Classic ML and EventML

During this lecture, we are going to learn about a programming language called **Classic ML**.

We will actually use a language called **EventML** (developed by the Nuprl team [CAB+86, Kre02, ABC+06]). EventML is based on Classic ML and a logic called the Logic of Events [Bic09, BC08, BCG11].

We will focus at the Classic ML part of EventML.

### Where does ML come from?

ML was originally designed, as part of a proof system called LCF (Logic for Computable Functions), to perform proofs within \( \text{PP}\lambda \) (Polymorphic Predicate \( \lambda \)-calculus), a formal logical system [GMM+78, GMW79].

By the way, what does ML mean? It means *Meta Language* because of the way it was used in LCF.

We refer to this original version of ML as Classic ML.

Many modern programming languages are based on Classic ML: SML (Standard ML), OCaml (object-oriented programming language), F# (a Microsoft product)... Nowadays ML is often used to refer to the collection of these programming languages.

### Where is ML used?

- **F#** is a Microsoft product used, e.g., in the .NET framework.

- **OCaml** is developed by the INRIA. It has inspired F#.
  The Coq theorem prover is written in OCaml. It has been used in the implementation of Ensemble [Hay98, BCH+00]. It is also used by companies.

- **SML** has formally defined static and dynamic semantics.
  The HOL theorem prover is written in SML. It is nowadays mainly used for teaching and research.
What is Classic ML (or just ML for short)?

ML is a strongly typed higher-order impure functional programming language.

What does it mean?

(Nowadays, ML often refers to a family of languages such as Classic ML, SML, Caml, F#...)

What is ML?

Higher-order.

Functions can do nothing (we will come back to that one):

\( \lambda x. x \)

Functions can take numerical arguments:

\( \lambda x. x + 1 \)

\[
\text{let plus_three } x = x + 3 \; ;
\]

Functions can take Boolean arguments:

\( \lambda a. \lambda b. a \lor b \)

What is ML?

Higher-order.

Functions can also take other functions as arguments.

Function application:

\[
\text{let app = } \lambda f. \lambda x. (f x) \; ;
\]

Function composition:

\[
\text{let comp g h = } \lambda x. (g (h x)) \; ;
\]

Note that, e.g., app can be seen as a function that takes a function \( f \) as input and outputs a function \( \lambda x. (f x) \).

BTW, a function of the form \( \lambda x. e \) (where \( e \) is an expression) is called a \( \lambda \)-expression.

The terms of the forms \( x \) (a variable), \( (e1 e2) \) (an application), and \( \lambda x. e \) (a \( \lambda \)-expression) are the terms of the \( \lambda \)-calculus [Chu32, Bar84].

In 1932, Church [Chu32] introduced a system (that led to the \( \lambda \)-calculus we know) for “the foundation of formal logic”, which was a formal system for logic and functions.
What is ML?

Impure and functional.

**Functional.** Functions are first-class objects: functions can build functions, take functions as arguments, return functions...

**Impure.** Expressions can have side-effects: references, exceptions.

(We are only going to consider the pure part of ML.)

Other functional(-like) programming language: Haskell (pure), SML (impure), F# (impure)...

---

What is ML?

Strongly typed.

What is a type?

A type bundles together “objects” (syntactic forms) sharing a same semantics.

(Types started to be used in formal systems, providing foundations for Mathematics, in the early 1900s to avoid paradoxes (Russell [Rus08]).)

A type system (typing rules) dictates what it means for a program to have a type (to have a static semantics).

What are types good for?

Types are good, e.g., for checking the well-defined behavior of programs (e.g., by restricting the applications of certain functions – see below).

---

What is ML?

Strongly typed.

What else?

**Flexibility.** One of the best things about ML is that is has almost full type inference (type annotations are sometimes required). Each ML implementation has a type inferencer that, given a semantically correct program, finds a type.

