkLiteralString #-}
end-instance

-- | internal, build-in number represenation
type BuiltInDecimal

declare builtInDecimalEq :: BuiltInDecimal -> BuiltInDecimal -> Bool
declare non-exec builtInDecimalEqP :: BuiltInDecimal -> BuiltInDecimal -> Prop

declare builtInNumToDecimal :: BuiltInNum -> BuiltInDecimal

instance EqP BuiltInDecimal where
  define non-exec (===) := builtInDecimalEqP
  {-# inline (===) #-}
end-instance

instance Eq BuiltInDecimal where
```

```
117   define (==) := builtInDecimalEq
118   {-# inline (==) #-}
119 end-instance
120
121 -- | type class for making decimal literals
122 class LiteralNum 'a => LiteralDecimal 'a where
123   declare mkLiteralDecimal :: BuiltInDecimal -> 'a
124 end-class
125 {-# static resolve LiteralDecimal #-}
126
127 instance LiteralNum BuiltInDecimal where
128   define mkLiteralNum (_, i) := builtInNumToDecimal i
129   {-# inline mkLiteralNum #-}
130 end-instance
131
132 instance LiteralDecimal BuiltInDecimal where
133   define mkLiteralDecimal d := d
134   {-# inline mkLiteralDecimal #-}
135 end-instance
136
137
138 end-module
```

1.23 Prelude/Maybe.ad

Outline

Content

```
1  module Prelude.Maybe (
2    Maybe(..),
3    isJust,
4    isNothing,
5    fromMaybe,
6    map,
7    fromMaybeMap,
8    bind
9  ) where
10
```

```
{-# NoImplicitPrelude #-}

import Prelude.Bool
import Prelude.Eq
import Prelude.Ord
import Prelude.Default
import Prelude.Template
import Prelude.Functor

datatype Maybe 'a :=
    Nothing
  | Just 'a

instance Default (Maybe 'a) where
  define default := Nothing
end-instance

declare map :: ('a -> 'b) -> Maybe 'a -> Maybe 'b
define map _ Nothing := Nothing
     | map f (Just x) := Just (f x)

instance Functor Maybe where
  define fmap := map
  {-# inline fmap #-}
end-instance

declare bind :: Maybe 'a -> ('a -> Maybe 'b) -> Maybe 'b
define bind Nothing _ := Nothing
     | bind (Just x) f := f x

declare fromMaybe :: 'a -> Maybe 'a -> 'a
define fromMaybe d Nothing := d
     | fromMaybe _d (Just v) := v

declare fromMaybeMap :: 'b -> ('a -> 'b) -> Maybe 'a -> 'b
define fromMaybeMap d f m := fromMaybe d (map f m)

generate-code GenerateRecognisers(type Maybe)

derive Maybe instances (EqP, Eq, Ord)

end-module
```

1.24 Prelude/Monad.ad

Outline

Content

```
1  module Prelude.Monad (
2    Monad(..),
3    bind, (=<<),
4    pure,
5    sequence,
6    sequence_,
7    mapM,
8    mapM_,
9    guard,
10   MonadFail(..)
11 ) where
12
13 {-# NoImplicitPrelude #-}
14
15 import Prelude.Function
16 import Prelude.Bool
17 import Prelude.Eq
18 import Prelude.Unit ()
19 import Prelude.String
20 import Prelude.BasicClasses
21 import           Prelude.List hiding (foldl, foldr, elem, length, null)
22 import qualified Prelude.List as List
23 import qualified Prelude.Maybe as Maybe
24 import           Prelude.Maybe hiding (map, bind)
25
26
27 {-# static resolve Monad #-}
28 class Monad ('M :: * -> *) where
```

```
29    declare (>>=)  :: 'M 'a -> ('a -> 'M 'b) -> 'M 'b
30    declare (>>)   :: 'M 'a -> 'M 'b -> 'M 'b
31    declare return :: 'a -> 'M 'a
32
33    define (>>) m k := m >>= (constant k)
34    {-# inline (>>) #-}
35
36    property EqP ('M 'b) => monad_left_identity (f :: 'a -> 'M 'b) a := ((return a) >>= f) === (f a)
37    property EqP ('M 'b) => monad_right_identity (a :: 'M 'b) := (a >>= return) === a
38    property EqP ('M 'b) => monad_assoc (m :: 'M 'a) (f :: 'a -> 'M 'c) (g :: 'c -> 'M 'b) := ((m >>= f) >
39  end-class
40
41  alias bind := (>>=)
42  alias pure := return
43
44  declare (=<<) :: !Monad 'M => ('a -> 'M 'b) -> 'M 'a -> 'M 'b
45  define (=<<) f m := m >>= f
46  {-# inline (=<<) #-}
47
48  declare guard :: (!Monoid ('M ()), !Monad 'M) => Bool -> 'M ()
49  define guard True := return ()
50       | guard False := mempty
51  {-# inline guard #-}
52
53  declare sequence' :: (['a] -> 'mas)
54    -> ('ma -> ('a -> 'mas) -> 'mas)
55    -> ('mas -> ['a] -> 'mas) -> 'mas)
56    -> ['ma]
57    -> 'mas
58  define sequence' ret bind1 bindl :=
59    List.foldr (fn p q ->
60      bind1 p (fn x -> bindl q (fn xs -> ret (x:xs)))
61    ) (ret [])
62
63
64  declare sequence :: !Monad 'M => ['M 'a] -> 'M ['a]
65  define sequence := sequence' return bind bind
66  {-# inline sequence #-}
67
68  declare sequence_ :: !Monad 'M => ['M 'a] -> 'M ()
69  define sequence_ l := sequence l >> return ()
70  {-# inline sequence_ #-}
71
72  declare mapM' :: (['b] -> 'mbs) -> ('mb -> ('b -> 'mbs) -> 'mbs) -> ('mbs -> (['b] -> 'mbs) -> 'mbs) ->
73  define mapM' ret b1 bl f xs := sequence' ret b1 bl (map f xs)
74
75  declare mapM :: !Monad 'M => ('a -> 'M 'b) -> ['a] -> 'M ['b]
76  define mapM := mapM' return bind bind
77  {-# inline mapM #-}
78
79  declare mapM_ :: !Monad 'M => ('a -> 'M 'b) -> ['a] -> 'M ()
80  define mapM_ f xs := (mapM f xs) >> return ()
81  {-# inline mapM_ #-}
82
83
84  instance Monad List where
85    define return x := [x]
86    {-# inline return #-}
87
```

```
88    define (>>=) xs f := concatMap f xs
89    {-# inline (>>=) #-}
90  end-instance
91
92  instance Monad Maybe where
93    define return := Just
94    {-# inline return #-}
95
96    define (>>=) := Maybe.bind
97    {-# inline (>>=) #-}
98  end-instance
99
100
101
102 {-# static resolve MonadFail #-}
103 class !Monad 'M => MonadFail ('M :: * -> *) where
104   declare fail :: String -> 'M 'a
105
106   property EqP ('M 'b) => monad_fail (f :: 'a -> 'M 'b) s := ((fail s) >>= f) === fail s
107 end-class
108
109 instance MonadFail Maybe where
110   define fail := constant Nothing
111   {-# inline fail #-}
112 end-instance
113
114 end-module
```

1.25 Prelude/Num.ad

Outline

Content

```
1  module Prelude.Num (
2    NumPlus(..),
3    NumMinus(..),
4    NumMult(..),
5    NumDiv(..),
6    NumDivMod(..),
7    Natural(..)
8  ) where
9
10 {-# NoImplicitPrelude #-}
11
12 import Prelude.Bool
13 import Prelude.Default
14 import Prelude.Eq
15 import Prelude.Ord
16 import Prelude.Literal
17 import Prelude.Tuple
18
19 datatype rec Natural :=
20     Zero
21   | Suc Natural
22
23 instance Default Natural where
24   define default := Zero
25   {-# inline default #-}
26 end-instance
27
28 derive Natural instances (EqP, Eq, Ord)
29
30 class NumPlus 'a where
31   declare (+) :: 'a -> 'a -> 'a
32 end-class
33
34 class NumMinus 'a where
35   declare (-) :: 'a -> 'a -> 'a
```

```
36 end-class
37
38 class NumMult 'a where
39   declare (*) :: 'a -> 'a -> 'a
40 end-class
41
42 class NumDiv 'a where
43   declare (/) :: 'a -> 'a -> 'a
44 end-class
45
46 class NumDivMod 'a where
47   declare divmod :: 'a -> 'a -> ('a, 'a)
48   declare div :: 'a -> 'a -> 'a
49   declare mod :: 'a -> 'a -> 'a
50
51   define divmod (x :: 'a) y := (x `div` y, x `mod` y)
52   define div x y := fst (divmod x y)
53   define mod x y := snd (divmod x y)
54 end-class
55
56
57 instance LiteralNum Natural where
58   define mkLiteralNum _ := ???
59 end-instance
60
61 instance NumPlus Natural where
62   define rec (+) n Zero := n
63              | (+) n (Suc m) := (+) (Suc n) m
64 end-instance
65
66 instance NumMinus Natural where
67   define rec (-) n Zero := n
68              | (-) Zero _ := Zero
69              | (-) (Suc n) (Suc m) := (-) n m
70 end-instance
71
72 instance NumMult Natural where
73   define rec (*) _ Zero := Zero
74              | (*) n (Suc m) := (n + (n * m))
75 end-instance
76
77 instance NumDivMod Natural where
78   define rec divmod _ Zero := ???
79              | divmod x y :=
80                   if x < y then
81                     (0, x)
82                   else
83                     let
84                       val (d, m) = divmod (x - y) y
85                     in (d+1, m) end
86 end-instance
87
88 instance NumDiv Natural where
89   define (/) := div
90   {-# inline (/) #-}
91 end-instance
92
93
94 instance LiteralNum Bool where
```

```
95    define mkLiteralNum (_, 0) := False
96         | mkLiteralNum (_, _) := True
97
98    {-# set-num-literal-boundaries 0 1 #-}
99  end-instance
100
101
102 end-module
```

1.26 Prelude/Ord.ad

Outline

Content

```
module Prelude.Ord (
  type Ordering(..),
  class Ord(..),
  comparing,
  lexCompare
) where

{-# NoImplicitPrelude #-}

import Prelude.Bool
import Prelude.Eq
import Prelude.Template
import Prelude.Default
import Prelude.Unit()

datatype Ordering :=
    LT
  | EQ
  | GT

derive Ordering instances (EqP, Eq)

instance Default Ordering where
  define default := EQ
  {-# inline default #-}
end-instance


class Eq 'a => Ord 'a where
  declare compare :: 'a -> 'a -> Ordering

  define compare x y :=
    cases
        (x == y) -> EQ
      | (x < y) -> LT
      | otherwise -> GT
    end

  declare (<) :: 'a -> 'a -> Bool
  declare (<=) :: 'a -> 'a -> Bool
  declare (>) :: 'a -> 'a -> Bool
  declare (>=) :: 'a -> 'a -> Bool
  declare min :: 'a -> 'a -> 'a
  declare max :: 'a -> 'a -> 'a

  define (<=) x y := (compare x y /= GT)
  define (<) x y := (compare x y == LT)
  define (>=) x y := (compare x y /= LT)
  define (>) x y := (compare x y == GT)

  define max x y := if (x <= y) then y else x
  define min x y := if (x <= y) then x else y

  test lt_alt_defs (a :: 'a) b :=
    ((a > b) <-> (b < a)) &&
    ((a >= b) <-> (b <= a)) &&
    ((a <= b) <-> ((a < b) || (a == b)))
```

