Milner's legacy
The ML language

Camilo Rueda [1]

[1] Universidad Javeriana-Cali

PUJ 2010

Robin Milner

A life of “theory becoming practice”

Agenda

- Introduction: functional models
- ML, the model
- ML, the language
- Conclusions

Functional models

- Functions as first-class citizens
- basic computation step: function application
- terms:
  - variable: x
  - function: $\lambda x.x\ y$
  - function application: $(\lambda x.x\ y)z$
- Semantic model: expression rewriting
  - α equivalence: changing bound variables
  - term reduction: $(\lambda x.s)r \rightarrow s[r/x]$
- issues:
  - call-by-name(lazy evaluation)
  - call-by-value (eager evaluation)
  - confluence

Functional models: adding types

- A set of basic types: $\mathcal{G}$
- basic terms (or "pre-terms"). These include things we do not want, such as

  $\lambda x : \tau.\ x\ x$
- Type judgements: disallow the above
- The types: $\tau := \mathcal{G} \mid \tau \to \tau$
- The pre-terms: $\mathcal{T} := V \mid \mathcal{T}\ \mathcal{T} \mid \lambda V : \tau.\mathcal{T}$
- The context (x_i has type T_i): $x_1 : T_1, ..., x_i : T_i, ...x_m : T_m$
- The judgements:

$$\frac{}{x_1 : T_1, ..., x_m : T_m \vdash x_i : T_i}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash r : A}{\Gamma \vdash f\ r :\ B} \qquad \frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s :\ A \to B}$$

The ML model (untyped)

identifiers::=	
x	variables
c	constants←new
m	memory location←new
expressions ::=	
x	identifier
$\lambda x.t$	function
$t\ t$	application
let $z = t$ **in** t	local definition←new
Evaluation contexts::=	
[]	empty context
$\mathcal{E}\ t$	left side of application
$v\ \mathcal{E}$	right side of application
let $z = \mathcal{E}$ **in** t	local def

The ML model: semantics

$(\lambda z.t)v \to [z \mapsto v]t \quad v$ is a constant value

let $z = v$ **in** $t \to [z \mapsto v]t$

$$\frac{(t,u) \to^{\delta} (t',u')}{(t,u) \to (t',u')} \qquad \to^{\delta}\text{: reduction by constants,e.g. built-in functions}$$

$$\frac{(t,u) \to (t',u') \quad dom(u'')\#dom(u') \quad ran(u'')\#dom(u' \setminus u)}{(t,uu'') \to (t',u'u'')}$$

u is a *store*: mapping from memory locations to values

The ML model: semantics

$(\lambda z.t)v \to [z \mapsto v]t \quad v$ is a constant value

let $z = v$ **in** $t \to [z \mapsto v]t$

$$\frac{(t,u) \to^{\delta} (t',u')}{(t,u) \to (t',u')} \qquad \to^{\delta}\text{: reduction by constants,e.g. built-in functions}$$

$$\frac{(t,u) \to (t',u') \quad dom(u'')\#dom(u') \quad ran(u'')\#dom(u' \setminus u)}{(t,uu'') \to (t',u'u'')}$$

u is a *store*: mapping from memory locations to values

let $z = E$ **in** S is the same as $(\lambda z.\ S)\ E$
so, what is new?

The ML model: summary

- First-class functions
- Data structures (i.e. constants) built out of sums and products
- Mutable memory cells (references)
- call-by-value semantics

and..

A type system based on the simply typed λ*calculus*
with a simple form of polymorphism
introduced by let declarations

The ML model: type system

Hindley-Milner, 1978

Key contributions:

- Restricted, practical polymorphism
- A decidable type inference
- A type inference algorithm efficient in practical cases

The ML type system: basics

- Type variable: a name (different from program vars) standing for a type
- A type is a first-order term:
  1. a type variable
  2. a function type $T \rightarrow T$
- Substitution: function assigning a type to (some) type variables, $[\overrightarrow{X} \mapsto \overrightarrow{T}]$
- Type scheme: an object S of the form $\forall \overrightarrow{x} : T$
- Type instance: any type of the form $[\overrightarrow{X} \mapsto \overrightarrow{T}]T$ is an instance of $\forall \overrightarrow{x} : T$
- Type environment (Γ): a function assigning a type scheme to (some) program identifiers
  (every type T is a type scheme $\forall \emptyset . T$)

