Modular implicits for OCaml
how to assert success

Jacques Garrigue
Nagoya University

Frédéric Bour
Sponsored by Jane Street LLC

Gallium Seminar, 14-03-2016

Modular implicits for OCaml

- A proposal by Bour, White and Yallop [2014/2015]
- Apply Scala implicits to OCaml
- Modules and functors can be made implicit
- Can express basic Haskell type classes, and many extensions
- Static implicit scope, no global uniqueness restriction
- Emphasis on non-ambiguity of implicit resolution

Basic type class

```
module type Show = sig type t val show : t -> string end
let show {M : Show} x = M.show x
val show : {M : Show} -> M.t -> string
implicit module Show_int = struct
  type t = int
  let show = string_of_int
end
implicit module Show_float = struct
  type t = float
  let show = string_of_float
end
show 1                                          (* uses Show_int *)
- : string = "1"
show 1.0                                      (* uses Show_float *)
- : string = "1.0"
```

Implicit abstraction

Abstraction is explicit, and registers an implicit module.

```
let print {S : Show} (x : S.t) =
  print_endline (show x)                          (* uses S *)
val print : {S : Show} -> S.t -> unit

print (1+1)                                (* uses Show_int *)
2
print (1.0 + 2.0)                        (* uses Show_float *)
3.0
```

Implicit functor = evidence construction

```
implicit module Show_pair {A : Show} {B : Show} = struct
  type t = A.t * B.t
  let show (a,b : t) = "(" ^ A.show a ^ "," ^ B.show b ^ ")"
end

implicit module Show_list {X : Show} = struct
  type t = X.t list
  let show (xs : t) =
      "[" ^ String.concat "; " (List.map X.show xs) ^ "]"
end

print [("hello", 1); ("world", 2)]
              (* uses Show_list(Show_pair(Show_string,Show_int)) *)
                    (*  : Show with type t = (string * int) list *)
[("hello", 1); ("world", 2)]
```

Constructor classes

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end

let return {M : Monad} x = M.return x
val return : {M : Monad} -> 'a -> 'a M.t

let (>>=) {M : Monad} m k = M.bind m k
val ( >>= ) : {M : Monad} -> 'a M.t -> ('a -> 'b M.t) -> 'b M.t

(return 3 : int list)
- : int list = [3]
```

Associated types

An implicit signature with multiple types can encode a relation between types.

```
module type Array = sig
  type t and elt
  val create : int -> elt -> t
  val get : t -> int -> elt
  val set : t -> int -> elt -> unit
end

implicit module Array_Bytes = struct
  type t = Bytes.t and elt = char
  let create = Bytes.make
  let get = Bytes.get
  let set = Bytes.set
end
```

Implicit arguments

Implicits can also be defined only locally, to express Haskell-style implicit arguments.

```
module type S = sig type state val gensym : unit -> int end
let gensym {M : S} () = M.gensym ()
val gensym : {M : S} -> unit -> int

let name_val {M : S} expr body =           (* M becomes implicit *)
  let name = gensym () in                           (* uses M *)
  Let (name, expr, body name)
val name_val : {M : S} -> expr -> (int -> expr) -> expr

module G : S = ...                       (* not declared as implicit *)
name_val {G} (Cst 3) (fun x -> Var x)
- : expr = Let (7, Cst 3, Var 7)
```

Virtual implicit arguments *new!*

If a module contains only type members, and we can infer all its type definitions, then we do not need to declare a concrete definition for it.

```
implicit module Show_poly_list {virtual X : sig type t end} =
  struct
    type t = X.t list
    let show (l : t) =
      "[" ^ string_of_int (List.length l) ^ " elements]"
  end

