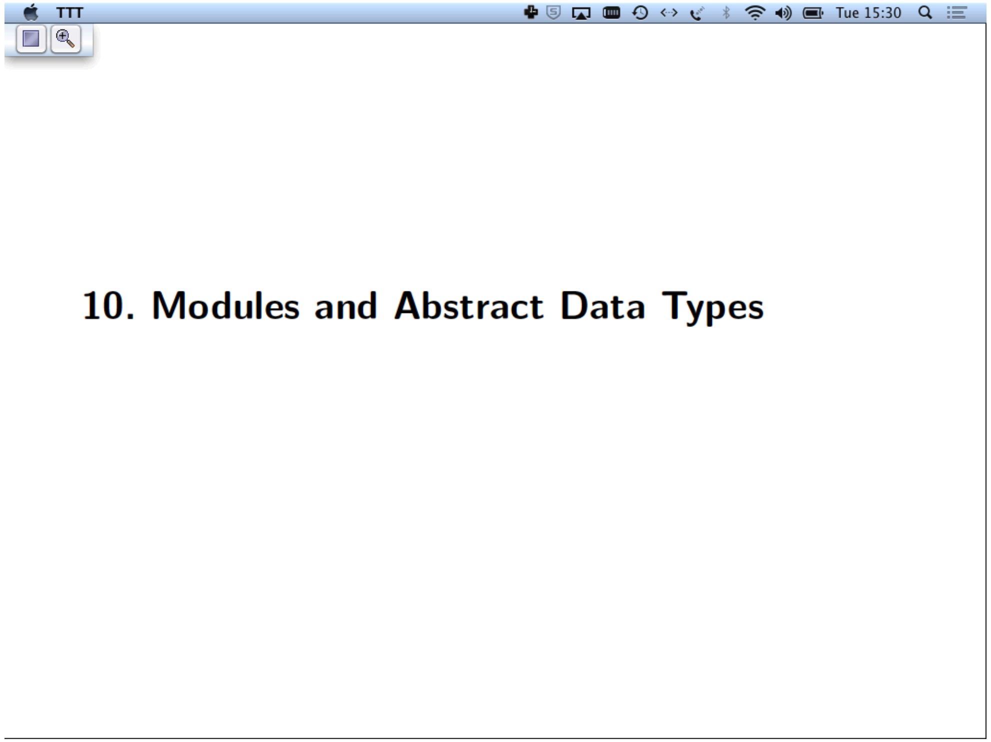
Script generated by TTT

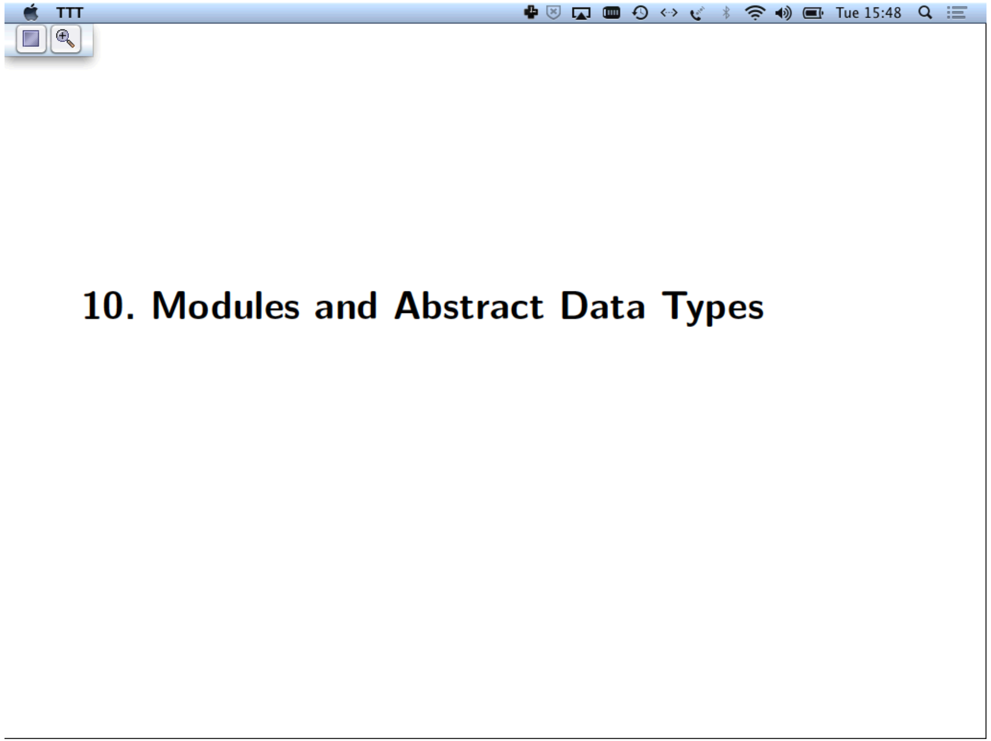
Title: Nipkow: Info2 (07.01.2014)