The ML type system: rules

$$\frac{\Gamma(x) : S}{\Gamma \vdash x : S}$$

$$\frac{\Gamma \vdash t_1 : S \quad \Gamma; z : S \vdash t_2 : T}{\Gamma \vdash \textbf{let}\ z = t_1\ \textbf{in}\ t_2\ :\ T}$$

$$\frac{\Gamma; z : T \vdash t : T'}{\Gamma \vdash \lambda z.t :\ T \to T'}$$

$$\frac{\Gamma \vdash t : T \qquad \overrightarrow{X} \# ftv(\Gamma)}{\Gamma \vdash t :\ \forall \overrightarrow{X}.T}$$

$$\frac{\Gamma \vdash t_1 : T \to T' \quad \Gamma \vdash t_2 :\ T}{\Gamma \vdash t_1\ t_2 :\ T'}$$

$$\frac{\Gamma \vdash t : \forall \overrightarrow{X}.T}{\Gamma \vdash t : [\overrightarrow{X} \mapsto \overrightarrow{T}]T}$$

Types are stratified. Type schemes are not allowed deep inside expressions

Example of a type derivation

$$(2)\dfrac{(4)\dfrac{(1)\overline{z_1 : X \vdash z_1 : X} \quad (1)\overline{z_1 : X; z_2 : X \vdash z_2 : X}}{z_1 : X \vdash \textbf{let } z_2 = z_1 \textbf{ in } z_2 \ : \ X}}{\emptyset \vdash \lambda z_1.\textbf{let } z_2 = z_1 \textbf{ in } z_2 \ : \ X \to X}$$

Type system: properties

Soundness:

Well-typed programs do not go wrong

Inference:

There is an algorithm that,
given a closed environment Γ and a program t,
tells whether t is well-typed w.r.t Γ and, if so,
produces a principal type scheme S

A principal type scheme is such that:

- it is valid. i.e. $\Gamma \vdash t : S$ holds
- it is most general:
  every judgement $\Gamma \vdash t : S'$
  follows from $\Gamma \vdash t : S$ by (6) and (5)

The inference algorithm: the idea

- Generate a set of type equations

A function f, mapping a pair (Γ, t) to a type T Γ: a type environment, t : an expression
(1) $f(\Gamma, v) = \tau$, where $(v \mapsto \tau) \in \Gamma$
(2) $f(\Gamma, g\ e) = \tau$, where $f(\Gamma, g) = \tau_1 \to \tau$, and $f(\Gamma, e) = \tau_1$
(3) $f(\Gamma, \lambda v : \tau.e) = \tau \to f(\Gamma_1, e)$, where $\Gamma_1 = \Gamma \cup \{v \mapsto \tau\}$

- Unify

A function U, from type equations to substitutions
(1) $U\ \emptyset = \mathbf{id}$
(2) $U\,([\alpha = T] \cup C) = U\,(C') \cdot (\alpha \mapsto T)$, where C' is C with the substitution $\alpha \mapsto T$ applied to it
(3) $U\,([S \to S' = T \to T'] \cup C) = U\,(\{[S = T], [S' = T']\} \cup C)$

The inference algorithm: strategy

A nice scheme based on constraints (Damas-Milner, 1982)

- Type inference problem: Given a type environment Γ and an expression t, find a type T such that $\Gamma \vdash t : T$ holds
- Constraints type inference problem: given a type environment Γ, an expression t, and a type T, find a satisfiable constraint C such that $C, \Gamma \vdash t : T$ holds

Reducing a type inference problem to a constraint problem:

produce a constraint C that is both sufficient and necessary for $C, \Gamma \vdash t : T$ to hold

The ML language

"The definition of standard ML"
Milner-Tofte-Harper, 1990

The ML language: features

- Safe: well-typed programs do not go wrong
- Modular: supports modules, called *structures*, and interfaces, called *signatures*.
- Functional: first-class functions
- Strict: call-by-value semantics
- Polymorphic: data-type and function polymorphism
  $e.g. list\ 'a \quad fun\ \ I(x :' a) :' a = x \quad (identityfunction)$
- Exception handling
- Inmutable data types: once initialized, the structure does not change
- Udatable references
- Abstract data types: support for information hiding
- Parametric modules: a *functor* taking the signature of another module as argument

