Introduction to Extended ML (EML)

Anne E. Haxthausen

`ah@imm.dtu.dk`

Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark

Extended ML, EML

Extended ML (EML) is a specification language created by Kahrs, Sannella and Tarlecki.

EML extends the syntax of ML with axioms in signatures and in place of code in structures and functors.

Axioms state *what* programs are supposed to.

Development paradigm:

1. specify in EML
2. implement in executable ML subset of EML
3. prove that the implementation satisfies the specification

Why EML?

Allows for

- more precise interfaces
- abstraction and stepwise development

Learning Objectives

- How to specify ML functions and modules using EML.
- Correctness: When do functions and modules satisfy specifications in EML?

These foils are based on four lecture notes made by Don Sannella.

EML specification of a function maxelem

`maxelem` should find the maximum element of an int list

Specification in EML:

```
val maxelem : int list -> int

axiom
  Forall l => l<>nil implies member(maxelem l, l)

axiom
  Forall (a,l) => member(a,l) implies (maxelem l) >= a
```


EML notation for axioms

All expressions of type `bool` can be used as *axioms*.
Additional constructs in EML:

- `implies`, `iff`
- `==`, `=/=` (logical equality and inequality) are not restricted to types that admit equality
- `Forall` x `=>` ϕ, `Exists` x `=>` ϕ, `Exists_unique` x `=>` ϕ

An ML program for maxelem

```
fun maxelem nil = 0
  | maxelem([a]) = a
  | maxelem(a::b::l) =
      if a>maxelem(b::l) then a else maxelem(b::l)
```


Proving that maxelem satisfies its specification

- The specified type `int list -> int` of `maxelem` must be an instance of the type inferred for the implementation of `maxelem`.
- The two axioms must be satisfied by `maxelem`.

Proving `Forall l => l<>nil implies member(maxelem l, l)`

```
fun maxelem nil = 0
  | maxelem([a]) = a
  | maxelem(a::b::l) =
      if a>maxelem(b::l) then a else maxelem(b::l)
```

Proof: By induction on l.

```
Base cases:
  l=nil: the precondition is false  OK
  l=[a]: member(maxelem [a],[a]) = member(a,[a])
                                  = true  OK
```


Proving `Forall l => l<>nil implies member(maxelem l, l)`

```
fun maxelem nil = 0
  | maxelem([a]) = a
  | maxelem(a::b::l) =
      if a>maxelem(b::l) then a else maxelem(b::l)

Step case: l=a::b::l'
  Inductive hypothesis: member(maxelem b::l',b::l')
  To prove: member(maxelem a::b::l',a::b::l').

  Case 1: a > maxelem(b::l')
  member(maxelem a::b::l',a::b::l') = member(a,a::b::l')
                                     = true   OK

  Case 2: maxelem(b::l') >= a
  member(maxelem a::b::l',a::b::l')
     = member(maxelem(b::l'),a::b::l')
     = true  [IH]     OK
```


Proving `Forall (a,l) => member(a,l) implies (maxelem l) >= a`

Proof: By induction on l.

See lecture notes.

Specifying structures: example

Here is a structure that implements an unbounded array of integers indexed starting from 0, using a list of integers to represent the array:

```
structure Array =
  struct
    type array = int list
    val empty = nil
    fun put(n,v,nil) =
          if n=0 then v::nil else 0::put(n-1,v,nil)
      | put(n,v,w::l) =
          if n=0 then v::l else w::put(n-1,v,l)
    fun retrieve(n,nil) = 0
      | retrieve(n,v::l) =
          if n=0 then v else retrieve(n-1,l)
  end
```

The signature of `Array` is:

```
signature ARRAY =
  sig
    type array
    val empty : array
    val put : int * int * array -> array
    val retrieve : int * array -> int
  end
```


Adding axioms to form a more complete interface gives:

```
signature ARRAY =
  sig
    type array
    val empty : array
    val put : int * int * array -> array
    val retrieve : int * array -> int
    axiom Forall (n,v,w,l) => put(n,v,put(n,w,l)) == put(n,v,l)
    axiom Forall (n,m,v,w,l) =>
            n<>m implies put(n,v,put(m,w,l)) == put(m,w,put(n,v,l))
    axiom Forall n =>
            retrieve(n,empty) == 0
    axiom Forall (n,v,l) =>
            retrieve(n,put(n,v,l)) == v
    axiom Forall (n,m,v,l) =>
            n<>m implies retrieve(n,put(m,v,l)) == retrieve(n,l)
  end
```


We can ascribe this signature to `Array`, using the usual syntax:

```
structure Array :> ARRAY =
  struct
    type array = int list
    . . .
  end
```


Satisfaction relation :>

A structure `str` satisfies/implements an EML signature `sig`, `str :> sig`, iff

1. `str` matches `sig′`, where `sig′` is `sig` minus axioms
   => can be decided by a type checker
2. the functions in `str` satisfy the axioms in `sig`
   => must be proved either by hand or using a theorem prover

Example: `Array :> ARRAY`

Functor specification

Specification:
The format of a functor specification in EML is:

```
functor F(X : sig1) :> sig2 = ?
```

where $sig1$ and $sig2$ are EML signatures.

Satisfaction/implementation:
An implementation `functor F(X : sig1) :> sig2 = str`
satisfies this specification if:

- str satisfies $sig2$ under the assumptions of $sig1$

Applications:
Applications `F(A)` of the functor are only *legal*, if:

- A satisfies $sig1$

Summary

You have seen how ML programs can be specified and implementations proved to satisfy their specification.

You can learn more about formal specification (in RAISE) and program development in
course 02263 Formal Aspects of Software Engineering