```
58
59    test lt_irrefl (a :: 'a) := not (a < a)
60    test lt_trans (a :: 'a) b c := ((a < b) && (b < c)) --> (a < c)
61
62    {-# minimal compare | (<) #-}
63  end-class
64
65  instance Ord () where
66    define compare () () := EQ
67  end-instance
68
69  declare comparing :: Ord 'a => ('b -> 'a) -> 'b -> 'b -> Ordering
70  define comparing p x y := compare (p x) (p y)
71
72  declare lexCompare :: Ord 'a => 'a -> 'a -> Ordering -> Ordering
73  define lexCompare x y next :=
74    case (compare x y) of
75        EQ -> next
76      | other -> other
77    end
78  {-# inline lexCompare #-}
79
80
81  template DeriveInstanceOrdBody (_ :: ClassID, ty :: TypeID, _ :: [String]) := '''
82    {{ var constrs := ty.constructors;
83       var clauses := [] :: [String];
84       foreach var i in constrs.indices {
85         name-scope {
86           var (args1, constr1) := !ConstWithArgs(True, "x", constrs[i]);
87           var (args2, constr2) := !ConstWithArgs(True, "y", constrs[i]);
88           var cl := "compare " ++ constr1 ++ " " ++ constr2 ++ " := ";
89           if (args1.length > 0) {
90             var rhs := [] :: [String];
91             for var j := 1 to (args1.length - 1) {
92               var el := "lexCompare " ++ args1[j-1] ++ " " ++ args2[j-1];
93               rhs := insertAtEnd(el, rhs);
94             }
95             rhs := insertAtEnd("compare "++ args1[args1.length - 1] ++ " " ++ args2[args1.length - 1], rhs
96             cl := cl ++ !Intercalate(" $ ", rhs);
97           } else {
98             cl := cl ++ "EQ"
99           }
100          clauses := insertAtEnd(cl, clauses);
101
102          var constrWc := !ConstWithWildcardArgs(True, constrs[i]);
103          if (i < constrs.length - 1) {
104           clauses := insertAtEnd("compare " ++ constrWc ++ " _ := LT", clauses);
105           clauses := insertAtEnd("compare _ " ++ constrWc ++ " := GT", clauses);
106          }
107        }
108      }
109      var recDecl, indentRec := "";
110      if (ty.isRec) { recDecl := "rec "; indentRec := "    "; }
111    }}
112 {{= "define " ++ recDecl ++ !Intercalate("\n     "++indentRec++"| ", clauses) =}}
113 '''
114
115 register-derive-template Ord DeriveInstanceOrdBody
116
```

```
117 derive Ordering instances Ord
118 derive Bool instances Ord
119
120 end-module
```

1.27 Prelude/String.ad

Outline

Content

```
1  module Prelude.String (
2    String(..),
3    mkString
4  ) where
5
6  {-# NoImplicitPrelude #-}
7
8  import Prelude.List ()
9  import Prelude.Char
10 import Prelude.Literal
11
12 type String := [Char]
13
14 declare mkString :: BuiltInString -> String
15
16 instance LiteralString String where
17   define mkLiteralString := mkString
18   {-# inline mkLiteralString #-}
19 end-instance
20
21 end-module
```

1.28 Prelude/Template.ad

Outline

Content

```
1  -- auxiliary template functions
2  module Prelude.Template (
3    GetTypeVar,
4    GetTypeVars,
5    GetTypeVarsNumbered,
6    Intercalate,
7    Commafy,
8    BuildTuple,
9    BuildTupleLeft,
10   BuildTupleRight,
11   BuildInstanceHeader,
12   BuildInstance,
13   GenerateRecognisers,
14
15   IsSimpleEnumDatatype,
16
17   ConstWithArgs,
18   TypeWithArgs,
19   ClassWithArgs,
20   ConstWithWildcardArgs,
21
22   DeriveInstanceComputeSuperDefault,
23   DeriveInstance,
24   DerivingInstance
25 ) where
26
27 {-# NoImplicitPrelude #-}
28
29 -- generate the type variable with given number 'a, 'b, 'c, ...
30 template GetTypeVar (n :: Nat) :: String := {{
31   var cn := n % 26;
32   n := n / 26;
33   var result := "\'" ++ chr(cn + 97);
34   if (n == 0) {
35     return result;
36   } else {
37     return (result ++ n);
38   }
39 }}
40
41 -- generate a list of type variable of given length, 'a, 'b, ...
42 template GetTypeVars (n :: Nat) :: [String] := {{
43   var result := [] :: [String];
```

```
  for var y := 1 to n {
    result := insertAtEnd(!GetTypeVar(y-1), result);
  };
  return result;
}}

-- generate a list of type variable of given length, 'p1, 'p2, ...
template GetTypeVarsNumbered (p :: String, n :: Nat) :: [String] := {{
  var result := [] :: [String];
  for var y := 1 to n {
    var vn := "'" ++ p ++ y;
    result := insertAtEnd(vn, result);
  };
  return result;
}}


template Intercalate (sep :: String, s :: [String]) :: String := {{
  if (s.length == 0) {
    return ""
  } else {
    var result := s[0];
    for var i := 1 to (s.length - 1) {
      result := result ++ sep ++ s[i];
    };
    return result;
  }
}}

-- commafy an array of strings and return the resulting string
template Commafy (s :: [String]) :: String := {{
  return !Intercalate(", ", s)
}}

-- commafy an array and also add parenthesis around it, i.e. it creates a tuple
template BuildTuple (s :: [String]) :: String := {{
  if (s.length == 0) {
    error("tuples should contain at least one element");
    return "error";
  } else-if (s.length == 1) {
    return s[0];
  } else {
    return ("(" ++ !Commafy(s) ++ ")");
  }
}}

-- creates a left-leaning tuple, i.e something like (((a, b), c), d)
template BuildTupleLeft (s :: [String]) :: String := {{
  if (s.length == 0) {
    error("tuples should contain at least one element");
    return "error";
  } else-if (s.length == 1) {
    return s[0];
  } else {
    var res := s[0];
    for var i := 1 to (s.length - 1) {
      res := "(" ++ res ++ ", " ++ s[i] ++")";
    }
    return res;
```

```
103   }
104 }}
105
106
107 -- creates a right-leaning tuple, i.e something like (a, (b, (c, d)))
108 template BuildTupleRight (s :: [String]) :: String := {{
109   if (s.length == 0) {
110     error("tuples should contain at least one element");
111     return "error";
112   } else-if (s.length == 1) {
113     return s[0];
114   } else {
115     var res := s[s.length - 1];
116     for var i := (s.length - 2) downto 0 {
117       res := "(" ++ s[i] ++ ", " ++ res ++")";
118     }
119     return res;
120   }
121 }}
122
123
124 template IsSimpleEnumDatatype (ty :: TypeID) :: Bool := {{
125   if (not (ty.typeVariety == "datatype")) return False;
126
127   foreach var c in ty.constructors {
128     if (c.argsNo > 0) return False;
129   }
130   if (ty.constructors.length == 0) return False;
131
132   return True;
133 }}
134
135
136 -- | add all the arguments separated by space to a base.
137 template AddArgs (forceParens :: Bool, base :: String, args :: [String]) :: String := {{
138   var res := base;
139   foreach var arg in args {
140     res := res ++ " " ++ arg;
141   }
142   if ((args.length > 0) && forceParens) {
143     res := "("++res++")";
144   }
145   return res
146 }}
147
148 -- | add the appropriate number of wildcard arguments to a constant and return
149 -- the resulting string. If at least one argument is used and if the parameter
150 -- 'forceParens' is set, parenthesis are added around the result
151 template ConstWithWildcardArgs (forceParens :: Bool, c :: ConstID) :: String := {{
152   var args := replicate(c.argsNo, "_");
153   return (!AddArgs(forceParens, c, args));
154 }}
155
156
157 -- | add the appropriate number of arguments to a constant and return
158 -- the added arguments as well as the resulting string. The names of the
159 -- arguments are generated using the given prefix.
160 -- If at least one argument is used and if the parameter
161 -- 'forceParens' is set, parenthesis are added around the result
```

```
template ConstWithArgs (forceParens :: Bool, prefix :: String, c :: ConstID) :: ([String], String) := {{
  var args := generateNames(prefix, c.argsNo);
  return (args, !AddArgs(forceParens, c, args));
}}


-- | add the appropriate number of arguments to a type and return
-- the added arguments as well as the resulting string. The names of the
-- arguments are take from the definition of the type.
-- If at least one argument is used and if the parameter
-- 'forceParens' is set, parenthesis are added around the result
template TypeWithArgs (forceParens :: Bool, ty :: TypeID) :: ([String], String) := {{
  var args := ty.args;
  return (args, !AddArgs(forceParens, ty, args));
}}

-- | build recognisers for all constructors of a data-type
template GenerateRecognisers (ty :: TypeID) := '''
{{ var (_, tyFull) := !TypeWithArgs(False, ty) }}
{{% foreach var constr in ty.constructors %}}
{{ var name := "is" ++ constr;
   var constrWithWc := !ConstWithWildcardArgs(True, constr)}}
declare {{= name =}} :: {{=tyFull=}} -> Bool
define {{= name =}} {{= constrWithWc =}} := True
     | {{= name =}} _ := False

{{% end-foreach %}}
'''


-- | add the appropriate number of arguments to a type and return
-- the added arguments as well as the resulting string. The names of the
-- arguments are take from the definition of the class.
-- If at least one argument is used and if the parameter
-- 'forceParens' is set, parenthesis are added around the result
template ClassWithArgs (forceParens :: Bool, cl :: ClassID) :: ([String], String) := {{
  var args := cl.args;
  return (args, !AddArgs(forceParens, cl, args));
}}



-- | given a list of super-constraints and the main instance, build the instance-header string
template BuildInstanceHeader (super :: [String], inst :: String) := '''
{{ var superS := "";
   if (super.length > 0) {
     superS := !BuildTuple(super) ++ " => ";
   }
}}
instance {{= superS =}}{{= inst =}} where
'''

-- | call 'BuildInstanceHeader' followed by a body and the end-instance
template BuildInstance (super :: [String], inst :: String, body :: String) := '''
{{! BuildInstanceHeader(super, inst) !}}
  {{= body =}}
end-instance
'''

```

```
221 -- | Auxliary template for building instances of type-classes. Given a type-class with
222 -- exactly one argument and a type-ID to use for this argument, it looks up the arguments
223 -- of the type and uses each found argument with the main type-class as a class constraint of this instan
224 -- This is useful for defining instances of classes like Eq automatically.
225 -- It returns the super-constraints, the instance as well as the full type and the used type args
226 template DeriveInstanceComputeSuperDefault (cl :: ClassID, ty :: TypeID) :: ([String], String, [String])
227   if (not (cl.argsNo == 1)) error("only type-classes with 1 argument supported");
228
229   var (tyArgs, tyFull) := !TypeWithArgs(True, ty);
230   var inst := cl ++ " " ++ tyFull;
231
232   var super := [] :: [String];
233   foreach var t in tyArgs {
234     super := insertAtEnd(cl ++ " " ++ t, super);
235   }
236
237   return (super, inst, tyArgs);
238 }}
239
240 -- | the build-in template used for "derive" statements. It gets a template to compute the class constrai
241 -- the instance and a template to compute the body. If for a type-class no other template is registered,
242 -- 'DeriveInstanceComputeSuperDefault' is used automatically
243 template DeriveInstance (cl :: ClassID, ty :: TypeID,
244    superInst :: TemplateID(ClassID -> TypeID -> ([String], String, [String])),
245    body :: TemplateID(ClassID -> TypeID -> [String] -> Text)) := '''
246 {{ var (super, inst, tyArgs) := !superInst(cl, ty); }}
247 {{! DerivingInstance (cl, ty, tyArgs, super, inst, body) !}}
248 '''
249
250 -- | the build-in template used for "derivig" statements.
251 template DerivingInstance (cl :: ClassID, ty :: TypeID, tyArgs :: [String], super :: [String], inst :: St
252 {{! BuildInstanceHeader(super, inst) !}}
253   {{= !body(cl, ty, tyArgs) =}}
254 end-instance
255 '''
256
257
258 end-module
```

1.29 Prelude/Text.ad

Outline

Content