Date: Tue Jan 07 15:30:35 CET 2014

Duration: 88:46 min

Pages: 119





10.1 Modules

10.1 Modules

Module = collection of type, function, class etc definitions

10.1 Modules

Module = collection of type, function, class etc definitions

Purposes:

- Grouping
- Interfaces
- Division of labour
- Name space management: `M.f` vs `f`

10.1 Modules

Module = collection of type, function, class etc definitions

Purposes:

- Grouping
- Interfaces
- Division of labour
- Name space management: `M.f` vs `f`

- Information hiding

10.1 Modules

Module = collection of type, function, class etc definitions

Purposes:

- Grouping
- Interfaces
- Division of labour
- Name space management: `M.f` vs `f`
- Information hiding

GHC: one module per file

10.1 Modules

Module = collection of type, function, class etc definitions

Purposes:

- Grouping
- Interfaces
- Division of labour
- Name space management: `M.f` vs `f`
- Information hiding

GHC: one module per file

Recommendation: module `M` in file `M.hs`

Module header

```
module M where    -- M must start with capital letter
↑
```

All definitions must start in this column

Module header

```
module M where    -- M must start with capital letter
↑
```

All definitions must start in this column

- Exports everything defined in `M` (at the top level)

Module header

```
module M where    -- M must start with capital letter
↑
```

All definitions must start in this column

- Exports everything defined in `M` (at the top level)

Selective export:

```
module M (T, f, ...)  where
```

Exporting data types

```
module M (T) where
data T = ...
```

Exporting data types

```
module M (T) where
data T = ...
```

- Exports only `T`, but not its constructors

Exporting data types

```
module M (T) where
data T = ...
```

- Exports only `T`, but not its constructors

```
module M (T(C,D,...)) where
data T = ...
```

Exporting data types

```
module M (T) where
data T = ...
```

- Exports only T, but not its constructors

```
module M (T(C,D,...)) where
data T = ...
```

- Exports T and its constructors C, D, ...

```
module M (T(..)) where
data T = ...
```

Exporting data types

```
module M (T) where
data T = ...
```

- Exports only T, but not its constructors

```
module M (T(C,D,...)) where
data T = ...
```

- Exports T and its constructors C, D, ...

```
module M (T(..)) where
data T = ...
```


- Exports T and all of its constructors

Not permitted: `module M (T,C,D) where`

Exporting modules

Exporting modules

By default, modules do not export names from imported modules

Exporting modules

By default, modules do not export names from imported modules

```
module B where
import A
...
```

```
module A where
f = ...
...
```

Exporting modules

By default, modules do not export names from imported modules

```
module B where
import A
...
```

```
module A where
f = ...
...
```

⟹ B does not export f

Unless the names are mentioned in the export list

```
module B (f) where
import A
...
```

Or the whole module is exported

Exporting modules

By default, modules do not export names from imported modules

```
module B where
import A
...
```

```
module A where
f = ...
...
```

⟹ B does not export f

Unless the names are mentioned in the export list

```
module B (f) where
import A
...
```

Or the whole module is exported

```
module B (module A) where
import A
...
```

import

By default, everything that is exported is imported

import

By default, everything that is exported is imported

```
module B where          module A where
import A                f = ...
...                     g = ...
```

⟹ B imports f and g

Unless an import list is specified

import

By default, everything that is exported is imported

```
module B where          module A where
import A                f = ...
...                     g = ...
```

⟹ B imports f and g

Unless an import list is specified

```
module B where
import A (f)
...
```

⟹ B imports only f

import

By default, everything that is exported is imported

```
module B where          module A where
import A                f = ...
...                     g = ...
```

⟹ B imports f and g

Unless an import list is specified

```
module B where
import A (f)
...
```

⟹ B imports only f

Or specific names are hidden

```
module B where
import A hiding (g)
...
```

qualified

```
import A
import B
import C
... f ...
```

Where does f come from??

qualified

```
import A
import B
import C
... f ...
```

Where does `f` come from??

