Classic ML

CS 5860 - Introduction to Formal Methods

Vincent Rahli

Nuprl team

Cornell University

September 6, 2011

Classic ML an EventML

Where does ML come from?

Where is ML used?

What is Classic ML?

ML types

Polymorphism

Recursion

Typing rules

Type inference

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 PPλ (Polymorphic Predicate λ-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):

```
\x. x
```

Functions can take numerical arguments:

```
\x. x + 1
```

```
let plus_three x = x + 3 ;;
```

Functions can take Boolean arguments:

```
\a. \b. a or b
```

What is ML?

Higher-order.

Functions can also take other **functions as arguments**.

Function application:

```
let app = \f. \x. (f x);;
```

Function composition:

```
let comp g h = \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 (\x. (f x)).

What is ML?

Higher-order.

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

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

In 1932, Church [Chu32] introduced a system (that led to the λ-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 \x. x + 2. 2 is an integer. + takes two integers and returns an integer. This means that x is constrained to be an integer. \x. x + 2 is then a function that takes an integer and returns an integer.

What is ML?

Strongly typed.

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):

```
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):

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

What is ML?

Strongly typed.

What *does* type check then?
one can apply our plus_three function to integers:

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

One can test whether two integers are equal:

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

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.

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

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 'a List $\rightarrow$ Bool:

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

Polymorphism

let declarations allow one to define polymorphic functions while lambda expression do not. For example, the following piece of code is typable:

```
let x = (\x. x) in (x 1, x true)
```

However, the following piece of code is not typable:

```
(\x. (x 1, x true)) (\x. x)
```

In the first example, the two last x's stand for the identity function for two different types. In the second example, the two bound x's in \x. (x 1, x true) have to be the same function.

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:

```
letrec length lst =
   case lst of [] => 0
            of x . xs => 1 + length xs;;
```

Recursion

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

The "while" solution:

```
q := 0; r := x;
while r >= y do q := q + 1; r := r - y; od
return (q, r);
```

The recursive solution:

```
let quot_and_rem x y =
    letrec aux q r =
      if r < y then (q, r)
      else aux (q + 1) (r - y)
    in aux 0 x ;;
```

Recursion

Another example: the factorial.

The "while" solution:

```
f := 1; i := 1;
while i <= x do
  f := i * f;;
  i := i + 1;;
od
```

The recursive solution:

```
let f x = if x <= 1
          then 1
          else x * f (x - 1);;
```

Typing rules

Let us consider the following expression language (sometimes referred to as core ML:

$$\begin{array}{ll} v & \in \mathrm{Var} \quad \text{(a countably infinite set of variables)} \\ exp & \in \mathrm{Exp} ::= v \mid exp_1\ exp_2 \mid \backslash v\,.\,exp \mid \texttt{let}\ v\ \texttt{=}\ exp_1\ \texttt{in}\ exp_2 \end{array}$$

Let us consider the following type language:

$$\begin{array}{lll} a \in \mathrm{TyVar} & & \text{(a countably infinite set of type variables)} \\ \tau \in \mathrm{ITy} & ::= a \mid \tau_1 \ \rightarrow \ \tau_2 & \\ \sigma \in \mathrm{ITyScheme} & ::= \forall\{a_1, \ldots, a_n\}.\tau & \end{array}$$

Let environments (metavariable Γ) be partial functions from program variables to type schemes. We write environments as follows: $\{v_1 \mapsto \sigma_1, \ldots, v_n \mapsto \sigma_n\}$.

We sometimes write $a \mapsto \tau$ for $a \mapsto \forall\varnothing.\tau$.

Typing rules

the function 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: $\mathsf{dom}(\{v_1 \mapsto \sigma_1, \ldots, v_n \mapsto \sigma_n\}) = \{a_1, \ldots, a_n\}$.

Let substitutions (metavariable *sub*) be partial functions from type variables to types. We write substitutions as follows: $\{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$.