```
module Prelude.Text (
  Text,
  pack,
  unpack
) where

{-# NoImplicitPrelude #-}

import Prelude.List ()
import Prelude.Eq
import Prelude.BasicClasses
import Prelude.Ord
import Prelude.String
import Prelude.Function
import Prelude.Literal
import Prelude.Bool

datatype Text := Text String

declare pack :: String -> Text
define pack := Text

declare unpack :: Text -> String
define unpack (Text s) := s

instance LiteralString Text where
  define mkLiteralString := pack . mkLiteralString
  {-# inline mkLiteralString #-}
end-instance

derive Text instances (EqP, Eq, Ord)

instance Semigroup Text where
  define (<>) (Text s1) (Text s2) := Text (s1 <> s2)
end-instance

instance Monoid Text where
  define mempty := pack ""
end-instance



end-module
```

1.30 Prelude/Tuple.ad

Outline

Content

```
1  module Prelude.Tuple (
2    fst,
3    snd,
4    uncurry,
5    curry,
6
7    Tuple2(..),
8    Tuple3(..),
9    Tuple4(..),
10   Tuple5(..),
11   Tuple6(..),
12   Tuple7(..),
13   Tuple8(..),
14   Tuple9(..)
15 ) where
16
17 {-# NoImplicitPrelude #-}
18
19 import Prelude.Bool
20 import Prelude.Eq
21 import Prelude.Ord
22 import Prelude.Function
23 import Prelude.Template
24 import Prelude.Functor
25
26 declare fst :: ('a, 'b) -> 'a
27 define fst (x, _) := x
28
29 declare snd :: ('a, 'b) -> 'b
30 define snd (_, y) := y
31
32 declare curry :: (('a, 'b) -> 'c) -> ('a -> 'b -> 'c)
33 define curry f x y :=  f (x, y)
34
35 declare uncurry :: ('a -> 'b -> 'c) -> (('a, 'b) -> 'c)
36 define uncurry f (x, y) := f x y
37
38 template BuildTupleAbbrev (n :: Nat) :=
39   '''type Tuple{{= n =}} \'a := {{=!BuildTuple(replicate(n, "'a"))=}}'''
40
41
42 template BuildFunctorTupleInstance (n :: Nat) := '''
43 {{ name-scope {
44      var xs := generateNames("x", n)
45    }
46    var f_xs := [] :: [String];
```

```
    foreach var x in xs {
      f_xs := insertAtEnd("f " ++ x, f_xs);
    }
}}
{{! BuildInstanceHeader([] :: [String], "Functor Tuple" ++ n) !}}
  define fmap f {{= !BuildTuple(xs) =}} := {{= !BuildTuple(f_xs) =}}
end-instance
'''

generate-code '''
  {{% for var i := 2 to 9 %}}
  {{! BuildTupleAbbrev(i) !}}

  {{! BuildFunctorTupleInstance(i) !}}

  {{% end-for %}}
'''

template BuildTupleInstanceVars (className :: String, n :: Nat) :: ([String], [String], [String], String)
{{ name-scope {
     var tyVars := !GetTypeVars(n);
     var xs := generateNames("x", n);
     var ys := generateNames("y", n);

     var super := [] :: [String];
     foreach var tyV in tyVars {
       super := insertAtEnd(className ++ " " ++ tyV, super);
     }

     var inst := className ++ " " ++ !BuildTuple(tyVars);
   }
   return (xs, ys, super, inst);
}}

template BuildEqPTupleInstance (n :: Nat) := '''
{{ var (xs, ys, super, inst) := !BuildTupleInstanceVars("EqP", n);
   var comps := [] :: [String];
   foreach var xy in zip(xs, ys) {
     var el := "(" ++ #1(xy) ++ " === " ++ #2(xy) ++ ")";
     comps := insertAtEnd(el, comps);
   }
}}
{{! BuildInstanceHeader(super, inst) !}}
  define non-exec (===) {{= !BuildTuple(xs) =}} {{= !BuildTuple(ys) =}} :=
    {{= !Intercalate(" /\\\n", comps) =}}
end-instance
'''

template BuildEqTupleInstance (n :: Nat) := '''
{{ var (xs, ys, super, inst) := !BuildTupleInstanceVars("Eq", n);
   var comps := [] :: [String];
   foreach var xy in zip(xs, ys) {
     var el := "(" ++ #1(xy) ++ " == " ++ #2(xy) ++ ")";
     comps := insertAtEnd(el, comps);
   }
}}
{{! BuildInstanceHeader(super, inst) !}}
  define (==) {{= !BuildTuple(xs) =}} {{= !BuildTuple(ys) =}} :=
    {{= !Intercalate(" &&\n", comps) =}}
```

```
end-instance
'''

template BuildOrdTupleInstance (n :: Nat) := '''
{{ var (xs, ys, super, inst) := !BuildTupleInstanceVars("Ord", n);
   var comps := [] :: [String];
   for var i := 0 to (n - 2) {
     var el := "lexCompare " ++ xs[i] ++ " " ++ ys[i];
     comps := insertAtEnd(el, comps);
   }
}}
{{! BuildInstanceHeader(super, inst) !}}
  define compare {{= !BuildTuple(xs) =}} {{= !BuildTuple(ys) =}} :=
    {{= !Intercalate(" $\n", comps) =}} $
    compare {{=xs[n-1]=}} {{=ys[n-1]=}}
end-instance
'''

generate-code '''
  {{% for var i := 2 to 9 %}}
  {{! BuildEqPTupleInstance(i) !}}

  {{! BuildEqTupleInstance(i) !}}

  {{! BuildOrdTupleInstance(i) !}}

  {{% end-for %}}
'''
end-module
```

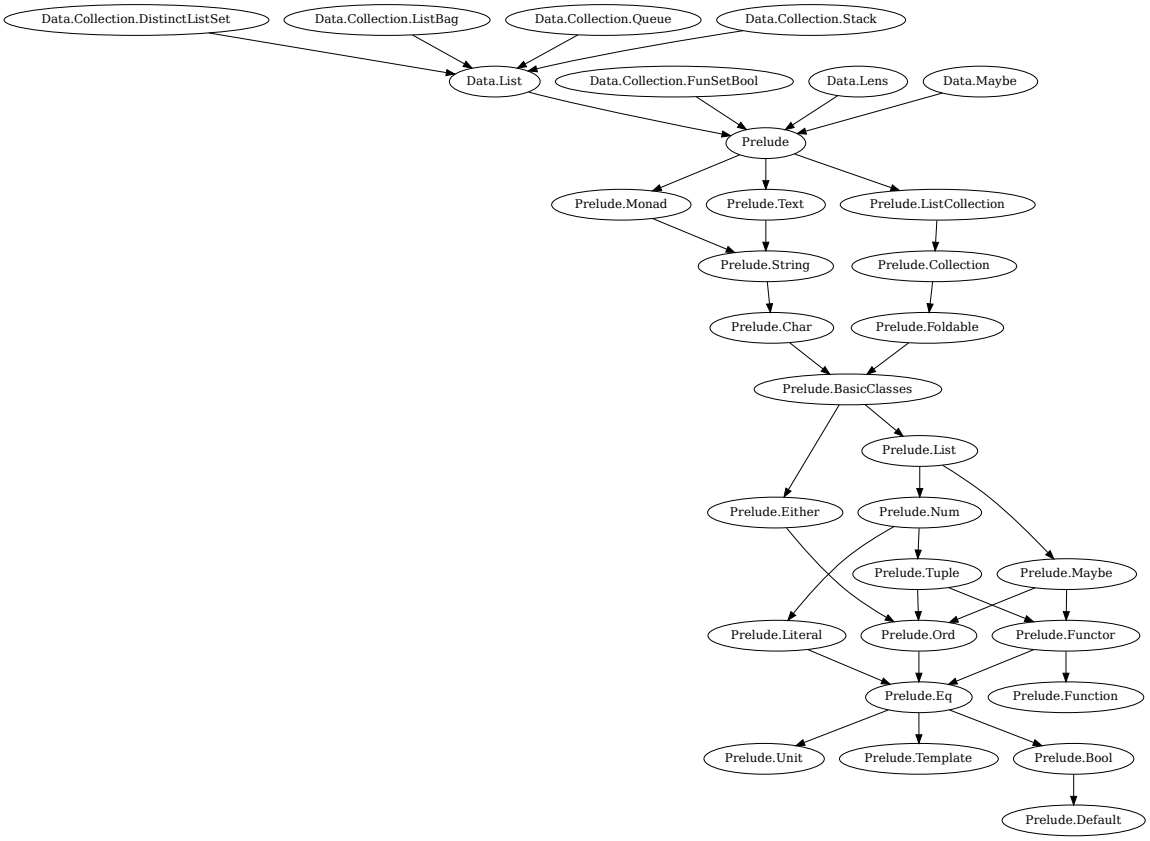
1.31 Prelude/Unit.ad

Outline

Content