Clearer: *qualified names*

```
... A.f ...
```

qualified

```
import A
import B
import C
... f ...
```

Where does `f` come from??

Clearer: *qualified names*

```
... A.f ...
```


Can be enforced:

```
import qualified A
```

Renaming modules

```
import TotallyAwesomeModule

... TotallyAwesomeModule.f ...
```

Renaming modules

```
import TotallyAwesomeModule

... TotallyAwesomeModule.f ...
```


Painful

More readable:

```
import qualified TotallyAwesomeModule as TAM
```

For the full description of the module system
see the Haskell report

Renaming modules

```
import TotallyAwesomeModule

... TotallyAwesomeModule.f ...
```

Painful

More readable:

```
import qualified TotallyAwesomeModule as TAM

... TAM.f ...
```

10.2 Abstract Data Types

10.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

10.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

Why? Example: sets implemented as lists without duplicates

10.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

Why? Example: sets implemented as lists without duplicates

- Could create illegal value: `[1, 1]`

10.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

Why? Example: sets implemented as lists without duplicates

- Could create illegal value: `[1, 1]`
- Could distinguish what should be indistinguishable: `[1, 2] /= [2, 1]`

10.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

Why? Example: sets implemented as lists without duplicates

- Could create illegal value: `[1, 1]`
- Could distinguish what should be indistinguishable: `[1, 2] /= [2, 1]`
- Cannot easily change representation later

Example: Sets

```
module Set where
-- sets are represented as lists w/o duplicates
type Set a = [a]
empty  :: Set a
empty  =  []
insert :: a -> Set a -> Set a
insert x xs  =  ...
isin :: a -> Set a -> Set a
isin x xs  =  ...
size :: Set a -> Integer
size xs  =  ...
```


Example: Sets

```
module Set where
-- sets are represented as lists w/o duplicates
type Set a = [a]
empty  :: Set a
empty  =  []
insert :: a -> Set a -> Set a
insert x xs  =  ...
isin :: a -> Set a -> Set a
isin x xs  =  ...
size :: Set a -> Integer
size xs  =  ...
```


Exposes everything

Example: Sets

```
module Set where
-- sets are represented as lists w/o duplicates
type Set a = [a]
empty  :: Set a
empty  =  []
insert :: a -> Set a -> Set a
insert x xs  =  ...
isin :: a -> Set a -> Set a
isin x xs  =  ...
size :: Set a -> Integer
size xs  =  ...
```

Exposes everything

Allows nonsense like `Set.size [1,1]`

Better

```
module Set (Set, empty, insert, isin, size) where
```

Better

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty  :: Set a
insert :: Eq a => a -> Set a -> Set a
isin   :: Eq a => a -> Set a -> Bool
size   :: Set a -> Int
```

Better

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty  :: Set a
insert :: Eq a => a -> Set a -> Set a
isin   :: Eq a => a -> Set a -> Bool
size   :: Set a -> Int
-- Implementation
type Set a = [a]
...
```

Better

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty  :: Set a
insert :: Eq a => a -> Set a -> Set a
isin   :: Eq a => a -> Set a -> Bool
size   :: Set a -> Int
-- Implementation
type Set a = [a]
...
```

- Explicit export list/interface
- But representation still not hidden

Better

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty  :: Set a
insert :: Eq a => a -> Set a -> Set a
isin   :: Eq a => a -> Set a -> Bool
size   :: Set a -> Int
-- Implementation
type Set a = [a]
...
```

- Explicit export list/interface
- But representation still not hidden
  Does not help: hiding the type name Set

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
```

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]
```

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
```

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
```