First-class functions in ML: a dictionary

```
exception SORRY          /* exception definition
> con SORRY : exn

val initdic = fn(key) => raise SORRY;
>  val initdic = fn : 'a→'b

fun update(cont, key)(dict) =
  fn(key1) => if key = key1 then cont else dict(key1)
>  val update = fn : ('a * "b)→ (("b→ 'a)→ ("b→ 'a))

fun delete(key)(dict) =
  fn(key1) =>
    if key = key1 then raise SORRY else dict(key1);
>  val delete = fn : ''a→ (((''a→ 'b)→ (''a→ 'b))
```

First-class functions in ML: a dictionary

```
val  engfrench =
    (update("souris","ant")
     (update("chat","cat")
      (update("cheval", "horse")
        initdic)))
> val engfrench = fn : string→ string
```

Inmutable data types: type constructors

- Cartesian product: $(\ldots * \ldots)$ Function space: $(\ldots \rightarrow \ldots)$

Disjoint union: ensuring unique types

$A \mid B =^{def} A \times \{(A, B, l)\} \cup B \times \{(A, B, r)\}$
e.g. let $A = \{1, 2\}$ $B = \{3, 4\}$, $A \cup B = \{1, 2, 3, 4\}$
$A\|B = \{(1, (A, B, l)), (2, (A, B, l)), (3, (A, B, r)), (4, (A, B, r))\}$

datatype $type_0 = fun_1$ *of* $type_1$
$\mid fun_2$ *of* $type_2$;

Inmutable data types: type constructors (2)

Recursive type definitions:

```
datatype tree = empty
              | mktree of real * tree * tree;

fun sum(empty) = 0,0
      | sum(mktree(r, t1, t2) = r + sum(t1) + sum(t2)
> val sum = fn : tree → real

mktree(5.0,mktree(1.0,empty,empty))
```

Polymorphic types

```
datatype ('a)tree = empty
                  | mktree of ('a) * ('a)tree * ('a)tree;

> datatype 'a tree = empty | mktree of 'a *('a tree)*('a tree)
  con mktree = fn : ('a *('a tree)*('a tree))→('a tree)
  con empty = empty : 'a tree
```

Polymorphic functions

```
fun apply(f,x) = f(x) ;
> val apply fn : ((’a → ’b)* ’a )→ ’b
```

If “being of type” is the same as “belongs to the set”, then every object has a unique (concrete) type. What is the type of the above function?

Polymorphic object:

A function from types to objects

Polymorphic functions

```
fun apply(f,x) = f(x) ;
> val apply fn : ((’a → ’b)* ’a )→ ’b
```

If “being of type” is the same as “belongs to the set”, then every object has a unique (concrete) type. What is the type of the above function?

depends on the context:

$apply((fn(n) => n + 1), 2)$

$apply(sin, 3{,}14)$

Polymorphic object:

A function from types to objects

Iterators

```
fun itnat n f a = if n = 0 then a else f(itnat (n − 1) f a ;

fun itlist [] f b = b
  | itlist (a :: lis) f b =
        f (a, itlist lis f b)
```

Abstract data types

defines an environment or module

```
structure Stack =
  struct
    exception no_entry;
    type ('a) stack = int * (int -> 'a) ;
    val empty = (0, (fn(k : int) => raise no_entry));
    fun push(a, (n, f)) =
          (n + 1, fn(n1) => if n1 = n + 1 then a else f(n1) );
    fun top(n, f) = f(n) ;
    fun pop(n, f) =
          if n = 0 then (n, f) else (n - 1, f);
end ;
```

```
let open Stack in top(push(5, empty)) end ;
```

Interfaces

Signature

```
signature DICT =
  sig
    exception not_found;
    type key;
    type cont;
    type dict;
    val initial : dict;
    val ins :
      key * cont * dict → dict;
    val find :
      key * dict → cont;
end ;
```

module

```
structure Dict1 : DICT =
 struct
   exception not_found;
   type key = int;
   type cont = real;
   datatype
    dict = empty
   | node of
      (key * cont) * dict * dict;
   val initial =
    fn(k) => raise not_found;
   fun ins(k, c, d) = ...
end ;
```

So, what is next?

As a way of conclusion...

- Functional programming:
  ML + lazy evaluation + more powerful pattern matching: Haskell
- Object-oriented programming:
  ML + objects: Ocaml
- Concurrent programming:
  ML + concurrency: Earlang, Haskell, Oz(?)
- Distributed programming:
  ML + distribution: Pict(?)← Milner