```
module Prelude.Unit (
  Unit(..),
  Itself(..)
) where

{-# NoImplicitPrelude #-}

datatype Unit := Unit


datatype Itself 'a := Itself

end-module
```

2 Modules



- Data.Collection.DistinctListSet
  - Data/Collection/DistinctListSet.ad . . . . . lines 1 - 96
- Data.Collection.FunSetBool
  - Data/Collection/FunSetBool.ad . . . . . lines 1 - 52
- Data.Collection.ListBag
  - Data/Collection/ListBag.ad . . . . . lines 1 - 155
- Data.Collection.Queue
  - Data/Collection/Queue.ad . . . . . lines 1 - 91
- Data.Collection.Stack
  - Data/Collection/Stack.ad . . . . . lines 1 - 82
- Data.Lens
  - Data/Lens.ad . . . . . lines 2 - 259
- Data.List
  - Data/List.ad . . . . . lines 1 - 82
- Data.Maybe
  - Data/Maybe.ad . . . . . lines 1 - 30

- Prelude.Text
  - Prelude/Text.ad . . . . . lines 1 - 43
- Prelude.Tuple
  - Prelude/Tuple.ad . . . . . .lines 1 - 134
- Prelude.Unit
  - Prelude/Unit.ad . . . . . lines 1 - 13

3 Indices

3.1 Classes

3.1.1 B

- `Bag`
  - `Prelude.Bag` (alias for `Prelude.Collection.Bag`)
  - `Prelude.Collection.Bag` . . . . . Prelude/Collection.ad, lines 205, 206
    - arguments: `'c`, `'e`
    - super-classes:
      - `BagLike 'c 'e`
      - `CollectionWithDelete 'c 'e`
      - `CollectionWithEmptynessTest 'c 'e`
    - functional-dependencies:
      - `'c -> 'e`
    - instances:
      - `Eq 'e => Bag (Data.Collection.ListBag.ListBag 'e) 'e` . . . . . Data/Collection/ListBag.ad, lines 149, 150
- `BagLike`
  - `Prelude.BagLike` (alias for `Prelude.Collection.BagLike`)
  - `Prelude.Collection.BagLike` . . . . . Prelude/Collection.ad, lines 195 - 197
    - arguments: `'c`, `'e`
    - super-classes:
      - `CollectionWithMembershipTest 'c 'e`
    - functional-dependencies:
      - `'c -> 'e`
    - instances:
      - `BagLike (Data.Collection.ListBag.ListBag 'e) 'e` . . . . . Data/Collection/ListBag.ad, lines 146, 147
- `Bounded`
  - `Prelude.Bounded` (alias for `Prelude.BasicClasses.Bounded`)
  - `Prelude.BasicClasses.Bounded` . . . . . Prelude/BasicClasses.ad, lines 76 - 86
    - arguments: `'a`
    - super-classes:
      - `Ord 'a`
    - functional-dependencies: -
    - instances:
      - `Bounded Bool` . . . . . Prelude/BasicClasses.ad, line 278
      - `Bounded Ordering` . . . . . Prelude/BasicClasses.ad, line 277

3.1.2 C

- `Collection`
  - `Prelude.Collection` (alias for `Prelude.Collection.Collection`)
  - `Prelude.Collection.Collection` . . . . . Prelude/Collection.ad, lines 45 - 63
    - arguments: `'c`, `'e`
    - super-classes:
      - `Monoid 'c`
    - functional-dependencies:
      - `'c -> 'e`
    - instances:
      - `Eq 'e => Collection (Data.Collection.DistinctListSet.DistinctListSet 'e) 'e` . . . . . .Data/Collection/DistinctListSet.ad, lines 18 - 27
      - `Eq 'e => Collection (Data.Collection.FunSetBool.FunSetBool 'e) 'e` . . . . . Data/Collection/FunSetBool.ad, lines 17 - 27
      - `Collection (Data.Collection.ListBag.ListBag 'e) 'e` . . . . . .Data/Collection/ListBag.ad, lines 30 - 41
      - `Collection (Data.Collection.Queue.Queue 'e) 'e` . . . . . .Data/Collection/Queue.ad, lines 60 - 69
      - `Collection (Data.Collection.Stack.Stack 'e) 'e` Data/Collection/Stack.ad, lines 57 - 66
      - `Collection ['e] 'e` . . . . . Prelude/ListCollection.ad, lines 13 - 24
- `CollectionWithDelete`
  - `Prelude.CollectionWithDelete` (alias for `Prelude.Collection.CollectionWithDelete`)
  - `Prelude.Collection.CollectionWithDelete` . . . . . Prelude/Collection.ad, lines 107 - 140
    - arguments: `'c`, `'e`
    - super-classes:
      - `Eq 'e`
      - `CollectionWithMembershipTest 'c 'e`
    - functional-dependencies:
      - `'c -> 'e`
    - instances:
      - `Eq 'e => CollectionWithDelete (Data.Collection.DistinctListSet.DistinctListSet 'e) 'e` . . . . . .Data/Collection/DistinctListSet.ad, lines 52 - 70
      - `Eq 'e => CollectionWithDelete (Data.Collection.FunSetBool.FunSetBool 'e) 'e` . . . . . Data/Collection/FunSetBool.ad, lines 34 - 47
      - `Eq 'e => CollectionWithDelete (Data.Collection.ListBag.ListBag 'e) 'e` . . . . . .Data/Collection/ListBag.ad, lines 70 - 90
      - `Eq 'e => CollectionWithDelete ['e] 'e` . . . . . Prelude/ListCollection.ad, lines 50 - 65
- `CollectionWithEmptynessTest`
  - `Prelude.CollectionWithEmptynessTest` (alias for `Prelude.Collection.CollectionWithEmptynessTest`)
  - `Prelude.Collection.CollectionWithEmptynessTest` . . . . . Prelude/Collection.ad, lines 79 - 83
    - arguments: `'c`, `'e`
    - super-classes:
      - `Collection 'c 'e`
    - functional-dependencies:
      - `'c -> 'e`

3.1.3 D

- `Default`
  - `Prelude.Default` (alias for `Prelude.Default.Default`)
  - `Prelude.Default.Default` . . . . . Prelude/Default.ad, lines 8 - 16
    - arguments: `'a`
    - super-classes: -
    - functional-dependencies: -
    - instances:
      - `Default Bool` . . . . . Prelude/Bool.ad, lines 25, 26
      - `Default Char` . . . . . Prelude/Char.ad, lines 24 - 27
      - `Default (Maybe 'a)` . . . . . Prelude/Maybe.ad, lines 24 - 26
      - `Default Natural` . . . . . Prelude/Num.ad, lines 23 - 26
      - `Default Ordering` . . . . . Prelude/Ord.ad, lines 23 - 26
      - `Default ['a]` . . . . . Prelude/List.ad, lines 55 - 57
- `Dot`
  - `Prelude.Dot` (alias for `Prelude.Function.Dot`)
  - `Prelude.Function.Dot` . . . . . Prelude/Function.ad, lines 11 - 13
    - arguments: `'a`, `'b`, `'c`
    - super-classes: -
    - functional-dependencies:
      - `'a, 'b -> 'c`
    - instances:
      - `Dot ('b -> 'c) ('a -> 'b2) ('a -> 'c) | 'b2 -> 'b` Prelude/Function.ad, lines 15 - 18
      - `Dot (Data.Lens.Getter 'a 'b) (Data.Lens.Getter 'b2 'c) (Data.Lens.Getter 'a 'c) | 'b2 -> 'b` . . . . . Data/Lens.ad, lines 120 - 123
      - `Dot (Data.Lens.Getter 'a 'b) (Data.Lens.Lens 'b2 'x 'c 'y) (Data.Lens.Getter 'a 'c) | 'b2 -> 'b` . . . . . Data/Lens.ad, lines 125 - 128
      - `Dot (Data.Lens.Lens 'a 'x 'b 'y) (Data.Lens.Getter 'b2 'c) (Data.Lens.Getter 'a 'c) | 'b2 -> 'b` . . . . . Data/Lens.ad, lines 130 - 133
      - `Dot (Data.Lens.Lens 's1 't1 'a1 'b1) (Data.Lens.Lens 's2 't2 'a2 'b2) (Data.Lens.Lens 's1 't1 'a2 'b2) | 'a1 -> 's2, 'b1 -> 't2` . . . . . Data/Lens.ad, lines 149 - 152
      - `Dot (Data.Lens.Lens 's1 't1 'a1 'b1) (Data.Lens.Setter 's2 't2 'b2) (Data.Lens.Setter 's1 't1 'b2) | 'a1 -> 's2, 'b1 -> 't2` . . . . . Data/Lens.ad, lines 144 - 147

3.1.4 E

- `Elem1`
  - `Data.Lens.Elem1` . . . . . Data/Lens.ad, line 191
    - arguments: `'s`, `'t`, `'a`, `'b`
    - super-classes: -
    - functional-dependencies:
      - `'s -> 'a`
      - `'s, 'b -> 't`
    - instances:

- `Elem9`
  - `Data.Lens.Elem9` . . . . . Data/Lens.ad, line 191
    - arguments: `'s`, `'t`, `'a`, `'b`
    - super-classes: -
    - functional-dependencies:
      - `'s -> 'a`
      - `'s, 'b -> 't`
    - instances:
      - `Data.Lens.Elem9 ('a1, 'a2, 'a3, 'a4, 'a5, 'a6, 'a7, 'a8, 'a9) 'x 'z 'y | 'x -> ('a1, 'a2, 'a3, 'a4, 'a5, 'a6, 'a7, 'a8, 'y), 'z -> 'a9` . . . Data/Lens.ad, line 191
- `Enum`
  - `Prelude.Enum` (alias for `Prelude.BasicClasses.Enum`)
  - `Prelude.BasicClasses.Enum` . . . . . Prelude/BasicClasses.ad, lines 25 - 65
    - arguments: `'a`
    - super-classes:
      - `Ord 'a`
    - functional-dependencies: -
    - instances:
      - `Enum Bool` . . . . . Prelude/BasicClasses.ad, line 278
      - `Enum Char` . . . . . Prelude/Char.ad, lines 43 - 48
      - `Enum Natural` . . . . . Prelude/BasicClasses.ad, lines 67 - 72
      - `Enum Ordering` . . . . . Prelude/BasicClasses.ad, line 277
- `Eq`
  - `Prelude.Eq` (alias for `Prelude.Eq.Eq`)
  - `Prelude.Eq.Eq` . . . . . Prelude/Eq.ad, lines 90 - 104
    - arguments: `'a`
    - super-classes:
      - `EqP 'a`
    - functional-dependencies: -
    - instances:
      - `(Eq 'a, Eq 'b) => Eq ('a, 'b)` . . . . . Prelude/Tuple.ad, line 124
      - `(Eq 'a, Eq 'b, Eq 'c) => Eq ('a, 'b, 'c)` . . . . . Prelude/Tuple.ad, line 124
      - `(Eq 'a, Eq 'b, Eq 'c, Eq 'd) => Eq ('a, 'b, 'c, 'd)` . . . . Prelude/Tuple.ad, line 124
      - `(Eq 'a, Eq 'b, Eq 'c, Eq 'd, Eq 'e) => Eq ('a, 'b, 'c, 'd, 'e)` . . . . . Prelude/Tuple.ad, line 124
      - `(Eq 'a, Eq 'b, Eq 'c, Eq 'd, Eq 'e, Eq 'f) => Eq ('a, 'b, 'c, 'd, 'e, 'f)` . . . . . Prelude/Tuple.ad, line 124
      - `(Eq 'a, Eq 'b, Eq 'c, Eq 'd, Eq 'e, Eq 'f, Eq 'g) => Eq ('a, 'b, 'c, 'd, 'e, 'f, 'g)` . . . . . Prelude/Tuple.ad, line 124
      - `(Eq 'a, Eq 'b, Eq 'c, Eq 'd, Eq 'e, Eq 'f, Eq 'g, Eq 'h) => Eq ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h)` . . . . . Prelude/Tuple.ad, line 124
      - `(Eq 'a, Eq 'b, Eq 'c, Eq 'd, Eq 'e, Eq 'f, Eq 'g, Eq 'h, Eq 'i) => Eq ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i)` . . . . . Prelude/Tuple.ad, line 124
      - `Eq ()` . . . . . Prelude/Eq.ad, lines 138 - 140
      - `Eq Bool` . . . . . Prelude/Eq.ad, line 146

3.1.5 F

    * arguments: `'c`, `'e`
    * super-classes:
        · `Bag 'c 'e`
        · `FiniteCollection 'c 'e`
    * functional-dependencies:
        · `'c -> 'e`
    * instances:
        · `Eq 'e => FiniteBag (Data.Collection.ListBag.ListBag 'e) 'e` . . . . . Data/Collection/ListBag.ad, lines 152, 153

- `FiniteCollection`
  - `Prelude.FiniteCollection` (alias for `Prelude.Collection.FiniteCollection`)
  - `Prelude.Collection.FiniteCollection` . . . . . Prelude/Collection.ad, lines 144 - 157
    * arguments: `'c`, `'e`
    * super-classes:
        · `CollectionWithMembershipTest 'c 'e`
        · `Foldable 'c 'e`
        · `CollectionWithEmptynessTest 'c 'e`
    * functional-dependencies:
        · `'c -> 'e`
    * instances:
        · `Eq 'e => FiniteCollection (Data.Collection.DistinctListSet.DistinctListSet 'e) 'e` . . . . . Data/Collection/DistinctListSet.ad, lines 84, 85
        · `Eq 'e => FiniteCollection (Data.Collection.ListBag.ListBag 'e) 'e` . . . . . Data/Collection/ListBag.ad, lines 56, 57