print [1;2;3]
[3 elements]
```

The problem at hand

- We want to find all possible combinations of implicit modules and implicit functor applications that match an implicit function application.
- If there is only one solution, use this (extended) module path as argument to the implicit function, and check again its correctness.
- If there is no solution, or if there are more than one solution (ambiguity), the search fails.

Specification of the search

An intermediate state of the search can be seen a partial extended module path, with named holes

$$\mathtt{M} = \mathtt{Show\_list}(\mathtt{Show\_pair}(\mathtt{A})(\mathtt{B}))$$

or, seen as a tree,

$$\dfrac{\dfrac{\mathtt{A} : \mathtt{Show} \qquad\qquad\qquad \mathtt{B} : \mathtt{Show}}{\mathtt{N} = \mathtt{Show\_pair}(\mathtt{A})(\mathtt{B}) : \mathtt{Show}}}{\mathtt{M} = \mathtt{Show\_list}(\mathtt{N}) : \mathtt{Show}}$$

together with a constraint which these holes should satisfy:

$$\mathtt{M.t} = (\mathtt{string} * \mathtt{int})\mathtt{list} \wedge \mathtt{N.t} = \mathtt{string} * \mathtt{int} \wedge \mathtt{A.t} = \mathtt{string} \wedge \mathtt{B.t} = \mathtt{int}$$

The constraints at work

In order to express module subtyping, we need

- type equalities (including universal type variable, for definitions)
- instantiation constraints, for value fields

The constraints contain both

- flexible type constructors, corresponding to types declared in implicit module parameters of implicit functions and functors, and also type variables of the implicit function application
- fixed type constructors and universal variables, corresponding to types defined outside, and parameters of type declarations

Incompleteness

- Implicit functors may express complex relations between their input and output.
- Whether proof search will terminate seems undecidable in general.
- We introduce a termination criterion for applying an implicit functor at a leaf in the tree.
- If the criterion is not satisfied, we block it until constraints introduced by other nodes make it satisfiable.
- If some blocked nodes remain, we must assume ambiguity.

Higher-order unification

Parametric flexible type constructors hint at the need for higher-order unification.

$$\texttt{(return 3 : int list)} \Rightarrow \texttt{int M.t} = \texttt{int list}$$

This has two solutions, `type 'a t = int list` and `type 'a t = 'a list`.

Since higher-order unification is known to be undecidable, we shall stick to the simpler approach of dynamic higher-order pattern unification, *i.e.* only substitute a parametric flexible when all its parameters are distinct type variables.

Trying $\texttt{M} = \texttt{Monad\_list}$, *i.e.* `type 'a t = 'a list`:

$$\frac{\dfrac{\alpha\ \texttt{M.t} = \alpha\ \texttt{list} \wedge \texttt{int M.t} = \texttt{int list}}{\alpha\ \texttt{M.t} = \alpha\ \texttt{list} \wedge \texttt{int list} = \texttt{int list}}}{\alpha\ \texttt{M.t} = \alpha\ \texttt{list}}$$

Principality of constraint solving

In order to make the result of proof search independent of search order, we need to have constraint solving commute with node filling.

In particular, we need the following 2 properties.

- Deriving a contradiction from a constraint is monotonous
  Ambiguity reduces with the addition of constraints

- Resolving flexible type definitions into rigid ones is monotonous
  Satisfiable termination constraints and solvable virtual implicit parameters increase with the addition of constraints

Model theoretic approach

– Usually proofs of unification model solutions by substitutions

– However, we need to delay constraints on parametric flexible types, which may have no concrete solution

– Intuitive solution: define a model where parametric flexible types are relations rather than functions

– Consequence: parametric flexible types should not appear in equations, as they would break transitivity, or incur a high cost
Example:

