10. Modules and Abstract Data Types
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: \texttt{M.f} vs \texttt{f}
- Information hiding

GHC: one module per file
Recommendation: module \texttt{M} in file \texttt{M.hs}

module \texttt{M} where  -- M must start with capital letter
\uparrow
All definitions must start in this column

module \texttt{M} where  -- M must start with capital letter
\uparrow
All definitions must start in this column
- \textbf{Exports everything defined in M} (at the top level)
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

• Exports only T, but not its constructors

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 = ...
  • Exports T and all of its constructors

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

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 = ...
...

⇒ 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
...

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
...

Exporting modules

By default, everything that is exported is imported

import

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

```
module B where
import A
...
⇒ 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
import A
...
⇒ B imports f and g
```

Unless an import list is specified

```
module B where
import A where
import A
...
⇒ B imports only f
```

import

By default, everything that is exported is imported

```
module B where
import A
...
⇒ B imports f and g
```

Unless an import list is specified

```
module B where
import A
...
⇒ B imports only f
```

qualified

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

Where does \( f \) come from??

Or specific names are hidden

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

Where does f come from??

Clearer: *qualified names*

... A.f ...

Renaming modules

import TotallyAwesomeModule

... TotallyAwesomeModule.f ...

Renaming modules
For the full description of the module system see the Haskell report.

Renaming modules

```haskell
import TotallyAwesomeModule
...

Painful

More readable:
import qualified TotallyAwesomeModule as TAM
...
```

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
  • 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 = ...

Better

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

Exposes everything
Allows nonsense like Set.size [1,1]
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

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

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)
  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 \leadsto 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 `==`

```haskell
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 `==`

```haskell
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!

---

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)

---

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)
- Define the right `==` on Tree:
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 \mid _

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

NB: We relate Haskell to mathematics.

From lists to sets

Each list \([x_1, \ldots, x_n]\) represents the set \(\{x_1, \ldots, x_n\}\).
From lists to sets

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

**Abstraction function** \(\alpha : [a] \rightarrow \{a\}\)

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

In Haskell style:

\[\alpha [] = \{\}\]

\[\alpha (x:xs) = \{x\} \cup \alpha xs\]

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

10.3 Correctness

Why is module \(\text{Set}\) a correct implementation of (finite) sets?

Because

- \(\text{empty} \) simulates \(\{\}\)
- \(\text{insert} \) simulates \(\{\} \cup \_\)
- \(\text{isin} \) simulates \(\_ \in \_\)
- \(\text{size} \) 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** \(\alpha : [a] \rightarrow \{a\}\)

\(\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}\)

In Haskell style:
\[
\begin{align*}
\alpha \; [] &= \{} \\
\alpha \; (x:xs) &= \{x\} \cup \alpha \; xs
\end{align*}
\]

What does it mean that “lists simulate (implement) sets”:
\[
\alpha \; \text{(concrete operation)} = \text{abstract operation}
\]

\(\alpha \; \text{empty} = \{}\)

\(\alpha \; \text{insert} \; x \; \text{xs} = \{x\} \cup \alpha \; \text{xs}\)

From lists to sets

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

**Abstraction function** \(\alpha : [a] \rightarrow \{a\}\)

\(\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}\)

In Haskell style:
\[
\begin{align*}
\alpha \; [] &= \{} \\
\alpha \; (x:xs) &= \{x\} \cup \alpha \; xs
\end{align*}
\]

What does it mean that “lists simulate (implement) sets”:
\[
\alpha \; \text{(concrete operation)} = \text{abstract operation}
\]

\(\alpha \; \text{empty} = \{}\)

\(\alpha \; \text{insert} \; x \; \text{xs} = \{x\} \cup \alpha \; \text{xs}\)

\(\text{isin} \; x \; \text{xs} = x \in \alpha \; \text{xs}\)

\(\text{size} \; \text{xs} = |\alpha \; \text{xs}|\)
For the mathematically inclined:

\( \alpha \) 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**  
\( \alpha :: [a] \rightarrow \{a\} \)

\( \alpha[x_1,\ldots,x_n] = \{x_1,\ldots,x_n\} \)

In Haskell style:

\( \alpha [] = \{\} \)

\( \alpha (x:xs) = \{x\} \cup \alpha xs \)

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

\( \alpha \) (concrete operation) = abstract operation

\( \alpha \) empty = \{\}

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

isin x xs = x \in \alpha xs

size xs = |\alpha 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:

\( \alpha \) empty = \{\}
Implementation I: lists with duplicates

\[
\begin{align*}
\text{empty} &= [] \\
\text{insert } x \ x s &= x : x s \\
\text{isin } x \ x s &= \text{elem } x \ x s \\
\text{size } x s &= \text{length}(\text{nub } x s)
\end{align*}
\]

The simulation requirements:

\[
\begin{align*}
\alpha \text{ empty} &= \{\} \\
\alpha (\text{insert } x \ x s) &= \{x\} \cup \alpha x s
\end{align*}
\]

Implementation II: lists without duplicates

\[
\begin{align*}
\text{empty} &= [] \\
\text{insert } x \ x s &= \text{if elem } x \ x s \text{ then } x s \text{ else } x : x s \\
\text{isin } x \ x s &= \text{elem } x \ x s \\
\text{size } x s &= \text{length } x s
\end{align*}
\]