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

Typing rules

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

$$\begin{aligned} & \tau \prec \forall\{a_1, \ldots, a_n\}.\tau' \\ \iff & \exists \tau_1, \ldots, \tau_n.\ (\tau = \tau'[\{a_i \mapsto \tau_i \mid i \in \{1, \ldots, n\}\}]) \end{aligned}$$

We also define a function to "merge" environments:

$$\begin{aligned} & \Gamma_1 + \Gamma_2 \\ = & \{a \mapsto \tau \mid \Gamma_2(a) = \tau \text{ or } (\Gamma_1(a) = \tau \text{ and } a \notin \mathrm{dom}(\Gamma_2))\} \end{aligned}$$

Typing rules

(A variant of Damas and Milner's type system, sometimes referred to as the Hindley-Milner type system and therefore often called DM or HM.)

$$\frac{\tau \prec \Gamma(vid)}{v : \langle \Gamma, \tau \rangle}$$

$$\frac{exp_1 : \langle \Gamma, \tau_1 \ \rightarrow \ \tau_2 \rangle \quad exp_2 : \langle \Gamma, \tau_1 \rangle}{exp_1 \ exp_2 : \langle \Gamma, \tau_2 \rangle}$$

$$\frac{exp : \langle \Gamma + \{v \mapsto \tau\}, \tau' \rangle}{\backslash v \,.\, exp : \langle \Gamma, \tau \ \rightarrow \ \tau' \rangle}$$

$$\frac{exp : \langle \Gamma, \tau \rangle \quad exp_2 : \langle \Gamma + \{v \mapsto \forall(\mathrm{fv}(\tau) \setminus \mathrm{fv}(\Gamma)).\tau\}, \tau' \rangle}{\texttt{let}\ v = exp_1\ \texttt{in}\ exp_2 : \langle \Gamma, \tau' \rangle}$$

Typing rules

For example:

Let $\Gamma = \{\mathsf{f} \mapsto (a_1 \to a_2), \mathsf{g} \mapsto (a_2 \to a_3), \mathsf{v} \mapsto a_1\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{g} : \langle \Gamma, a_2 \to a_3 \rangle} \qquad \dfrac{\dfrac{}{\mathsf{f} : \langle \Gamma, a_1 \to a_2 \rangle} \qquad \dfrac{}{\mathsf{v} : \langle \Gamma, a_1 \rangle}}{\mathsf{f\ v} : \langle \Gamma, a_2 \rangle}}{\mathsf{g\ (f\ v)} : \langle \Gamma, a_3 \rangle}}{\backslash \mathsf{v} . \mathsf{g\ (f\ v)} : \langle \{\mathsf{f} \mapsto (a_1 \to a_2), \mathsf{g} \mapsto (a_2 \to a_3)\}, a_1 \ \to \ a_3 \rangle}}{\backslash \mathsf{g} . \backslash \mathsf{v} . \mathsf{g\ (f\ v)} : \langle \{\mathsf{f} \mapsto (a_1 \to a_2)\}, (a_2 \to a_3) \ \to \ a_1 \ \to \ a_3 \rangle}}{\backslash \mathsf{f} . \backslash \mathsf{g} . \backslash \mathsf{v} . \mathsf{g\ (f\ v)} : \langle \varnothing, (a_1 \to a_2) \ \to \ (a_2 \to a_3) \ \to \ a_1 \ \to \ a_3 \rangle}$$

Typing rules

For example:

Let $\Gamma = \{\mathrm{id} \mapsto \forall\{a\}.a \rightarrow a\}$.
Let $\tau = a_1 \rightarrow a_1$

$$\dfrac{\dfrac{\overline{\mathrm{id} : \langle\{\mathrm{id} \mapsto a\}, a\rangle}}{\backslash\mathrm{id}\,.\,\mathrm{id} : \langle\varnothing, a \rightarrow a\rangle} \qquad \dfrac{\overline{\mathrm{id} : \langle\Gamma, \tau \rightarrow \tau\rangle} \qquad \overline{\mathrm{id} : \langle\Gamma, \tau\rangle}}{\mathrm{id}\ \mathrm{id} : \langle\Gamma, \tau\rangle}}{\texttt{let}\ \mathrm{id}\ \texttt{=}\ \backslash\mathrm{id}\,.\,\mathrm{id}\ \texttt{in}\ \mathrm{id}\ \mathrm{id} : \langle\varnothing, \tau\rangle}$$

Type inference

Type inference vs. **type checking**. Let S be a type system:

- Type checking: given a (closed) expression *exp* and a type τ, a type checker checks that *exp* has type τ w.r.t. S.
- Type inference: given a (closed) expression *exp*, a type inferencer infers a type τ such that *exp* has type τ w.r.t. S, or fails if no such type exists.