This frees the programmer from explicitly writing down types: if a program has a type, the type inferencer will find one.

Given a semantically correct program, the inferred type provides a static semantics of the program.

Consider \( \lambda x. x + 2 \). 2 is an integer. + takes two integers and returns an integer. This means that \( x \) is constrained to be an integer. \( \lambda x. x + 2 \) is then a function that takes an integer and returns an integer.

Can type inferencers infer more than one type? Is each type as good as the others?

In ML it is typical that a program can have several types. The more general the inferred types are the more flexibility the programmer has (we will come back to that once we have learned about polymorphism).

(ML’s type system has principal type but not principal typing [Wel02] (a typing is a pair type environment/type).)
What is ML?
Strongly typed.

Using types, some operations become only possible on values with specific types.

For example, one cannot apply an integer to another integer: integers are not functions. The following does not type check (it does not have a type/a static semantics):

```ml
let fu = (8 6) ;;
```

Another example: using the built-in equality, one cannot check whether a Boolean is equal to an integer. The following does not type check (and will be refused at compile time):

```ml
let is_eq = (true = 1) ;;
```

ML types

**Integer.** For example, 12 + 3 has type `Int`.

**Boolean.** For example, `true` has type `Bool` (stands for the Boolean negation).

**List.** For example, `[1;7;5;3]` has type `Int List`.

**Function type.** For example, `let plus3 x = x + 3;;` has type `Int -> Int`.

**Product type.** For example, `(true , 3)` has type `Bool × Int`.

**Disjoint union type.** For example, `inl (1 + 5)` has type `Int + Int`.

What is ML?
Strongly typed.

What does type check then?

One can apply our `plus_three` function to integers:

```ml
let plus_three x = x + 3 ;;
let fu = plus_three 6 ;;
```

One can test whether two integers are equal:

```ml
let i1 = 11;;
let i2 = 22;;
let is_eq = (i1 = i2) ;;
```

Polymorphism

We claimed that `inl (1 + 5)` has type `Int + Int`. But it can also have type `Int + Bool, Int + Int List, . . .`

For all type `T`, `inl (1 + 5)` has type `Int + T`. This can be represented with a **polymorphic type**: `Int + 'a`, where `'a` is called a **type variable**, meaning that it can be any type.

Let us consider a simpler example: `let id x = x;;`

What’s its type?

The action `id` performs does not depend on its argument’s type. It can be applied to an integer, a Boolean, a function, . . . It always returns its argument. id’s type cannot be uniquely determined. To automatically assign a (monomorphic type) to `id` one would have to make a non-deterministic choice.

Instead, we assign to `id` the polymorphic type: `'a -> 'a`. 
Polymorphism

Formally, this form of polymorphism is expressed using the $\forall$ quantification.

This form of polymorphism is sometimes called **infinitary parametric** polymorphism [Str00, CW85] and $\forall$ types are called type schemes (see, e.g., system F [Gir71, Gir72]).

Polymorphism complicates type inference but does not make it impossible.

Polymorphism allows one to express that a single program can have more than one meaning. Using the $\forall$ quantification, one can express that a single program has an infinite number of meaning, i.e., can be used in an infinite number of ways.

The following function null has type `\texttt{`a Lsit $\rightarrow$ Bool}:

```ml
let null lst =
  case lst of
    [] => true
    x . xs => false
;;
```

Recursion

Another important feature of ML (and functional languages in general) is **recursion**. Recursion allows functions to call themselves. Recursion accomplishes what "while" loops accomplish in imperative languages but in a functional way: functions call functions.

For example, to compute the length of a list, one wants to iterate through the list to count how many elements are in the list. The following function computes the length of a list:

```ml
let rec length lst =
  case lst of
    [] => 0
    x . xs => 1 + length xs
;;
```
Recursion

Given $x$ and $y$, find $q$ (quotient) and $r$ (remainder) such that $x = (q \cdot y) + r$.