        · `Eq 'e => FiniteCollection (Data.Collection.Queue.Queue 'e) 'e` . . . . . Data/Collection/Queue.ad, lines 88, 89
        · `Eq 'e => FiniteCollection (Data.Collection.Stack.Stack 'e) 'e` . . . . . Data/Collection/Stack.ad, lines 79, 80
        · `FiniteCollection ['e] 'e` . . . . . Prelude/ListCollection.ad, lines 67, 68

- `FiniteSet`
  - `Prelude.FiniteSet` (alias for `Prelude.Collection.FiniteSet`)
  - `Prelude.Collection.FiniteSet` . . . . . Prelude/Collection.ad, lines 202, 203
    * arguments: `'c`, `'e`
    * super-classes:
        · `Set 'c 'e`
        · `FiniteCollection 'c 'e`
    * functional-dependencies:
        · `'c -> 'e`
    * instances:
        · `Eq 'e => FiniteSet (Data.Collection.DistinctListSet.DistinctListSet 'e) 'e` . . . . . Data/Collection/DistinctListSet.ad, lines 93, 94

- `Foldable`
  - `Prelude.Foldable` (alias for `Prelude.Foldable.Foldable`)
  - `Prelude.Foldable.Foldable` . . . . . Prelude/Foldable.ad, lines 18 - 50
    * arguments: `'t`, `'e`

    * super-classes: -
    * functional-dependencies:
        · `'t -> 'e`
    * instances:
        · `Foldable ('a, 'a) 'a` . . . . . Prelude/Foldable.ad, lines 107 - 114
        · `Foldable ('a, 'a, 'a) 'a` . . . . . Prelude/Foldable.ad, lines 116 - 123
        · `Foldable ('a, 'a, 'a, 'a) 'a` . . . . . Prelude/Foldable.ad, lines 125 - 132
        · `Foldable ('a, 'a, 'a, 'a, 'a) 'a` . . . . . Prelude/Foldable.ad, lines 134 - 141
        · `Foldable ('a, 'a, 'a, 'a, 'a, 'a) 'a` . . . . . Prelude/Foldable.ad, lines 143 - 150
        · `Foldable ('a, 'a, 'a, 'a, 'a, 'a, 'a) 'a` . . . . . Prelude/Foldable.ad, lines 152 - 160
        · `Foldable ('a, 'a, 'a, 'a, 'a, 'a, 'a, 'a) 'a` . . . . Prelude/Foldable.ad, lines 162 - 169
        · `Foldable (Data.Collection.DistinctListSet.DistinctListSet 'e) 'e` . . . . . Data/Collection/DistinctListSet.ad, lines 34 - 43
        · `Foldable (Data.Collection.ListBag.ListBag 'e) 'e` . . . . . Data/Collection/ListBag.ad, lines 43 - 45
        · `Foldable (Data.Collection.Queue.Queue 'e) 'e` . Data/Collection/Queue.ad, lines 71 - 80
        · `Foldable (Data.Collection.Stack.Stack 'e) 'e` . Data/Collection/Stack.ad, lines 68 - 71
        · `Foldable (Maybe 'a) 'a` . . . . . Prelude/Foldable.ad, lines 83 - 104
        · `Foldable ['a] 'a` . . . . . Prelude/Foldable.ad, lines 52 - 80

- `Functor`
  – `Prelude.Functor` (alias for `Prelude.Functor.Functor`)
  – `Prelude.Functor.Functor` . . . . . Prelude/Functor.ad, lines 12 - 17
    * arguments: `'F`
    * super-classes: -
    * functional-dependencies: -
    * instances:
        · `Functor Data.Collection.ListBag.ListBag` . . . . Data/Collection/ListBag.ad, lines 13 - 15
        · `Functor Data.Collection.Queue.Queue` . . . . . Data/Collection/Queue.ad, lines 31 - 33
        · `Functor Data.Collection.Stack.Stack` . . . . . Data/Collection/Stack.ad, lines 28 - 30
        · `Functor List` . . . . . Prelude/List.ad, lines 64 - 67
        · `Functor Maybe` . . . . . Prelude/Maybe.ad, lines 32 - 35
        · `Functor Tuple2` . . . . . Prelude/Tuple.ad, line 56
        · `Functor Tuple3` . . . . . Prelude/Tuple.ad, line 56
        · `Functor Tuple4` . . . . . Prelude/Tuple.ad, line 56
        · `Functor Tuple5` . . . . . Prelude/Tuple.ad, line 56
        · `Functor Tuple6` . . . . . Prelude/Tuple.ad, line 56
        · `Functor Tuple7` . . . . . Prelude/Tuple.ad, line 56
        · `Functor Tuple8` . . . . . Prelude/Tuple.ad, line 56
        · `Functor Tuple9` . . . . . Prelude/Tuple.ad, line 56

3.1.6 I

- `IsGetter`
  - `Data.Lens.IsGetter` . . . . . Data/Lens.ad, lines 68 - 74
    - arguments: `'v`, `'a`, `'b`
    - super-classes: -
    - functional-dependencies:
      - `'v -> 'a, 'b`
    - instances:
      - `Data.Lens.IsGetter (Data.Lens.Getter 'a 'b) 'a 'b` . . . . . Data/Lens.ad, lines 87 - 90
      - `Data.Lens.IsGetter (Data.Lens.Lens 's 't 'a 'b) 's 'a` . . Data/Lens.ad, lines 92 - 95
- `IsSetter`
  - `Data.Lens.IsSetter` . . . . . Data/Lens.ad, lines 79 - 85
    - arguments: `'v`, `'s`, `'t`, `'b`
    - super-classes: -
    - functional-dependencies:
      - `'v -> 's, 't, 'b`
    - instances:
      - `Data.Lens.IsSetter (Data.Lens.Lens 's 't 'a 'b) 's 't 'b` . . . . . Data/Lens.ad, lines 107 - 110
      - `Data.Lens.IsSetter (Data.Lens.Setter 's 't 'b) 's 't 'b` . . . . . Data/Lens.ad, lines 102 - 105

3.1.7 L

- `LiteralChar`
  - `Prelude.LiteralChar` (alias for `Prelude.Literal.LiteralChar`)
  - `Prelude.Literal.LiteralChar` . . . . . Prelude/Literal.ad, lines 55 - 57
    - arguments: `'a`
    - super-classes:
      - `Eq 'a`
    - functional-dependencies: -
    - instances:
      - `LiteralChar BuiltInChar` . . . . . Prelude/Literal.ad, lines 70 - 73
      - `LiteralChar Char` . . . . . Prelude/Char.ad, lines 37 - 40
- `LiteralDecimal`
  - `Prelude.LiteralDecimal` (alias for `Prelude.Literal.LiteralDecimal`)
  - `Prelude.Literal.LiteralDecimal` . . . . . Prelude/Literal.ad, lines 122 - 124
    - arguments: `'a`
    - super-classes:
      - `LiteralNum 'a`
    - functional-dependencies: -
    - instances:
      - `LiteralDecimal BuiltInDecimal` . . . . . Prelude/Literal.ad, lines 132 - 135
- `LiteralNum`
  - `Prelude.LiteralNum` (alias for `Prelude.Literal.LiteralNum`)

3.1.8 M

    * arguments: `'a`
    * super-classes:
        · `Semigroup 'a`
    * functional-dependencies: -
    * instances:
        · `Monoid ()` . . . . . Prelude/BasicClasses.ad, lines 134 - 137
        · `Eq 'e => Monoid (Data.Collection.DistinctListSet.DistinctListSet 'e)` . . . . . Data/Collection/DistinctListSet.ad, lines 14 - 16
        · `Eq 'e => Monoid (Data.Collection.FunSetBool.FunSetBool 'e)` . . . . . Data/Collection/FunSetBool.ad, lines 13 - 15
        · `Monoid (Data.Collection.ListBag.ListBag 'a)` . Data/Collection/ListBag.ad, lines 21 - 24
        · `Monoid (Data.Collection.Queue.Queue 'a)` . . . . . Data/Collection/Queue.ad, lines 51 - 54
        · `Monoid (Data.Collection.Stack.Stack 'a)` . . . . . Data/Collection/Stack.ad, lines 48 - 51
        · `Monoid Ordering` . . . . . Prelude/BasicClasses.ad, lines 155 - 157
        · `Monoid Text` . . . . . Prelude/Text.ad, lines 37 - 39
        · `Monoid ['a]` . . . . . Prelude/BasicClasses.ad, lines 144 - 147

3.1.9 N

- `NumDiv`
  - `Prelude.NumDiv` (alias for `Prelude.Num.NumDiv`)
  - `Prelude.Num.NumDiv` . . . . . Prelude/Num.ad, lines 42 - 44
    * arguments: `'a`
    * super-classes: -
    * functional-dependencies: -
    * instances:
        · `NumDiv Natural` . . . . . Prelude/Num.ad, lines 88 - 91
- `NumDivMod`
  - `Prelude.NumDivMod` (alias for `Prelude.Num.NumDivMod`)
  - `Prelude.Num.NumDivMod` . . . . . Prelude/Num.ad, lines 46 - 54
    * arguments: `'a`
    * super-classes: -
    * functional-dependencies: -
    * instances:
        · `NumDivMod Natural` . . . . . Prelude/Num.ad, lines 77 - 86
- `NumMinus`
  - `Prelude.NumMinus` (alias for `Prelude.Num.NumMinus`)
  - `Prelude.Num.NumMinus` . . . . . Prelude/Num.ad, lines 34 - 36
    * arguments: `'a`
    * super-classes: -
    * functional-dependencies: -
    * instances:
        · `NumMinus Natural` . . . . . Prelude/Num.ad, lines 66 - 70
- `NumMult`
  - `Prelude.NumMult` (alias for `Prelude.Num.NumMult`)

3.1.10 O

3.1.11 S

- `Semigroup`
  - `Prelude.Semigroup` (alias for `Prelude.BasicClasses.Semigroup`)
  - `Prelude.BasicClasses.Semigroup` . . . . . Prelude/BasicClasses.ad, lines 112 - 114
    - arguments: `'a`
    - super-classes: -
    - functional-dependencies: -
    - instances:
      - `Semigroup ()` . . . . . Prelude/BasicClasses.ad, lines 129 - 132
      - `Eq 'e => Semigroup (Data.Collection.DistinctListSet.DistinctListSet 'e)` . . . . . Data/Collection/DistinctListSet.ad, lines 9 - 12
      - `Eq 'e => Semigroup (Data.Collection.FunSetBool.FunSetBool 'e)` . . . . . Data/Collection/FunSetBool.ad, lines 8 - 11
      - `Semigroup (Data.Collection.ListBag.ListBag 'a)` . . . . . Data/Collection/ListBag.ad, lines 17 - 19
      - `Semigroup (Data.Collection.Queue.Queue 'a)` . . Data/Collection/Queue.ad, lines 47 - 49
      - `Semigroup (Data.Collection.Stack.Stack 'a)` . . Data/Collection/Stack.ad, lines 44 - 46
      - `Semigroup Ordering` . . . . . Prelude/BasicClasses.ad, lines 149 - 153
      - `Semigroup Text` . . . . . Prelude/Text.ad, lines 33 - 35
      - `Semigroup ['a]` . . . . . Prelude/BasicClasses.ad, lines 139 - 142
- `Set`
  - `Prelude.Set` (alias for `Prelude.Collection.Set`)
  - `Prelude.Collection.Set` . . . . . Prelude/Collection.ad, lines 199, 200
    - arguments: `'c`, `'e`
    - super-classes:
      - `SetLike 'c 'e`
      - `CollectionWithDelete 'c 'e`
      - `CollectionWithEmptynessTest 'c 'e`
    - functional-dependencies:
      - `'c -> 'e`
    - instances:
      - `Eq 'e => Set (Data.Collection.DistinctListSet.DistinctListSet 'e) 'e` . . . . . Data/Collection/DistinctListSet.ad, lines 90, 91
- `SetLike`
  - `Prelude.SetLike` (alias for `Prelude.Collection.SetLike`)
  - `Prelude.Collection.SetLike` . . . . . Prelude/Collection.ad, lines 179 - 181
    - arguments: `'c`, `'e`
    - super-classes:
      - `CollectionWithMembershipTest 'c 'e`
    - functional-dependencies:
      - `'c -> 'e`

    * instances:
        · `Eq 'e => SetLike (Data.Collection.DistinctListSet.DistinctListSet 'e) 'e` . . . . . Data/Collection/DistinctListSet.ad, lines 87, 88
        · `Eq 'e => SetLike (Data.Collection.FunSetBool.FunSetBool 'e) 'e` . . . . . .Data/Collection/FunSetBool.ad, lines 49, 50

3.2 Constants

3.2.1 (

- `($)`
  - `Prelude.($)` (alias for `Prelude.Function.($)`)
  - `Prelude.Function.($)` . . . . . .Prelude/Function.ad, line 31
    * constant
    * type `('a -> 'b) -> 'a -> 'b`
    * executable
- `(&&)`