Classic ML has **decidable** type inference: there exists an algorithm that given an expression *exp*, infers a type for *exp* which is valid w.r.t. the static semantics of Classic ML.

Classic ML seats between the simply typed λ-calculus [Bar92] (no polymorphism) and system F [Gir71, Gir72] (undecidable type inference).

Type inference

Type inference for Classic ML is exponential in theory. Many algorithms are **efficient in practice** (quasi-linear time under some assumptions).

Milner [Mil78] proposed a type inference algorithm, called the W algorithm, for an extension of core ML and proved it sound.

Damas (Milner's student) and Milner [DM82] later proved the completeness of W.

Type inference

The W algorithm takes two inputs: a type environment Γ and an expression *exp*; and returns two outputs: a type substitution s and a type τ; such that *exp* has type τ in the environment $\Gamma[s]$ w.r.t. the type system presented above.

W is defined by induction on the structure of its expression parameter.

Type inference

Remark 1: These inference algorithms use **first-order unification** [MM82, BN98].

Given an application exp_1 exp_2, W produces, among other things, τ_1 a type for exp_1, and τ_2 a type for exp_2. A unification algorithm is then used to unify τ_1 and $\tau_2 \rightarrow a$ where a is a "fresh" type variable (meaning that τ_1 has to be a function that takes an argument of type τ_2).

Remark 2: Many algorithms have been designed since the W. In some algorithms constraint generation and unification interleave [Mil78, DM82, LY98, McA99, Yan00], in others the constraint generation and constraint solving phases are separated [OSW99, Pot05, PR05].

Remark 3: EventML's inferencer is constraint based (second category).

Type inference

Example:

$$\mathsf{let}\ \ \mathrm{plus1}\ \ x = x + 1\ \mathsf{in}\ \mathrm{plus1}\ \ 3$$

- $+$ is a function that takes two Ints and returns an Int.
- 1 and x are constrained to be Ints.
- plus1 is constrained to be a function that takes an Int and returns an Int.
- plus1 3 is an Int.
- Therefore the whole expression is an Int.

Type inference

Example:

```
let app f x = f x in app (\x.x + 1) 3
```

- If x has type 'a then f is constrained to have type 'a $\rightarrow$ 'b.
- app has polymorphic type ('a $\rightarrow$ 'b) $\rightarrow$ 'a $\rightarrow$ 'b.
- + is a function that takes two Ints and returns an Int.
- 1 and x are constrained to be Ints.
- The function \x.x + 1 is constrained to have type Int $\rightarrow$ Int and 3 is an Int.
- An instance of app's type is (Int $\rightarrow$ Int) $\rightarrow$ Int $\rightarrow$ Int, where both 'a and 'b are instantiated to Int. This is the type of app's second occurrence.
- Therefore the whole expression is an Int.

Type inference

Example:

let id x = x in id id

- id has polymorphic type 'a → 'a. Each instance of id's type is a functional type.
- id's first bound occurrence is a function that takes a function as parameter.
- Therefore, id's first bound occurrence's type is an instance of 'a → 'a such that 'a is substituted by a functional type.
- That functional type has to be an instance of 'a → 'a.
- For example, we can assign ('b → 'b) → ('b → 'b) to id's first bound occurrence, and 'b → 'b to id's second bound occurrence.
- Therefore, the whole expression has type 'b → 'b.

Type inference

Example:

```
let quot_and_rem x y =
    letrec aux q r =
      if r < y then (q, r)
      else aux (q + 1) (r - y)
    in aux 0 x ;;
```

- Because + and − both take Ints and return Ints, q, r, and y are constrained to be Ints.
- aux's first bound occurrence is constrained to be a function that takes two Int's and returns a pair of Int's (aux has type Int → Int → (Int ∗ Int)).
- Because aux is applied to 0 and x in the last line, x is constrained to be an Int.
- quot_and_rem has type Int → Int → (Int ∗ Int).