The "while" solution:

\[
q := 0; r := x; \\
\text{while } r \geq y \text{ do } q := q + 1; r := r - y; \text{ od} \\
\text{return } (q, r);
\]

The recursive solution:

\[
\text{let } \quot \text{and}_\text{rem} x y = \text{letrec } \text{aux } q r = \\
\quad \text{if } r < y \text{ then } (q, r) \text{ else } \text{aux } (q + 1) (r - y) \text{ in aux } 0 x ; ;
\]

Recursion

Another example: the factorial.

The "while" solution:

\[
f := 1; i := 1; \\
\text{while } i \leq x \text{ do } f := i \cdot f; \text{ i := i + 1; } \text{ od}
\]

The recursive solution:

\[
\text{let } f x = \text{if } x \leq 1 \text{ then } 1 \text{ else } x \cdot f (x - 1); ;
\]

Typing rules

Let us consider the following expression language:

\[
\begin{align*}
\nu & \in \text{Var} \quad \text{(a countably infinite set of variables)} \\
\exp \in \text{Exp} ::= \nu | \exp \exp_2 | \\nu.\exp \mid \text{let } \nu = \exp_1 \text{ in } \exp_2
\end{align*}
\]

Let us consider the following type language:

\[
\begin{align*}
\alpha & \in \text{TyVar} \quad \text{(a countably infinite set of type variables)} \\
\tau \in \text{ITy ::= } & \alpha | \tau_1 \rightarrow \tau_2 \\
\sigma \in \text{ITyScheme ::= } & \forall \{\alpha_1, \ldots, \alpha_n\}. \tau
\end{align*}
\]

Let environments (metavariable $\Gamma$) be partial functions from program variables to type schemes. We write environments as follows:

\[
\{\nu_1 \mapsto \sigma_1, \ldots, \nu_n \mapsto \sigma_n\}.
\]

Let substitutions (metavariable $\text{sub}$) be partial functions from type variables to types. We write substitutions as follows:

\[
\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}.
\]

Typing rules

the function $\text{fv}$ computes the set of free type variables in a type or in a type environment.

We define the domain of an environment as follows:

\[
\text{dom}\{\nu_1 \mapsto \sigma_1, \ldots, \nu_n \mapsto \sigma_n\} = \{\alpha_1, \ldots, \alpha_n\}.
\]

We write substitution in a type as follows: $\tau[\text{sub}]$.

Let the instantiation of a type scheme be defined as follows:

\[
\tau \prec \forall \{\alpha_1, \ldots, \alpha_n\}. \tau' \iff \exists \tau_1, \ldots, \tau_n. (\tau = \tau'[\{\alpha_i \mapsto \tau_i \mid i \in \{1, \ldots, n\}\}])
\]

We also define a function to “merge” environments:

\[
\Gamma_1 + \Gamma_2 = \{\alpha \mapsto \tau \mid \Gamma_2(\alpha) = \tau \text{ or } (\Gamma_1(\alpha) = \tau \text{ and } \alpha \not\in \text{dom}(\Gamma_2))\}
\]
Typing rules

\[\begin{align*}
\tau &\prec \Gamma(\text{vid}) \\
\text{let } v &: (\Gamma, \tau) &\text{exp}_1 &: (\Gamma, \tau_1 \rightarrow \tau_2) &\text{exp}_2 &: (\Gamma, \tau_1) \\
\text{exp}_1 \text{ exp}_2 &: (\Gamma, \tau_2) \\
\text{exp} &: (\Gamma + \{v \mapsto \tau\}, \tau') \\
\text{\backslash v. exp} &: (\Gamma, \tau \rightarrow \tau') \\
\text{exp} &: (\Gamma, \tau) &\text{exp}_2 &: (\Gamma + \{v \mapsto (fv(\tau) \setminus fv(\Gamma)).\tau\}, \tau') \\
\end{align*}\]

References I


References II


References III