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
insert x (S xs) = S(if elem x xs then xs else x:xs)
```

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
insert x (S xs) = S(if elem x xs then xs else x:xs)
isin x (S xs) = elem x xs
```

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
insert x (S xs) = S(if elem x xs then xs else x:xs)
isin x (S xs) = elem x xs
size (S xs) = length xs
```

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
insert x (S xs) = S(if elem x xs then xs else x:xs)
isin x (S xs) = elem x xs
size (S xs) = length xs
```

Cannot construct values of type `Set` outside of module `Set` because `S` is not exported

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
insert x (S xs) = S(if elem x xs then xs else x:xs)
isin x (S xs) = elem x xs
size (S xs) = length xs
```

Cannot construct values of type Set outside of module Set because S is not exported

```
Test.hs:3:11:  Not in scope:  data constructor ‘S’
```

Uniform naming convention: S $\rightsquigarrow$ Set

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

Uniform naming convention: S $\rightsquigarrow$ Set

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

Uniform naming convention: S $\rightsquigarrow$ Set

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

Which Set is exported?

Slightly more efficient: newtype

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
newtype Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

Conceptual insight

Data representation can be hidden
by wrapping data up in a constructor that is not exported

What if Set is already a data type?

```
module SetByTree (Set, empty, insert, isin, size) where

-- Interface
empty  :: Set a
insert :: Ord a => a -> Set a -> Set a
isin   :: Ord a => a -> Set a -> Bool
size   :: Set a -> Integer

-- Implementation
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
```


What if Set is already a data type?

```
module SetByTree (Set, empty, insert, isin, size) where

-- Interface
empty  :: Set a
insert :: Ord a => a -> Set a -> Set a
isin   :: Ord a => a -> Set a -> Bool
size   :: Set a -> Integer

-- Implementation
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
```


No need for newtype:
The representation of Tree is hidden
as long as its constructors are hidden

Beware of ==

Beware of ==

```
module SetByTree (Set, empty, insert, isin, size) where
...
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq)
...
```


Beware of ==

```
module SetByTree (Set, empty, insert, isin, size) where
...
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq)
...
```


Class instances are automatically exported and cannot be hidden

Beware of ==

```
module SetByTree (Set, empty, insert, isin, size) where
...
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq)
...
```

Class instances are automatically exported and cannot be hidden

Client module:

```
import SetByTree
... insert 2 (insert 1 empty) ==
    insert 1 (insert 2 empty)
...
```

Beware of ==

```
module SetByTree (Set, empty, insert, isin, size) where
...
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
           deriving (Eq)
...
```

Class instances are automatically exported and cannot be hidden

Client module:

```
import SetByTree
... insert 2 (insert 1 empty) ==
    insert 1 (insert 2 empty)
...
```

Result is probably False — representation is partly exposed!

Beware of ==

```
module SetByTree (Set, empty, insert, isin, size) where
...
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
           deriving (Eq)
...
```

Class instances are automatically exported and cannot be hidden

Client module:

```
import SetByTree
... insert 2 (insert 1 empty) ==
    insert 1 (insert 2 empty)
...
```

Result is probably False — representation is partly exposed!

The proper treatment of ==

Some alternatives:

- Do not make Tree an instance of Eq
- Hide representation:

```
-- do not export constructor Set:
newtype Set a = Set (Tree a)
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq)
```


The proper treatment of ==

Some alternatives:

- Do not make Tree an instance of Eq
- Hide representation:

```
-- do not export constructor Set:
newtype Set a = Set (Tree a)
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq)
```


- Define the right == on Tree:

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$
and	`insert _ _`	simulates	$\{\_\} \cup \_$

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$
and	`insert _ _`	simulates	$\{\_\} \cup \_$
and	`isin _ _`	simulates	$\_ \in \_$
and	`size _`	simulates	$\lvert\_\rvert$

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$
and	`insert _ _`	simulates	$\{\_\} \cup \_$
and	`isin _ _`	simulates	$\_ \in \_$
and	`size _`	simulates	$\lvert\_\rvert$

Each concrete operation on the implementation type of lists
simulates its abstract counterpart on sets

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$
and	`insert _ _`	simulates	$\{\_\} \cup \_$
and	`isin _ _`	simulates	$\_ \in \_$
and	`size _`	simulates	$\lvert\_\rvert$

Each concrete operation on the implementation type of lists
simulates its abstract counterpart on sets

NB: We relate Haskell to mathematics

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$
and	`insert _ _`	simulates	$\{\_\} \cup \_$
and	`isin _ _`	simulates	$\_ \in \_$
and	`size _`	simulates	$\lvert\_\rvert$

Each concrete operation on the implementation type of lists
simulates its abstract counterpart on sets

NB: We relate Haskell to mathematics

For uniformity we write $\{a\}$ for the type of finite sets over type a

From lists to sets

Each list $[x_1, \dots, x_n]$ represents the set $\{x_1, \dots, x_n\}$.

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style: α `[] = {}`

α `(x:xs) = {x}` $\cup$ α `xs`

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style: α `[] = {}`

α `(x:xs) = {x}` $\cup$ α `xs`

What does it mean that "lists simulate (implement) sets":

10.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$
and	`insert _ _`	simulates	$\{\_\} \cup \_$
and	`isin _ _`	simulates	$\_ \in \_$
and	`size _`	simulates	$|\_|$

Each concrete operation on the implementation type of lists simulates its abstract counterpart on sets

NB: We relate Haskell to mathematics

For uniformity we write $\{a\}$ for the type of finite sets over type a

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style: α `[] = {}`

α `(x:xs) = {x}` $\cup$ α `xs`

What does it mean that "lists simulate (implement) sets":

α (concrete operation) = abstract operation

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style: α `[] = {}`

α `(x:xs) = {x}` $\cup$ α `xs`

What does it mean that "lists simulate (implement) sets":

α (concrete operation) = abstract operation

α `empty` = `{}`

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style: α `[] = {}`

α `(x:xs) = {x}` $\cup$ α `xs`

What does it mean that "lists simulate (implement) sets":

α (concrete operation) = abstract operation

α `empty` = `{}`

α `(insert x xs)` = `{x}` $\cup$ α `xs`

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style: α `[] = {}`

α `(x:xs) = {x}` $\cup$ α `xs`

What does it mean that "lists simulate (implement) sets":

α (concrete operation) = abstract operation

α `empty` = `{}`

α `(insert x xs)` = `{x}` $\cup$ α `xs`

`isin x xs` = `x` $\in$ α `xs`

`size xs` = $|\alpha$ `xs`$|$

For the mathematically enclined:

α must be a homomorphism

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style:

```
α []     = {}
α (x:xs) = {x} ∪ α xs
```

What does it mean that "lists simulate (implement) sets":

```
α (concrete operation)  =  abstract operation
α empty                 =  {}
α (insert x xs)         =  {x} ∪ α xs
isin x xs               =  x ∈ α xs
size xs                 =  |α xs|
```

Implementation I: lists with duplicates

```
empty        =  []
insert x xs  =  x : xs
isin x xs    =  elem x xs
size xs      =  length(nub xs)
```


Implementation I: lists with duplicates

```
empty        =  []
insert x xs  =  x : xs
isin x xs    =  elem x xs
size xs      =  length(nub xs)
```


The simulation requirements:

```
α empty  =  {}
```

Implementation I: lists with duplicates

```
empty        = []
insert x xs  = x : xs
isin x xs    = elem x xs
size xs      = length(nub xs)
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

Implementation I: lists with duplicates

```
empty        = []
insert x xs  = x : xs
isin x xs    = elem x xs
size xs      = length(nub xs)
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

Implementation I: lists with duplicates

```
empty        = []
insert x xs  = x : xs
isin x xs    = elem x xs
size xs      = length(nub xs)
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

$$\texttt{isin x xs} = \texttt{x} \in \alpha\ \texttt{xs}$$

$$\texttt{size xs} = |\alpha\ \texttt{xs}|$$

Two proofs immediate, two need lemmas proved by induction

Implementation II: lists without duplicates

```
empty        = []
insert x xs  = if elem x xs then xs else x:xs
isin x xs    = elem x xs
size xs      = length xs
```

Implementation II: lists without duplicates

```
empty         =  []
insert x xs   =  if elem x xs then xs else x:xs
isin x xs     =  elem x xs
size xs       =  length xs
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

Implementation II: lists without duplicates