References I

Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran.
Innovations in computational type theory using nuprl.
J. Applied Logic, 4(4):428–469, 2006.

Henk P. Barendregt.
The Lambda Calculus: Its Syntax and Semantics.
North-Holland, revised edition, 1984.

Henk P. Barendregt.
Lambda calculi with types.
In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. Oxford University Press, Inc., New York, NY, USA, 1992.

Mark Bickford and Robert L. Constable.
Formal foundations of computer security.
In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.

Mark Bickford, Robert Constable, and David Guaspari.
Generating event logics with higher-order processes as realizers.
Technical report, Cornell University, 2011.

Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels.
The Horus and Ensemble projects: Accomplishments and limitations.
In *In DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–160. IEEE Computer Society Press, 2000.

References II

Mark Bickford.
Component specification using event classes.
In Grace A. Lewis, Iman Poernomo, and Christine Hofmeister, editors, *CBSE*, volume 5582 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2009.

Franz Baader and Tobias Nipkow.
Term rewriting and all that.
Cambridge University Press, New York, NY, USA, 1998.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith.
Implementing mathematics with the Nuprl proof development system.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

Alonzo Church.
A set of postulates for the foundations of logic.
The Annals of Mathematics, 33(2):346–366, April 1932.

Luca Cardelli and Peter Wegner.
On understanding types, data abstraction, and polymorphism.
ACM Computing Surveys, 17(4):471–522, 1985.

Luis Damas and Robin Milner.
Principal type-schemes for functional programs.
In *POPL82*, pages 207–212, New York, NY, USA, 1982. ACM.

Jean-Yves Girard.
Une extension de l'interprétation de Gödel à l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types.
In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, 1971.

References III

Jean-Yves Girard.
Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.
PhD thesis, Universite de Paris VII, 1972.

Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth.
A metalanguage for interactive proof in LCF.
In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130, New York, NY, USA, 1978. ACM.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth.
Edinburgh LCF: A Mechanised Logic of Computation., volume 78 of *Lecture Notes in Computer Science*.
Springer-Verlag, 1979.

Mark Hayden.
The Ensemble System.
PhD thesis, Cornell University, Department of Computer Science, 1998.
Technical Report TR98-1662.

Christoph Kreitz.
The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide.
Cornell University, Ithaca, NY, 2002.
`http://www.nuprl.org/html/02cucs-NuprlManual.pdf` .

Oukseh Lee and Kwangkeun Yi.
Proofs about a folklore let-polymorphic type inference algorithm.
ACM Transanctions on Programming Languages and Systems, 20(4):707–723, jul 1998.

References IV

Bruce J. McAdam.
On the unification of substitutions in type inference.
In Kevin Hammond, Antony J. T. Davie, and Chris Clack, editors, *10th Int'l Workshop, IFL'98*, volume 1595 of *LNCS*, pages 137–152. Springer, 1999.

Robin Milner.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences, 17(3):348–375, December 1978.

Alberto Martelli and Ugo Montanari.
An efficient unification algorithm.
ACM Trans. Program. Lang. Syst., 4(2):258–282, 1982.

Martin Odersky, Martin Sulzmann, and Martin Wehr.
Type inference with constrained types.
Theor. Pract. Object Syst., 5(1):35–55, 1999.

François Pottier.
A modern eye on ML type inference: old techniques and recent developments.
Lecture notes for the APPSEM Summer School, September 2005.

François Pottier and Didier Rémy.
The essence of ML type inference.
In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

Bertrand Russell.
Mathematical logic as based on the theory of types.
American Journal of Mathematics, 30(3):222–262, 1908.

References V

Christopher Strachey.
Fundamental concepts in programming languages.
Higher Order Symbol. Comput., 13(1-2):11–49, 2000.

J. B. Wells.
The essence of principal typings.
In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming, 29th Int'l Colloq., ICALP 2002*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

Jun Yang.
Explaining type errors by finding the source of a type conflict.
In *SFP'99: Selected papers from the 1st Scottish Functional Programming Workshop*, pages 59–67, Exeter, UK, UK, 2000. Intellect Books.