$$\texttt{int t} = \texttt{int} \wedge \texttt{int t} = \texttt{bool}$$

Types and constraints

Types

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{parametric variable} \\
 & \mid & t(\vec{\tau}) & \text{datatype constructors} \\
 & \mid & v & \text{variable} \\
v & ::= & \varphi & \text{flexible type} \\
 & \mid & x & \text{unification variable}
\end{array}
$$

Constraints

$$
\begin{array}{rcll}
\psi & ::= & \Phi(\vec{\tau}, x) & \text{flexible constraint} \\
 & \mid & e \mid \emptyset \mid \psi \wedge \psi & \text{other cases} \\
e & ::= & \tau_1 = \ldots = \tau_n & \text{multi-equation}
\end{array}
$$

Translation into constraints

We can translate a type containing parametric flexible types into the combination of type containing unification variables and a conjunction of flexible constraints.

$$
\begin{array}{rcll}
T(\alpha) & = & (\alpha, \emptyset) & \\
T(t(\vec{\tau})) & = & (t(\vec{\tau}'), \bigwedge \vec{\psi}) & \text{where } (\vec{\tau}', \vec{\psi}) = unzip(map\ \ T\ \vec{\tau}) \\
T(\varphi) & = & (\varphi, \emptyset) & \\
T(\Phi(\vec{\tau})) & = & (x, \Phi(\vec{\tau}', x) \wedge \bigwedge \vec{\psi}) & \text{where } (\vec{\tau}', \vec{\psi}) = unzip(map\ \ T\ \vec{\tau}) \\
 & & & \text{and } x \text{ is fresh}
\end{array}
$$

We can also encode instantiation constraints.

$$\forall \vec{\alpha}_1 . \tau_1 \leq \forall \vec{\alpha}_2 . \tau_2 \longrightarrow [\vec{x}/\vec{\alpha}_1]\tau_1 = \tau_2$$

Constraint solving rules

merge

$$\frac{v = e \wedge v = e' \wedge \psi}{v = e = e' \wedge \psi}$$

decompose

$$\frac{t(\tau_1, \ldots, \tau_n) = t(\tau'_1, \ldots, \tau'_n) = e \wedge \psi}{\tau_1 = \tau'_1 \wedge \ldots \wedge \tau_n = \tau'_n \wedge t(\tau_1, \ldots, \tau_n) = e \wedge \psi}$$

promote

$$\frac{\varphi = t(\tau_1, \ldots, \tau_n) = e \wedge \psi}{\varphi = t(\varphi_1, \ldots, \varphi_n) = e \wedge \varphi_1 = \tau_1 \wedge \ldots \wedge \varphi_n = \tau_n \wedge \psi} \ \mathit{fu}(t(\tau_1, \ldots, \tau_n)) \neq \emptyset$$

define

$$\frac{\Phi(\vec{\alpha}, x) \wedge x = \tau = e \wedge \psi}{\Phi(\vec{\alpha}, x) \wedge x = \tau = e \wedge [\Phi(\vec{\alpha}, z) \mapsto z = \tau]\psi} \ \begin{array}{l}\mathit{fu}(\tau) = \emptyset \\ \mathit{fp}(\tau) \subset \vec{\alpha}\end{array}$$

subst

$$\frac{x = \tau = e \wedge \psi}{x = \tau = [\tau/x]e \wedge [\tau/x]\psi} \ \mathit{fu}(\tau) = \emptyset$$

cycle

$$\frac{\psi}{\bot} \ \psi \vdash v > v$$

clash

$$\frac{\alpha = t(\vec{\tau}) = e \wedge \psi}{\bot}$$

clash

$$\frac{\alpha = \alpha' = e \wedge \psi}{\bot} \ \alpha \neq \alpha'$$

clash

$$\frac{t(\vec{\tau}) = u(\vec{\tau}') = e \wedge \psi}{\bot} \ t \neq u$$

scope

$$\frac{\varphi = \tau = e \wedge \psi}{\bot} \ \mathit{fp}(\tau) \neq \emptyset$$

scope

$$\frac{\Phi(\vec{\alpha}, x) \wedge x = \tau = e \wedge \psi}{\bot} \ \mathit{fp}(\tau) \not\subset \vec{\alpha}$$

Model

We model flexible types, unification variables, and flexible constraints using the following 3 valuation functions.

$$
\begin{aligned}
&v_f : S(\varphi) \to \{\tau \in \mathcal{T} \mid \mathit{fp}(\tau) = \emptyset\} \\
&v_u : S(x) \to \{s \subset \mathcal{T} \mid s \neq \emptyset\} \\
&v_c : S(\Phi) \to \{R \in \mathcal{T}^n \times \mathcal{P}(\mathcal{T}) \to \mathit{bool} \mid \forall \vec{\alpha}\tau_1\sigma, R(\vec{\alpha},\sigma) \Rightarrow \tau_1 \in \sigma \Rightarrow \\
&\qquad\qquad \mathit{fp}(\tau_1) \subset \vec{\alpha} \wedge (\sigma = \{\tau_1\} \Rightarrow \forall \sigma'\vec{\tau}, R(\vec{\tau},\sigma') \Rightarrow \sigma' = \{[\vec{\tau}/\vec{\alpha}]\tau_1\})\}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathcal{T} = \{\tau \mid \mathit{fv}(\tau) = \emptyset\} \\
&\mathit{fp}(\tau) = \text{free parametric variables of } \tau \\
&\mathit{fu}(\tau) = \text{free unification variables of } \tau \\
&\mathit{fv}(\tau) = \text{free flexible types and unification variables of } \tau
\end{aligned}
$$

Interpretation

$$
\begin{array}{rcl}