  - `Prelude.(&&)` (alias for `Prelude.Bool.(&&)`)
  - `Prelude.Bool.(&&)` . . . . . .Prelude/Bool.ad, line 35
    * constant
    * type `Bool -> Bool -> Bool`
    * executable
- `(*)`
  - `Prelude.(*)` (alias for `Prelude.Num.(*)`)
  - `Prelude.Num.(*)` . . . . . .Prelude/Num.ad, line 39
    * constant of type-class `NumMult`
    * type `'a -> 'a -> 'a`
    * executable
- `(+)`
  - `Prelude.(+)` (alias for `Prelude.Num.(+)`)
  - `Prelude.Num.(+)` . . . . . .Prelude/Num.ad, line 31
    * constant of type-class `NumPlus`
    * type `'a -> 'a -> 'a`
    * executable
- `(++)`
  - `Data.List.(++)` (alias for `Prelude.List.(++)`)
  - `Prelude.(++)` (alias for `Prelude.List.(++)`)
  - `Prelude.List.(++)` . . . . . Prelude/List.ad, line 70
    * constant
    * type `['a] -> ['a] -> ['a]`
    * executable
- `(-)`
  - `Prelude.(-)` (alias for `Prelude.Num.(-)`)
  - `Prelude.Num.(-)` . . . . . .Prelude/Num.ad, line 35
    * constant of type-class `NumMinus`

        * constant of type-class `Ord`
        * type `'a -> 'a -> Bool`
        * executable

- `(>>)`
  - `Prelude.(>>)` (alias for `Prelude.Monad.(>>)`)
  - `Prelude.Monad.(>>)` . . . . . Prelude/Monad.ad, line 30
    * statically resolved constant of type-class `Monad`
    * type `'M 'a -> 'M 'b -> 'M 'b`
    * executable
- `(>>=)`
  - `Prelude.(>>=)` (alias for `Prelude.Monad.(>>=)`)
  - `Prelude.Monad.(>>=)` . . . . . Prelude/Monad.ad, line 29
    * statically resolved constant of type-class `Monad`
    * type `'M 'a -> ('a -> 'M 'b) -> 'M 'b`
    * executable
- `(\/)`
  - `Prelude.(\/)` (alias for `Prelude.Bool.(\/)`)
  - `Prelude.Bool.(\/)` . . . . . Prelude/Bool.ad, line 64
    * constant
    * type `Prop -> Prop -> Prop`
    * non executable
- `(^.)`
  - `Data.Lens.(^.)` (alias for `Data.Lens.view`)
- `(||)`
  - `Prelude.(||)` (alias for `Prelude.Bool.(||)`)
  - `Prelude.Bool.(||)` . . . . . Prelude/Bool.ad, line 39
    * constant
    * type `Bool -> Bool -> Bool`
    * executable

3.2.2 A

- `all`
  - `Data.List.all` (alias for `Prelude.List.all`)
  - `Prelude.all` (alias for `Prelude.Foldable.all`)
  - `Prelude.Foldable.all` . . . . . Prelude/Foldable.ad, line 32
    * constant of type-class `Foldable`
    * type `('e -> Bool) -> 't -> Bool`
    * executable
  - `Prelude.List.all` . . . . . Prelude/List.ad, line 197
    * constant
    * type `('a -> Bool) -> ['a] -> Bool`
    * executable
- `allP`

3.2.3 B

- `bind`
  - `Data.Maybe.bind` (alias for `Prelude.Maybe.bind`)
  - `Prelude.bind` (alias for `Prelude.Monad.(>>=)`)
  - `Prelude.Maybe.bind` . . . . . Prelude/Maybe.ad, line 37
    - constant
    - type `Maybe 'a -> ('a -> Maybe 'b) -> Maybe 'b`
    - executable
  - `Prelude.Monad.bind` (alias for `Prelude.Monad.(>>=)`)
- `bool2prop`
  - `Prelude.bool2prop` (alias for `Prelude.Bool.bool2prop`)
  - `Prelude.Bool.bool2prop` . . . . . Prelude/Bool.ad, line 58
    - constant
    - type `Bool -> Prop`
    - non executable
- `boolCase`
  - `Prelude.boolCase` (alias for `Prelude.Bool.boolCase`)
  - `Prelude.Bool.boolCase` . . . . . Prelude/Bool.ad, line 28
    - constant
    - type `Bool -> 'a -> 'a -> 'a`
    - executable
- `break`
  - `Data.List.break` . . . . . Data/List.ad, line 62
    - constant
    - type `('a -> Bool) -> ['a] -> (['a], ['a])`
    - executable

3.2.4 C

- `caseBool`
  - `Prelude.caseBool` (alias for `Prelude.Bool.caseBool`)
  - `Prelude.Bool.caseBool` . . . . . Prelude/Bool.ad, line 22
    - datatype case-constant
    - type `'a -> 'a -> Bool -> 'a`
    - executable
- `caseDistinctListSet`
  - `Data.Collection.DistinctListSet.caseDistinctListSet` Data/Collection/DistinctListSet.ad, line 7
    - datatype case-constant
    - type `(['a] -> 'b) -> Data.Collection.DistinctListSet.DistinctListSet 'a -> 'b`
    - executable
- `caseEither`
  - `Prelude.caseEither` (alias for `Prelude.Either.caseEither`)
  - `Prelude.Either.caseEither` . . . . . Prelude/Either.ad, line 11

  - `Prelude.List.Cons` (alias for `Prelude.List.(:)`)

- `constant`
  - `Prelude.constant` (alias for `Prelude.Function.constant`)
  - `Prelude.Function.constant` . . . . . Prelude/Function.ad, line 24
    * constant
    * type `'a -> 'b -> 'a`
    * executable

- `count`
  - `Data.List.count` (alias for `Prelude.List.count`)
  - `Prelude.count` (alias for `Prelude.List.count`)
  - `Prelude.List.count` . . . . . Prelude/List.ad, line 168
    * constant
    * type `'a -> ['a] -> Natural`
    * executable

- `curry`
  - `Prelude.curry` (alias for `Prelude.Tuple.curry`)
  - `Prelude.Tuple.curry` . . . . . Prelude/Tuple.ad, line 32
    * constant
    * type `(('a, 'b) -> 'c) -> 'a -> 'b -> 'c`
    * executable

3.2.5 D

- `default`
  - `Prelude.default` (alias for `Prelude.Default.default`)
  - `Prelude.Default.default` . . . . . Prelude/Default.ad, line 9
    * constant of type-class `Default`
    * type `'a`
    * executable

- `delete`
  - `Prelude.delete` (alias for `Prelude.Collection.delete`)
  - `Prelude.Collection.delete` . . . . . Prelude/Collection.ad, line 109
    * constant of type-class `CollectionWithDelete`
    * type `'e -> 'c -> 'c`
    * executable

- `dequeue`
  - `Data.Collection.Queue.dequeue` . . . . . Data/Collection/Queue.ad, line 20
    * constant
    * type `Data.Collection.Queue.Queue 'a -> Maybe ('a, Data.Collection.Queue.Queue 'a)`
    * executable

- `destEither`
  - `Prelude.destEither` (alias for `Prelude.Either.destEither`)
  - `Prelude.Either.destEither` . . . . . Prelude/Either.ad, line 16
    * constant

3.2.6 E

- `elem`
  - `Data.List.elem` (alias for `Prelude.List.elem`)
  - `Prelude.elem` (alias for `Prelude.Foldable.elem`)
  - `Prelude.Foldable.elem` . . . . . Prelude/Foldable.ad, line 31
    - constant of type-class `Foldable`
    - type `'e -> 't -> Bool`
    - executable
  - `Prelude.List.elem` . . . . . .Prelude/List.ad, line 214
    - constant
    - type `'a -> ['a] -> Bool`
    - executable
- `elem1`
  - `Data.Lens.elem1` . . . . . Data/Lens.ad, line 191
    - constant of type-class `Data.Lens.Elem1`
    - type `Data.Lens.Lens 's 't 'a 'b`
    - executable
- `elem2`
  - `Data.Lens.elem2` . . . . . Data/Lens.ad, line 191
    - constant of type-class `Data.Lens.Elem2`
    - type `Data.Lens.Lens 's 't 'a 'b`
    - executable
- `elem3`
  - `Data.Lens.elem3` . . . . . Data/Lens.ad, line 191
    - constant of type-class `Data.Lens.Elem3`
    - type `Data.Lens.Lens 's 't 'a 'b`
    - executable
- `elem4`
  - `Data.Lens.elem4` . . . . . Data/Lens.ad, line 191
    - constant of type-class `Data.Lens.Elem4`
    - type `Data.Lens.Lens 's 't 'a 'b`
    - executable
- `elem5`
  - `Data.Lens.elem5` . . . . . Data/Lens.ad, line 191
    - constant of type-class `Data.Lens.Elem5`
    - type `Data.Lens.Lens 's 't 'a 'b`
    - executable
- `elem6`
  - `Data.Lens.elem6` . . . . . Data/Lens.ad, line 191
    - constant of type-class `Data.Lens.Elem6`
    - type `Data.Lens.Lens 's 't 'a 'b`
    - executable
- `elem7`

3.2.7 F

3.2.8 G

3.2.9 I

- `id`
  - `Prelude.id` (alias for `Prelude.Function.id`)
  - `Prelude.Function.id` . . . . . Prelude/Function.ad, line 20
    - constant
    - type `'a -> 'a`
    - executable
- `image`
  - `Prelude.image` (alias for `Prelude.Collection.image`)
  - `Prelude.Collection.image` . . . . . Prelude/Collection.ad, line 74
    - constant
    - type `('e1 -> 'e2) -> 'C 'e1 -> 'C 'e2`
    - executable
- `insert`
  - `Prelude.insert` (alias for `Prelude.Collection.insert`)
  - `Prelude.Collection.insert` . . . . . Prelude/Collection.ad, line 46
    - constant of type-class `Collection`
    - type `'e -> 'c -> 'c`
    - executable
- `intersection`
  - `Prelude.intersection` (alias for `Prelude.Collection.intersection`)
  - `Prelude.Collection.intersection` . . . . . Prelude/Collection.ad, line 116
    - constant of type-class `CollectionWithDelete`
    - type `'c -> 'c -> 'c`
    - executable
- `isEmpty`
  - `Prelude.isEmpty` (alias for `Prelude.Collection.isEmpty`)
  - `Prelude.Collection.isEmpty` . . . . . Prelude/Collection.ad, line 80
    - constant of type-class `CollectionWithEmptynessTest`
    - type `'c -> Bool`
    - executable
- `isEmptyP`
  - `Prelude.isEmptyP` (alias for `Prelude.Collection.isEmptyP`)
  - `Prelude.Collection.isEmptyP` . . . . . Prelude/Collection.ad, line 57
    - constant of type-class `Collection`
    - type `'c -> Prop`
    - non executable
- `isJust`
  - `Data.Maybe.isJust` (alias for `Prelude.Maybe.isJust`)
  - `Prelude.isJust` (alias for `Prelude.Maybe.isJust`)
  - `Prelude.Maybe.isJust` . . . . . Prelude/Maybe.ad, line 48
    - constant
    - type `Maybe 'a -> Bool`

    * executable

- `isNothing`
  - `Data.Maybe.isNothing` (alias for `Prelude.Maybe.isNothing`)
  - `Prelude.isNothing` (alias for `Prelude.Maybe.isNothing`)
  - `Prelude.Maybe.isNothing` . . . . . Prelude/Maybe.ad, line 48
    * constant
    * type `Maybe 'a -> Bool`
    * executable
- `isProperSubsetOf`
  - `Prelude.isProperSubsetOf` (alias for `Prelude.Collection.isProperSubsetOf`)
  - `Prelude.Collection.isProperSubsetOf` . . . . . Prelude/Collection.ad, line 166
    * constant of type-class `CollectionWithSubsetTest`
    * type `'c -> 'c -> Bool`
    * executable
- `isSetEquiv`
  - `Prelude.isSetEquiv` (alias for `Prelude.Collection.isSetEquiv`)
  - `Prelude.Collection.isSetEquiv` . . . . . Prelude/Collection.ad, line 168
    * constant of type-class `CollectionWithSubsetTest`
    * type `'c -> 'c -> Bool`
    * executable
- `isSubsetOf`
  - `Prelude.isSubsetOf` (alias for `Prelude.Collection.isSubsetOf`)
  - `Prelude.Collection.isSubsetOf` . . . . . Prelude/Collection.ad, line 165
    * constant of type-class `CollectionWithSubsetTest`
    * type `'c -> 'c -> Bool`
    * executable
- `Itself`
  - `Prelude.Itself` (alias for `Prelude.Unit.Itself`)