```
empty         =  []
insert x xs   =  if elem x xs then xs else x:xs
isin x xs     =  elem x xs
size xs       =  length xs
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

Implementation II: lists without duplicates

```
empty         =  []
insert x xs   =  if elem x xs then xs else x:xs
isin x xs     =  elem x xs
size xs       =  length xs
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

$$\texttt{isin x xs} = \texttt{x} \in \alpha\ \texttt{xs}$$

$$\texttt{size xs} = |\alpha\ \texttt{xs}|$$

Implementation II: lists without duplicates

```
empty         =  []
insert x xs   =  if elem x xs then xs else x:xs
isin x xs     =  elem x xs
size xs       =  length xs
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] -> {a}`

$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$

In Haskell style:

```
α []     = {}
α (x:xs) = {x} ∪ α xs
```

What does it mean that "lists simulate (implement) sets":

```
α (concrete operation)  =  abstract operation
α empty                 =  {}
α (insert x xs)         =  {x} ∪ α xs
isin x xs               =  x ∈ α xs
size xs                 =  |α xs|
```

For the mathematically enclined:
α must be a homomorphism

Implementation II: lists without duplicates

```
empty       = []
insert x xs = if elem x xs then xs else x:xs
isin x xs   = elem x xs
size xs     = length xs
```

The simulation requirements:

```
α empty          =  {}
α (insert x xs)  =  {x} ∪ α xs
isin x xs        =  x ∈ α xs
size xs          =  |α xs|
```

Needs *invariant* that xs contains no duplicates

Implementation II: lists without duplicates

```
empty       = []
insert x xs = if elem x xs then xs else x:xs
isin x xs   = elem x xs
size xs     = length xs
```

The simulation requirements:

```
α empty          =  {}
α (insert x xs)  =  {x} ∪ α xs
isin x xs        =  x ∈ α xs
size xs          =  |α xs|
```

Needs *invariant* that xs contains no duplicates

```
invar :: [a] -> Bool
invar []     = True
invar (x:xs) = not(elem x xs) && invar xs
```

Implementation II: lists without duplicates

```
empty        =  []
insert x xs  =  if elem x xs then xs else x:xs
isin x xs    =  elem x xs
size xs      =  length xs
```

Revised simulation requirements:

$$
\begin{array}{rrcl}
 & \alpha\ \texttt{empty} & = & \{\} \\
\texttt{invar xs} \Longrightarrow & \alpha\ (\texttt{insert x xs}) & = & \{\texttt{x}\} \cup \alpha\ \texttt{xs} \\
\texttt{invar xs} \Longrightarrow & \texttt{isin x xs} & = & \texttt{x} \in \alpha\ \texttt{xs} \\
\texttt{invar xs} \Longrightarrow & \texttt{size xs} & = & |\alpha\ \texttt{xs}|
\end{array}
$$

Implementation II: lists without duplicates

```
empty        =  []
insert x xs  =  if elem x xs then xs else x:xs
isin x xs    =  elem x xs
size xs      =  length xs
```

Revised simulation requirements:

$$
\begin{array}{rrcl}
 & \alpha\ \texttt{empty} & = & \{\} \\
\texttt{invar xs} \Longrightarrow & \alpha\ (\texttt{insert x xs}) & = & \{\texttt{x}\} \cup \alpha\ \texttt{xs} \\
\texttt{invar xs} \Longrightarrow & \texttt{isin x xs} & = & \texttt{x} \in \alpha\ \texttt{xs} \\
\texttt{invar xs} \Longrightarrow & \texttt{size xs} & = & |\alpha\ \texttt{xs}|
\end{array}
$$

Proofs omitted.