[\![\alpha]\!] & = & \{\alpha\} \\
[\![t(\tau_1,\ldots,\tau_n)]\!] & = & \{t(\tau'_1,\ldots,\tau'_n) \mid \tau'_1 \in [\![\tau_1]\!],\ldots,\tau'_n \in [\![\tau_n]\!]\} \\
[\![\varphi]\!] & = & \{v_f(\varphi)\} \\
[\![x]\!] & = & v_u(x) \\
[\![\tau_1 = \ldots = \tau_n]\!] & = & ([\![\tau_1]\!] = [\![\tau_2]\!]) \wedge \ldots \wedge ([\![\tau_1]\!] = [\![\tau_n]\!]) \\
[\![\Phi(\tau_1,\ldots,\tau_n,x)]\!] & = & \bigwedge\{v_c(\Phi)(\tau'_1,\ldots,\tau'_n,[\![x]\!]) \mid \tau'_1 \in [\![\tau_1]\!],\ldots,\tau'_n \in [\![\tau_n]\!]\}
\end{array}
$$

Properties

Theorem 1 *Each constraint rewriting rule preserves the set of solutions.*

Theorem 2 *If ψ is a normal form constraint (i.e. applying* define *and* subst *do not change it, and none of the other rules apply), then it admits a valuation (v_f, v_u, v_c) such that $[\![\psi]\!] = true$, and $v_u(x)$ is a singleton only if we have an equation $x = \tau = e$ in ψ, and fu$(\tau) = \emptyset$.*

Corollary 1 *If ψ is in normal form, and contains $\tau = e$, then fu$(\tau) = \emptyset$ iff for any valuation satisfying ψ, $[\![\tau]\!]$ is a singleton. Moreover, the translation of $[\![\tau]\!]$ is the same $\{\tau'\}$ for any valuation satisfying ψ iff for all flexible variable φ in fv(τ), and each minimal variable such that $\psi \vdash \varphi > v$ (or $v = \varphi$ itself if it is minimal), there is an equation $v = \tau' = e$ with fv$(\tau') = \emptyset$ in ψ.*

Theorem 3 *There is a rewriting strategy that reaches a normal form in a finite number of steps.*

Global minimality

- Minimality is a weaker version of principality, where we assume that all polymorhic values are given the most general type
- Dependent function calls can be made minimal by usual techniques (cf. polymorphic methods)
- Ambiguity criterion: either there are two concrete solutions, or there exists a partial blocked solution:
  A partial tree matching the constraints, such that all leaves are blocked.
- Combined, this means that, assuming minimality of the environment, satisfying the ambiguity criterion is monotonous (less polymorphism means less ambiguity), and the solution found is unique.

Design decisions

- Local ambiguity or global ambiguity
  Global ambiguity requires backtracking, can be costly
- Ignore the type of value fields or not
  Selecting only on the base of type fields, and names of value fields, provides a simpler mental model
- Termination criterion
  Should use the known part of types, but there is some freedom
- Where to lauch constraint resolution
  Currently done when generalizing flexible types, allowing to handle simultaneously multiple implicit applications
- Hiding/overriding mechanism
  Sometimes we need to hide an implicit module/functor to avoid ambiguity
- Introduce some priorities, to avoid ambiguity?

The diamond problem

Signature inheritance leads to ambiguity.

```
module type Eq = sig type t val equal : t -> t -> bool end

implicit module Eq_int =
  struct type t = int val equal (x : int) y = (x=y) end

module type Cmp = sig include Eq val compare : t -> t -> int end

implicit module Cmp_int =
  struct include Eq_int val compare x y = (x-y) end
```

Both `Eq_int` and `Cmp_int` are subtypes of `Eq with type t = int`.

Practical solution: blessing signatures

Add an abstract type component to disambiguate signatures.

```
module type Eq = sig
  type eq
  type t
  val equal : t -> t -> bool
end

module type Cmp = sig
  type cmp
  include Eq with type eq := cmp                    (* just erases eq *)
  val compare : t -> t -> int
end
```

This blessing seems more meaningful than the types of value fields, and allows to simplify resolution.

Conclusion

- A very expressive framework
- Proof search can be made principal
- However, ambiguity is monotonous in a contravariant way, so that the type system cannot be principal
- The resulting system is well-behaved: minimal and symmetric
- Numerous design decisions