  - `Prelude.Unit.Itself` . . . . . Prelude/Unit.ad, line 11
    * constructor of datatype `Itself`
    * type `(::'a)`
    * executable

3.2.10 J

- `Just`
  - `Data.Maybe.Just` (alias for `Prelude.Maybe.Just`)
  - `Prelude.Just` (alias for `Prelude.Maybe.Just`)
  - `Prelude.Maybe.Just` . . . . . Prelude/Maybe.ad, line 22
    * constructor of datatype `Maybe`
    * type `'a -> Maybe 'a`
    * executable

3.2.11 L

3.2.12 M

- `mkString`
  - `Prelude.mkString` (alias for `Prelude.String.mkString`)
  - `Prelude.String.mkString` . . . . . Prelude/String.ad, line 14
    - constant
    - type `BuiltInString -> String`
    - executable
- `mod`
  - `Prelude.mod` (alias for `Prelude.Num.mod`)
  - `Prelude.Num.mod` . . . . . Prelude/Num.ad, line 49
    - constant of type-class `NumDivMod`
    - type `'a -> 'a -> 'a`
    - executable

3.2.13 N

- `Nil`
  - `Data.List.Nil` (alias for `Prelude.List.Nil`)
  - `Prelude.Nil` (alias for `Prelude.List.Nil`)
  - `Prelude.List.Nil` . . . . . Prelude/List.ad, line 50
    - constructor of datatype `List`
    - type `['a]`
    - executable
- `not`
  - `Prelude.not` (alias for `Prelude.Bool.not`)
  - `Prelude.Bool.not` . . . . . Prelude/Bool.ad, line 23
    - constant
    - type `Bool -> Bool`
    - executable
- `notElem`
  - `Data.List.notElem` (alias for `Prelude.List.notElem`)
  - `Prelude.notElem` (alias for `Prelude.List.notElem`)
  - `Prelude.List.notElem` . . . . . Prelude/List.ad, line 218
    - constant
    - type `'a -> ['a] -> Bool`
    - executable
- `notElemP`
  - `Data.List.notElemP` (alias for `Prelude.List.notElemP`)
  - `Prelude.List.notElemP` . . . . . Prelude/List.ad, line 227
    - constant
    - type `'a -> ['a] -> Prop`
    - non executable
- `Nothing`
  - `Data.Maybe.Nothing` (alias for `Prelude.Maybe.Nothing`)
  - `Prelude.Nothing` (alias for `Prelude.Maybe.Nothing`)

3.2.14 O

- or
  - Data.List.or (alias for Prelude.List.or)
  - Prelude.or (alias for Prelude.List.or)
  - Prelude.List.or . . . . . .Prelude/List.ad, line 188
    - constant
    - type [Bool] -> Bool
    - executable
- ord
  - Prelude.ord (alias for Prelude.Char.ord)
  - Prelude.Char.ord . . . . . .Prelude/Char.ad, line 32
    - constant
    - type Char -> Natural
    - executable
- orP
  - Data.List.orP (alias for Prelude.List.orP)
  - Prelude.orP (alias for Prelude.List.orP)
  - Prelude.List.orP . . . . . .Prelude/List.ad, line 193
    - constant
    - type [Prop] -> Prop
    - non executable

3.2.15 P

- pack
  - Prelude.pack (alias for Prelude.Text.pack)
  - Prelude.Text.pack . . . . . .Prelude/Text.ad, line 20
    - constant
    - type String -> Text
    - executable
- partition
  - Data.List.partition (alias for Prelude.List.partition)
  - Prelude.partition (alias for Prelude.Collection.partition)
  - Prelude.Collection.partition . . . . . Prelude/Collection.ad, line 119
    - constant of type-class CollectionWithDelete
    - type ('e -> Bool) -> 'c -> ('c, 'c)
    - executable
  - Prelude.List.partition . . . . . .Prelude/List.ad, line 105
    - constant
    - type ('a -> Bool) -> ['a] -> (['a], ['a])
    - executable
- pop
  - Data.Collection.Stack.pop . . . . . Data/Collection/Stack.ad, line 18
    - constant
    - type Data.Collection.Stack.Stack 'a -> Maybe ('a, Data.Collection.Stack.Stack 'a)
    - executable

- pred
  - Prelude.pred (alias for Prelude.BasicClasses.pred)
  - Prelude.BasicClasses.pred . . . . . Prelude/BasicClasses.ad, line 27
    - constant of type-class Enum
    - type 'a -> 'a
    - executable
- pure
  - Prelude.pure (alias for Prelude.Monad.return)
  - Prelude.Monad.pure (alias for Prelude.Monad.return)
- push
  - Data.Collection.Stack.push . . . . . Data/Collection/Stack.ad, line 15
    - constant
    - type 'a -> Data.Collection.Stack.Stack 'a -> Data.Collection.Stack.Stack 'a
    - executable

3.2.16 R

- removeFirst
  - Data.List.removeFirst (alias for Prelude.List.removeFirst)
  - Prelude.removeFirst (alias for Prelude.List.removeFirst)
  - Prelude.List.removeFirst . . . . . .Prelude/List.ad, line 123
    - constant
    - type ('a -> Bool) -> ['a] -> ['a]
    - executable
- replicate
  - Data.List.replicate . . . . . Data/List.ad, line 16
    - constant
    - type Natural -> 'a -> ['a]
    - executable
- return
  - Prelude.return (alias for Prelude.Monad.return)
  - Prelude.Monad.return . . . . . Prelude/Monad.ad, line 31
    - statically resolved constant of type-class Monad
    - type 'a -> 'M 'a
    - executable
- revAppend
  - Data.List.revAppend (alias for Prelude.List.revAppend)
  - Prelude.revAppend (alias for Prelude.List.revAppend)
  - Prelude.List.revAppend . . . . . .Prelude/List.ad, line 171
    - constant
    - type ['a] -> ['a] -> ['a]
    - executable
- reverse
  - Data.List.reverse (alias for Prelude.List.reverse)

3.2.17 S

    * constant of type-class `Foldable`
    * type `'t -> Natural`
    * executable

- `snd`
  - `Prelude.snd` (alias for `Prelude.Tuple.snd`)
  - `Prelude.Tuple.snd` . . . . . Prelude/Tuple.ad, line 29
    * constant
    * type `('a, 'b) -> 'b`
    * executable

- `snoc`
  - `Data.List.snoc` (alias for `Prelude.List.snoc`)
  - `Prelude.snoc` (alias for `Prelude.List.snoc`)
  - `Prelude.List.snoc` . . . . . Prelude/List.ad, line 74
    * constant
    * type `['a] -> 'a -> ['a]`
    * executable

- `span`
  - `Data.List.span` . . . . . Data/List.ad, line 53
    * constant
    * type `('a -> Bool) -> ['a] -> (['a], ['a])`
    * executable

- `splitAt`
  - `Data.List.splitAt` . . . . . Data/List.ad, line 30
    * constant
    * type `Natural -> ['a] -> (['a], ['a])`
    * executable

- `Suc`
  - `Prelude.Suc` (alias for `Prelude.Num.Suc`)
  - `Prelude.Num.Suc` . . . . . .Prelude/Num.ad, line 21
    * constructor of datatype `Natural`
    * type `Natural -> Natural`
    * executable

- `succ`
  - `Prelude.succ` (alias for `Prelude.BasicClasses.succ`)
  - `Prelude.BasicClasses.succ` . . . . . Prelude/BasicClasses.ad, line 26
    * constant of type-class `Enum`
    * type `'a -> 'a`
    * executable

3.2.18 T

- take
  - Data.List.take . . . . . Data/List.ad, line 20
    - constant
    - type Natural -> ['a] -> ['a]
    - executable
- takeFirst
  - Data.List.takeFirst (alias for Prelude.List.takeFirst)
  - Prelude.takeFirst (alias for Prelude.List.takeFirst)
  - Prelude.List.takeFirst . . . . . .Prelude/List.ad, line 114
    - constant
    - type ('a -> Bool) -> ['a] -> Maybe (['a], 'a)
    - executable
- takeWhile
  - Data.List.takeWhile . . . . . Data/List.ad, line 43
    - constant
    - type ('a -> Bool) -> ['a] -> ['a]
    - executable
- to
  - Data.Lens.to . . . . . .Data/Lens.ad, line 55
    - constant
    - type ('a -> 'b) -> Data.Lens.Getter 'a 'b
    - executable
- toEnum
  - Prelude.toEnum (alias for Prelude.BasicClasses.toEnum)
  - Prelude.BasicClasses.toEnum . . . . . Prelude/BasicClasses.ad, line 30
    - constant of type-class Enum
    - type Natural -> Maybe 'a
    - executable
- toGetter
  - Data.Lens.toGetter . . . . . .Data/Lens.ad, line 71
    - constant of type-class Data.Lens.IsGetter
    - type 'v -> Data.Lens.Getter 'a 'b
    - executable
- toList
  - Prelude.toList (alias for Prelude.Foldable.toList)
  - Prelude.Foldable.toList . . . . . Prelude/Foldable.ad, line 20
    - constant of type-class Foldable
    - type 't -> ['e]
    - executable
- toSetter
  - Data.Lens.toSetter . . . . . .Data/Lens.ad, line 82
    - constant of type-class Data.Lens.IsSetter

    * type 'v -> Data.Lens.Setter 's 't 'b
    * executable

- true
  - Prelude.true (alias for Prelude.Bool.true)
  - Prelude.Bool.true . . . . . Prelude/Bool.ad, line 55
    * constant
    * type Prop
    * non executable

- True
  - Prelude.True (alias for Prelude.Bool.True)
  - Prelude.Bool.True . . . . . Prelude/Bool.ad, line 22
    * constructor of datatype Bool
    * type Bool
    * executable

3.2.19 U

- uexistsB_fun
  - Prelude.uexistsB_fun (alias for Prelude.BasicClasses.uexistsB_fun)
  - Prelude.BasicClasses.uexistsB_fun . . . . . Prelude/BasicClasses.ad, line 98
    * constant of type-class Finite
    * type ('a -> Bool) -> Bool
    * executable

- uexistsP_fun
  - Prelude.uexistsP_fun (alias for Prelude.Bool.uexistsP_fun)
  - Prelude.Bool.uexistsP_fun . . . . . Prelude/Bool.ad, line 72
    * constant
    * type ('a -> Prop) -> Prop
    * non executable

- uncurry
  - Prelude.uncurry (alias for Prelude.Tuple.uncurry)
  - Prelude.Tuple.uncurry . . . . . Prelude/Tuple.ad, line 35
    * constant
    * type ('a -> 'b -> 'c) -> ('a, 'b) -> 'c
    * executable

- undefined
  - Prelude.undefined (alias for Prelude.Default.undefined)
  - Prelude.Default.undefined . . . . . Prelude/Default.ad, line 13
    * constant of type-class Default
    * type 'a
    * executable

- union
  - Prelude.union (alias for Prelude.Collection.union)
  - Prelude.Collection.union . . . . . Prelude/Collection.ad, line 69

    * constant
    * type `'c -> 'c -> 'c`
    * executable

- `Unit`
  - `Prelude.Unit` (alias for `Prelude.Unit.Unit`)
  - `Prelude.Unit.Unit` . . . . . Prelude/Unit.ad, line 8
    * constructor of datatype `()`
    * type `()`
    * executable

- `universeFoldWithAbort`
  - `Prelude.universeFoldWithAbort` (alias for `Prelude.BasicClasses.universeFoldWithAbort`)
  - `Prelude.BasicClasses.universeFoldWithAbort` . . . . . Prelude/BasicClasses.ad, line 93
    * statically resolved constant of type-class `Finite`
    * type `('acc -> Bool) -> ('acc -> 'a -> 'acc) -> 'acc -> 'acc`
    * executable

- `universeList`
  - `Prelude.universeList` (alias for `Prelude.BasicClasses.universeList`)
  - `Prelude.BasicClasses.universeList` . . . . . Prelude/BasicClasses.ad, line 91
    * constant of type-class `Finite`
    * type `['a]`
    * executable

- `unpack`
  - `Prelude.unpack` (alias for `Prelude.Text.unpack`)