Implementation II: lists without duplicates

```
empty        =  []
insert x xs  =  if elem x xs then xs else x:xs
isin x xs    =  elem x xs
size xs      =  length xs
```

Revised simulation requirements:

$$
\begin{array}{rrcl}
 & \alpha\ \texttt{empty} & = & \{\} \\
\texttt{invar xs} \Longrightarrow & \alpha\ (\texttt{insert x xs}) & = & \{\texttt{x}\} \cup \alpha\ \texttt{xs} \\
\texttt{invar xs} \Longrightarrow & \texttt{isin x xs} & = & \texttt{x} \in \alpha\ \texttt{xs}
\end{array}
$$

Implementation II: lists without duplicates

```
empty        =  []
insert x xs  =  if elem x xs then xs else x:xs
isin x xs    =  elem x xs
size xs      =  length xs
```

Revised simulation requirements:

Implementation II: lists without duplicates

```
empty         =  []
insert x xs   =  if elem x xs then xs else x:xs
isin x xs     =  elem x xs
size xs       =  length xs
```

Revised simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\texttt{invar xs} \Longrightarrow \alpha\ \texttt{(insert x xs)} = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

$$\texttt{invar xs} \Longrightarrow \texttt{isin x xs} = \texttt{x} \in \alpha\ \texttt{xs}$$

invar must be invariant!

In an imperative context:

If invar is true before an operation,
it must also be true after the operation

In a functional context:

If invar is true for the arguments of an operation,
it must also be true for the result of the operation

invar must be invariant!

In an imperative context:

If invar is true before an operation,
it must also be true after the operation

In a functional context:

If invar is true for the arguments of an operation,
it must also be true for the result of the operation

invar is *preserved* by every operation

invar must be invariant!

In an imperative context:

If invar is true before an operation,
it must also be true after the operation

In a functional context:

If invar is true for the arguments of an operation,
it must also be true for the result of the operation

invar is *preserved* by every operation

$$\texttt{invar empty}$$

$$\texttt{invar xs} \Longrightarrow \texttt{invar (insert x xs)}$$

`invar` must be invariant!

In an imperative context:

If `invar` is true before an operation,
it must also be true after the operation

In a functional context:

If `invar` is true for the arguments of an operation,
it must also be true for the result of the operation

`invar` is *preserved* by every operation

```
               invar empty
invar xs  ==>  invar (insert x xs)
```

Proofs do not even need induction

Summary

Summary

Let C and A be two modules that have the same interface:
a type T and a set of functions F
To prove that C is a correct implementation of A define
an *abstraction function* α :: $C.T$ -> $A.T$

Summary

Let C and A be two modules that have the same interface:
a type T and a set of functions F
To prove that C is a correct implementation of A define
an *abstraction function* α :: $C.T$ -> $A.T$
and an *invariant* `invar` :: $C.T$ -> `Bool`

Summary

Let C and A be two modules that have the same interface:
a type T and a set of functions F
To prove that C is a correct implementation of A define
an *abstraction function* α :: $C.T$ -> $A.T$
and an *invariant* invar :: $C.T$ -> Bool
and prove for each $f \in F$:

- invar is invariant:

$$\texttt{invar}\ x_1 \wedge \cdots \wedge \texttt{invar}\ x_n \implies \texttt{invar}\ (C.f\ x_1\ \ldots\ x_n)$$

Summary

Let C and A be two modules that have the same interface:
a type T and a set of functions F
To prove that C is a correct implementation of A define
an *abstraction function* α :: $C.T$ -> $A.T$
and an *invariant* invar :: $C.T$ -> Bool
and prove for each $f \in F$:

- invar is invariant:

$$\texttt{invar}\ x_1 \wedge \cdots \wedge \texttt{invar}\ x_n \implies \texttt{invar}\ (C.f\ x_1\ \ldots\ x_n)$$

(where invar is True on types other than $C.T$)

Summary

Let C and A be two modules that have the same interface:
a type T and a set of functions F
To prove that C is a correct implementation of A define
an *abstraction function* α :: $C.T$ -> $A.T$
and an *invariant* invar :: $C.T$ -> Bool
and prove for each $f \in F$:

- invar is invariant:

$$\texttt{invar}\ x_1 \wedge \cdots \wedge \texttt{invar}\ x_n \implies \texttt{invar}\ (C.f\ x_1\ \ldots\ x_n)$$

(where invar is True on types other than $C.T$)

- $C.f$ simulates $A.f$:

$$\texttt{invar}\ x_1 \wedge \cdots \wedge \texttt{invar}\ x_n \implies$$
$$\alpha(C.f\ x_1\ \ldots\ x_n) = A.f\ (\alpha\ x_1)\ \ldots\ (\alpha\ x_n)$$

(where α is the identity on types other than $C.T$)

11. Case Study: Huffman Coding