The simulation requirements:

\[
\begin{align*}
\alpha \text{ empty} &= \{\} \\
\alpha (\text{insert } x \ x s) &= \{x\} \cup \alpha x s \\
\text{isin } x \ x s &= x \in \alpha x s \\
\text{size } x s &= |\alpha x s|
\end{align*}
\]

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

The simulation requirements:

\[ \alpha \text{ empty } = \{\}\]

\[ \alpha \text{ (insert x xs)} = \{x\} \cup \alpha \text{ xs} \]

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:

\[ \alpha \text{ empty } = \{\}\]

\[ \alpha \text{ (insert x xs)} = \{x\} \cup \alpha \text{ xs} \]
From lists to sets

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

**Abstraction function** \(\alpha : [a] \to \{a\}\)

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

In Haskell style:

\[\begin{align*}
\alpha \ [] & = \{} \\
\alpha \ (x:xs) & = \{x\} \cup \alpha \ xs
\end{align*}\]

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

\[\begin{align*}
\alpha \text{ (concrete operation)} & = \text{abstract operation} \\
\alpha \text{ empty} & = \{}
\end{align*}\]

\[\alpha \text{ (insert x xs)} = \{x\} \cup \alpha \ xs\]

\[\text{isin} \ x \ xs = x \in \alpha \ xs\]

\[\text{size} \ xs = |\alpha \ xs|\]

Implementation II: lists without duplicates

\[\begin{align*}
\text{empty} & = [] \\
\text{insert} \ x \ xs & = \text{if elem} \ x \ xs \ \text{then} \ xs \ \text{else} \ x:xs \\
\text{isin} \ x \ xs & = \text{elem} \ x \ xs \\
\text{size} \ xs & = \text{length} \ xs
\end{align*}\]

The simulation requirements:

\[\begin{align*}
\alpha \text{ empty} & = \{}
\end{align*}\]

\[\begin{align*}
\alpha \text{ (insert x xs)} & = \{x\} \cup \alpha \ xs \\
\text{isin} \ x \ xs & = x \in \alpha \ xs \\
\text{size} \ xs & = |\alpha \ xs|
\end{align*}\]

Needs **invariant** that xs contains no duplicates

For the mathematically inclined:

\(\alpha\) must be a homomorphism

Implementation II: lists without duplicates

\[\begin{align*}
\text{empty} & = [] \\
\text{insert} \ x \ xs & = \text{if elem} \ x \ xs \ \text{then} \ xs \ \text{else} \ x:xs \\
\text{isin} \ x \ xs & = \text{elem} \ x \ xs \\
\text{size} \ xs & = \text{length} \ xs
\end{align*}\]

The simulation requirements:

\[\begin{align*}
\alpha \text{ empty} & = \{}
\end{align*}\]

\[\begin{align*}
\alpha \text{ (insert x xs)} & = \{x\} \cup \alpha \ xs \\
\text{isin} \ x \ xs & = x \in \alpha \ xs \\
\text{size} \ xs & = |\alpha \ xs|
\end{align*}\]

Needs **invariant** that xs contains no duplicates

\[\begin{align*}
\text{invar} : : [a] & \to \text{Bool} \\
\text{invar} \ [] & = \text{True} \\
\text{invar} \ (x:xs) & = \text{not(elem} \ x \ xs) \ \&\& \ \text{invar} \ xs
\end{align*}\]
Implementation II: lists without duplicates

```plaintext
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:

   \( \alpha \) empty = \{ \}
   \text{invar } xs \implies \alpha (\text{insert } x \ xs) = \{x\} \cup \alpha \ xs
   \text{invar } xs \implies \text{isin } x \ xs = x \in \alpha \ xs
   \text{invar } xs \implies \text{size } xs = |\alpha \ xs|
```

Proofs omitted.

Implementation II: lists without duplicates

```plaintext
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:

   \( \alpha \) empty = \{ \}
   \text{invar } xs \implies \alpha (\text{insert } x \ xs) = \{x\} \cup \alpha \ xs
   \text{invar } xs \implies \text{isin } x \ xs = x \in \alpha \ xs
   \text{invar } xs \implies \text{size } xs = |\alpha \ 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:

\[ \alpha \text{ empty} = \{\} \]
\[ \text{invar } xs \implies \alpha \text{ (insert } x \text{ xs)} = \{x\} \cup \alpha \text{ xs} \]
\[ \text{invar } xs \implies \text{isin } x \text{ xs } = x \in \alpha \text{ xs} \]

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

\[ \text{invar empty} \]
\[ \text{invar } xs \implies \text{invar (insert } x \text{ xs)} \]
**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 \rightarrow invar (insert x xs)`

Proofs do not even need induction

---

**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* `\alpha` :: `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* `\alpha` :: `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 $\alpha :: C.T \rightarrow A.T$
and an invariant $\text{invar} :: C.T \rightarrow \text{Bool}$
and prove for each $f \in F$:
- $\text{invar}$ is invariant:
  \[ \text{invar} \ x_1 \land \cdots \land \text{invar} \ x_n \implies \text{invar} \ (C.\ f \ x_1 \ldots x_n) \]

(where $\text{invar}$ is True on types other than $C.T$)

11. Case Study: Huffman Coding