  - `Prelude.Text.unpack` . . . . . Prelude/Text.ad, line 23
    * constant
    * type `Text -> String`
    * executable

- `unreachable`
  - `Prelude.unreachable` (alias for `Prelude.Default.unreachable`)
  - `Prelude.Default.unreachable` . . . . . Prelude/Default.ad, line 18
    * constant
    * type `'a`
    * executable

- `unzip`
  - `Data.List.unzip` . . . . . Data/List.ad, line 79
    * constant
    * type `[('a, 'b)] -> (['a], ['b])`
    * executable

3.2.20 V

- `view`
  - `Data.Lens.view` . . . . . Data/Lens.ad, line 69
    * constant of type-class `Data.Lens.IsGetter`
    * type `'v -> 'a -> 'b`
    * executable

3.2.21 Z

- `Zero`
  - `Prelude.Zero` (alias for `Prelude.Num.Zero`)
  - `Prelude.Num.Zero` . . . . . Prelude/Num.ad, line 20
    - constructor of datatype `Natural`
    - type `Natural`
    - executable
- `zip`
  - `Data.List.zip` . . . . . Data/List.ad, line 76
    - constant
    - type `['a] -> ['b] -> [('a, 'b)]`
    - executable

3.3 Templates

3.3.1 B

- `BuildInstance`
  - `Prelude.BuildInstance` (alias for `Prelude.Template.BuildInstance`)
  - `Prelude.Template.BuildInstance` . . . . . Prelude/Template.ad, lines 215 - 219
- `BuildInstanceHeader`
  - `Prelude.BuildInstanceHeader` (alias for `Prelude.Template.BuildInstanceHeader`)
  - `Prelude.Template.BuildInstanceHeader` . . . . . Prelude/Template.ad, lines 205 - 212
- `BuildRecordFieldLens`
  - `Data.Lens.BuildRecordFieldLens` . . . . . Data/Lens.ad, lines 224 - 238
- `BuildRecordFieldLenses`
  - `Data.Lens.BuildRecordFieldLenses` . . . . . Data/Lens.ad, line 257
- `BuildRecordFieldLensesPrefix`
  - `Data.Lens.BuildRecordFieldLensesPrefix` . . . . . Data/Lens.ad, lines 249 - 255
- `BuildRecordFieldLensesSuffix`
  - `Data.Lens.BuildRecordFieldLensesSuffix` . . . . . Data/Lens.ad, lines 241 - 247
- `BuildTuple`
  - `Prelude.BuildTuple` (alias for `Prelude.Template.BuildTuple`)
  - `Prelude.Template.BuildTuple` . . . . . Prelude/Template.ad, lines 79 - 88
- `BuildTupleLeft`
  - `Prelude.BuildTupleLeft` (alias for `Prelude.Template.BuildTupleLeft`)
  - `Prelude.Template.BuildTupleLeft` . . . . . Prelude/Template.ad, lines 91 - 104
- `BuildTupleRight`
  - `Prelude.BuildTupleRight` (alias for `Prelude.Template.BuildTupleRight`)
  - `Prelude.Template.BuildTupleRight` . . . . . Prelude/Template.ad, lines 108 - 121

3.3.2 C

- ClassWithArgs
  - Prelude.ClassWithArgs (alias for Prelude.Template.ClassWithArgs)
  - Prelude.Template.ClassWithArgs . . . . . Prelude/Template.ad, lines 197 - 200
- Commafy
  - Prelude.Commafy (alias for Prelude.Template.Commafy)
  - Prelude.Template.Commafy . . . . . Prelude/Template.ad, lines 74 - 76
- ConstWithArgs
  - Prelude.ConstWithArgs (alias for Prelude.Template.ConstWithArgs)
  - Prelude.Template.ConstWithArgs . . . . . Prelude/Template.ad, lines 162 - 165
- ConstWithWildcardArgs
  - Prelude.ConstWithWildcardArgs (alias for Prelude.Template.ConstWithWildcardArgs)
  - Prelude.Template.ConstWithWildcardArgs . . . . . Prelude/Template.ad, lines 151 - 154

3.3.3 D

- DeriveInstance
  - Prelude.DeriveInstance (alias for Prelude.Template.DeriveInstance)
  - Prelude.Template.DeriveInstance . . . . . Prelude/Template.ad, lines 243 - 248
- DeriveInstanceComputeSuperDefault
  - Prelude.DeriveInstanceComputeSuperDefault (alias for Prelude.Template.DeriveInstanceComputeSuperDefault)
  - Prelude.Template.DeriveInstanceComputeSuperDefault . . . . Prelude/Template.ad, lines 226 - 238
- DerivingInstance
  - Prelude.DerivingInstance (alias for Prelude.Template.DerivingInstance)
  - Prelude.Template.DerivingInstance . . . . . Prelude/Template.ad, lines 251 - 255

3.3.4 G

- GenerateRecognisers
  - Prelude.GenerateRecognisers (alias for Prelude.Template.GenerateRecognisers)
  - Prelude.Template.GenerateRecognisers . . . . . Prelude/Template.ad, lines 179 - 189
- GetTypeVar
  - Prelude.GetTypeVar (alias for Prelude.Template.GetTypeVar)
  - Prelude.Template.GetTypeVar . . . . . Prelude/Template.ad, lines 30 - 39
- GetTypeVars
  - Prelude.GetTypeVars (alias for Prelude.Template.GetTypeVars)
  - Prelude.Template.GetTypeVars . . . . . Prelude/Template.ad, lines 42 - 48
- GetTypeVarsNumbered
  - Prelude.GetTypeVarsNumbered (alias for Prelude.Template.GetTypeVarsNumbered)
  - Prelude.Template.GetTypeVarsNumbered . . . . . Prelude/Template.ad, lines 51 - 58

3.3.5 I

- Intercalate
  - Prelude.Intercalate (alias for Prelude.Template.Intercalate)
  - Prelude.Template.Intercalate . . . . . Prelude/Template.ad, lines 61 - 71
- IsSimpleEnumDatatype
  - Prelude.IsSimpleEnumDatatype (alias for Prelude.Template.IsSimpleEnumDatatype)
  - Prelude.Template.IsSimpleEnumDatatype . . . . . Prelude/Template.ad, lines 124 - 133

3.3.6 T

- TypeWithArgs
  - Prelude.TypeWithArgs (alias for Prelude.Template.TypeWithArgs)
  - Prelude.Template.TypeWithArgs . . . . . Prelude/Template.ad, lines 173 - 176

3.4 Types

3.4.1 B

- Bool
  - Prelude.Bool (alias for Prelude.Bool.Bool)
  - Prelude.Bool.Bool . . . . . Prelude/Bool.ad, line 22
    - datatype Bool
    - constructors
      - False :: Bool
      - True :: Bool
- BuiltInChar
  - Prelude.BuiltInChar (alias for Prelude.Literal.BuiltInChar)
  - Prelude.Literal.BuiltInChar . . . . . Prelude/Literal.ad, line 49
    - type BuiltInChar
- BuiltInDecimal
  - Prelude.BuiltInDecimal (alias for Prelude.Literal.BuiltInDecimal)
  - Prelude.Literal.BuiltInDecimal . . . . . Prelude/Literal.ad, line 104
    - type BuiltInDecimal
- BuiltInNum
  - Prelude.BuiltInNum (alias for Prelude.Literal.BuiltInNum)
  - Prelude.Literal.BuiltInNum . . . . . Prelude/Literal.ad, line 19
    - type BuiltInNum
- BuiltInString
  - Prelude.BuiltInString (alias for Prelude.Literal.BuiltInString)
  - Prelude.Literal.BuiltInString . . . . . Prelude/Literal.ad, line 77
    - type BuiltInString

3.4.2 C

- Char
  - Prelude.Char (alias for Prelude.Char.Char)
  - Prelude.Char.Char . . . . . .Prelude/Char.ad, line 20
    - datatype Char
    - constructors
      - Char :: Natural -> Char

3.4.3 D

- DistinctListSet
  - Data.Collection.DistinctListSet.DistinctListSet . . Data/Collection/DistinctListSet.ad, line 7
    - datatype DistinctListSet 'a
    - constructors
      - DistinctListSet :: ['a] -> Data.Collection.DistinctListSet.DistinctListSet 'a

3.4.4 E

- Either
  - Prelude.Either (alias for Prelude.Either.Either)
  - Prelude.Either.Either . . . . . .Prelude/Either.ad, lines 11 - 13
    - datatype Either 'a 'b
    - constructors
      - Left :: 'a -> Either 'a 'b
      - Right :: 'b -> Either 'a 'b

3.4.5 F

- FunSetBool
  - Data.Collection.FunSetBool.FunSetBool . . . . . .Data/Collection/FunSetBool.ad, line 6
    - datatype FunSetBool 'a
    - constructors
      - FunSetBool :: ('a -> Bool) -> Data.Collection.FunSetBool.FunSetBool 'a

3.4.6 G

- Getter
  - Data.Lens.Getter . . . . . .Data/Lens.ad, lines 50 - 52
    - record-type Getter 'a 'b
    - fields
      - getterGet :: Data.Lens.Getter 'a 'b -> 'a -> 'b

3.4.7 I

- Itself
  - Prelude.Itself (alias for Prelude.Unit.Itself)
  - Prelude.Unit.Itself . . . . . .Prelude/Unit.ad, line 11
    - datatype Itself 'a
    - constructors
      - Itself :: (::'a)

3.4.8 L

- Lens
  - Data.Lens.Lens . . . . . Data/Lens.ad, lines 41 - 44
    - record-type Lens 's 't 'a 'b
    - fields
      - lensGet :: Data.Lens.Lens 's 't 'a 'b -> 's -> 'a
      - lensSet :: Data.Lens.Lens 's 't 'a 'b -> 's -> 'b -> 't
- List
  - Data.List.List (alias for Prelude.List.List)
  - Prelude.List (alias for Prelude.List.List)
  - Prelude.List.List . . . . . Prelude/List.ad, line 50
    - datatype List 'a
    - constructors
      - Nil :: ['a]
      - (:) :: 'a -> ['a] -> ['a]
- ListBag
  - Data.Collection.ListBag.ListBag . . . . . Data/Collection/ListBag.ad, line 8
    - datatype ListBag 'a
    - constructors
      - ListBag :: ['a] -> Data.Collection.ListBag.ListBag 'a

3.4.9 M

- Maybe
  - Data.Maybe.Maybe (alias for Prelude.Maybe.Maybe)
  - Prelude.Maybe (alias for Prelude.Maybe.Maybe)
  - Prelude.Maybe.Maybe . . . . . Prelude/Maybe.ad, lines 20 - 22
    - datatype Maybe 'a
    - constructors
      - Nothing :: Maybe 'a
      - Just :: 'a -> Maybe 'a

3.4.10 N

- Natural
  - Prelude.Natural (alias for Prelude.Num.Natural)
  - Prelude.Num.Natural . . . . . Prelude/Num.ad, lines 19 - 21
    - datatype Natural
    - constructors
      - Zero :: Natural
      - Suc :: Natural -> Natural

3.4.11 O

- `Ordering`
  - `Prelude.Ordering` (alias for `Prelude.Ord.Ordering`)
  - `Prelude.Ord.Ordering` . . . . . Prelude/Ord.ad, lines 16 - 19
    - datatype `Ordering`
    - constructors
      - `LT` :: `Ordering`
      - `EQ` :: `Ordering`
      - `GT` :: `Ordering`

3.4.12 P

- `Prop`
  - `Prelude.Prop` (alias for `Prelude.Bool.Prop`)
  - `Prelude.Bool.Prop` . . . . . Prelude/Bool.ad, line 53
    - type `Prop`
    - non executable

3.4.13 Q

- `Queue`
  - `Data.Collection.Queue.Queue` . . . . . Data/Collection/Queue.ad, line 10
    - datatype `Queue 'a`
    - constructors
      - `Queue` :: `['a] -> ['a] -> Data.Collection.Queue.Queue 'a`

3.4.14 S

- `Setter`
  - `Data.Lens.Setter` . . . . . Data/Lens.ad, lines 60 - 62
    - record-type `Setter 's 't 'b`
    - fields
      - `setterSet` :: `Data.Lens.Setter 's 't 'b -> 's -> 'b -> 't`
- `SimpleLens`
  - `Data.Lens.SimpleLens` . . . . . Data/Lens.ad, line 47
    - abbreviation-type `SimpleLens 's 'a`
    - abbreviation for `Data.Lens.Lens 's 's 'a 'a`
- `SimpleSetter`
  - `Data.Lens.SimpleSetter` . . . . . Data/Lens.ad, line 64
    - abbreviation-type `SimpleSetter 's 'b`
    - abbreviation for `Data.Lens.Setter 's 's 'b`
- `Stack`
  - `Data.Collection.Stack.Stack` . . . . . Data/Collection/Stack.ad, line 10
    - datatype `Stack 'a`
    - constructors
      - `Stack` :: `['a] -> Data.Collection.Stack.Stack 'a`
- `String`

3.4.15 T

3.4.16 U