

IA010: Principles of Programming Languages

Modules

Achim Blumensath

blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Modules

$\langle expr \rangle ::= \dots \mid \mathbf{module} \langle id \rangle \{ \langle declarations \rangle \}$
 $\mid \mathbf{module} \langle id \rangle = \langle module-expr \rangle$
 $\mid \langle module-expr \rangle . \langle id \rangle$
 $\mid \mathbf{import} \langle module-expr \rangle$

$\langle module-expr \rangle ::= \langle id \rangle \mid \langle module-expr \rangle . \langle id \rangle$

```
module Stack {  
  type stack(a) = list(a);  
  
  let empty = [];  
  
  let top(s)    { head(s) };  
  let pop(s)    { tail(s) };  
  let push(s, x) { [x|s] };  
};
```

```
...  
let s = Stack.empty;  
...  
Stack.push(s, 13);  
...
```

```
import Stack;  
let s = empty;  
...  
push(s, 13);  
...
```

Encapsulation

Function of modules

- namespaces (avoid name clashes)
- break program into small, easier to understand parts

Encapsulation

Function of modules

- namespaces (avoid name clashes)
- break program into small, easier to understand parts

Encapsulation

Modules provide access only through a well-defined **interface**.

Information hiding

To reason about a module we only need to know its interface, not the implementation.

Encapsulation

Advantages of encapsulation

- can make program **easier to understand**
(reduces information by hiding implementations behind interfaces)
- can ensure **data integrity**
- increases **maintainability**
- allows simple implementation of **separate compilation**

Disadvantages of encapsulation

- badly designed **interfaces** can complicate code and reduce performance

Abstract data types

data type encapsulated in a module

```
module Stack {  
  type stack(a);  
  let empty : stack(a)  
  let push  : stack(a) -> a -> stack(a);  
  let top   : stack(a) -> a;  
  let pop   : stack(a) -> stack(a);  
};
```

```
module Stack {  
  type stack(a) = list(a);  
  let empty : stack(a) = nil;  
  let push(st : stack(a), x : a) : stack(a) {  
    pair(x, st)  
  };  
  let top(st : stack(a)) : a {  
    case st | pair(x, xs) => x  
  };  
  let pop(st : stack(a)) : stack(a) {  
    case st | nil => nil | pair(x, xs) => xs  
  };  
};
```

```
module Stack {
  type stack(a);
  let create : unit -> stack(a);
  let empty : stack(a) -> bool;
  let push : stack(a) -> a -> unit;
  let top : stack(a) -> a;
  let pop : stack(a) -> unit;
};
```

```
module Stack {
  let create() : stack(a) {
    [ elements = nil ]
  };
  let empty(st : stack(a)) : bool {
    is_nil(st.elements)
  };
  let push(st : stack(a), x : a) : unit {
    st.elements := [x|st.elements]
  };
  let top(st : stack(a)) : a {
    head(st.elements)
  };
  let pop(st : stack(a)) : unit{
    st.elements := tail(st.elements)
  };
};
```


Parametrised modules

```
 $\langle expr \rangle ::= \dots \mid \mathbf{module} \langle id \rangle ( \langle id \rangle , \dots , \langle id \rangle ) \{ \dots \}$   
 $\mid \mathbf{module} \langle id \rangle = \langle module\text{-}expr \rangle$   
 $\mid \langle module\text{-}expr \rangle . \langle id \rangle \mid \mathbf{import} \langle module\text{-}expr \rangle$   
 $\langle module\text{-}expr \rangle ::= \langle id \rangle \mid \langle module\text{-}expr \rangle . \langle id \rangle$   
 $\mid \langle module\text{-}expr \rangle ( \langle module\text{-}expr \rangle , \dots , \langle module\text{-}expr \rangle )$ 
```

```

interface KEY {
  type t;
  type ord = | LT | EQ | GT;
  let compare : t * t -> ord;
};
module Map(Key : KEY) {
  type map(a) =
  | Leaf
  | Node(Key.t, a, map(a), map(a));

  let empty : map(a) = Leaf;

  let add(m : map(a), k : Key.t, v : a) : map(a) {
    case m
    | Leaf => Node(k, v, Leaf, Leaf)
    | Node(k2, v2, l, r) => case compare(k, k2)
      | LT => Node(k2, v2, add(l, k, v), r)
      | EQ => Node(k2, v, l, r)
      | GT => Node(k2, v2, l, add(r, k, v))
  };
  ...
};

```