Typechecking of Overloading

in Mechanized Mathematics and Programming Languages

Arthur Charguéraud
with Martin Bodin, Louis Riboulet, et Jana Dunfield

Inria

March 12th, 2025

Mathematics : overloaded symbols

$$x + y \qquad \text{vs} \qquad x +_{\mathbb{Z}} y$$

$$\sum_{d \in \{i, 2i\}} \sum_{k \in -6..7} 3 \cdot e^{\frac{d \cdot \pi}{8}} \cdot M^{2 \cdot k^2} \cdot N$$

Three centuries of mathematics...
but where are the typing rules for overloading resolution???

Needed to mechanize mathematics the way they are casually written.

Programming : overloaded symbols

1. Overloading in mathematical formulae occurring inside programs
2. Overloading for function names, record fields and data constructors.

Challenging to resolve overloading in the presence of local type inference.

```
(* Without overloading *)
Array.iteri f (Array.map succ (Array.concat t (Array.of_list [2;3])))

(* With overloading *)
iteri f (map succ (concat t (to_array [2;3])))
```

```
type point2D = { x : int; y : int }
type point3D = { x : int; y : int; z : int }

let xvalues2D (ps : point2D list) = List.map (fun r -> r.x) ps
(* Error: The expression [ps] has type point2D list
   but an expression was expected of type point3D list *)
```

Prior work on overloading

- Javascript, Python, etc: dynamic resolution
- Java: static resolution with dynamic dispatch
- Haskell: typeclasses, also with runtime overheads
  $\rightarrow$ we are interested in static resolution with static dispatch
- C++: static resolution, but guided by arguments only
  $\rightarrow$ never guided by context: no overloading for constants

```
__instance empty : 'a set
__instance empty : ('a,'b) map
val f : int set -> int
let r = f empty
```

- PVS, ADA: static resolution, guided by arguments and context
  $\rightarrow$ but no type inference: all variables must be annotated
- Mechanized mathematics
  - Coq's notation scope $\rightarrow$ guided by the context only
  - Typeclasses $\rightarrow$ indirection: `Z.add x y` $\neq$ `plus Z_plus_inst x y`
  - Coq's canonical structures $\rightarrow$ complicated, scalability issues

Problem summary

We need a type inference algorithm for resolving overloading, both for programming language and mathematics

- guided by function arguments *and* by expected type
- with support for checking ML type schemes
- with local type inference of STLC types.

Desirable features:

1. Predictable
2. Efficient
3. Nice error reporting

Challenge of resolution

Static resolution of overloading is intertwined with typechecking:

- overloading resolution depends on types
- types of overloaded symbols depend on resolution.

Requires bidirectional propagation of information.

Assume literals can be `int` or `float`, and + can be on `int` or `float`.

```
let example_arith =
  let x = 0 in
  let y = 1 + x in
  let z = 2 in
  let t = (3 + y) + (4 + z) in
  (5 + z) + (6:float)
```

Challenge of partial applications

Partial applications add ambiguities, thus require more annotations, hence decrease the benefits of overloading.

```
__instance sum : int -> int -> int (* [sum x y] *)
__instance sum : int -> int -> int -> int (* [sum x y z] *)
let r = sum 3 4 (* first instance, or partial application of the second? *)
```

Proposal: a dedicated syntax for partial applications.

```
#(sum 3 4 _)        fun z -> sum 3 4 z
#(sum _ 4 5)        fun x -> sum x 4 5
#(sum _ 4 _)        fun x z -> sum x 4 z
```

Thereafter, we consider n-ary functions.

Contents of the talk

1. Complexity of the problem
2. Constraint-based resolution
3. Conjectures
4. Derived instances

Complexity

NP-hardness: encoding to 3-SAT

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_4 \vee \neg x_5) \wedge (x_2 \vee \neg x_3 \vee x_5) \wedge (\neg x_2 \vee x_3 \vee \neg x_5)$$

```
__instance 0 : int (* true and false *)
__instance 0 : float
__instance neg : float -> int (* negation *)
__instance neg : int -> float
__instance f : int -> float -> float -> unit (* at least one true arg *)
__instance f : float -> int -> float -> unit
__instance f : float -> float -> int -> unit
__instance f : int -> int -> float -> unit
__instance f : int -> float -> int -> unit
__instance f : float -> int -> int -> unit
__instance f : int -> int -> int -> unit
let x1 = 0 in (* int or float *)
let x2 = 0 in
let x3 = 0 in
let x4 = 0 in
let x5 = 0 in
f x1 x3 (neg x4); (* at least one argument must be int *)
f x1 x4 (neg x5);
f x2 (neg x3) x5;
f (neg x2) x3 (neg x6);
```

Constraint-based resolution

Overview

Typechecking algorithm:

1. gather contraints from ML typechecking—standard unifications
2. iteratively try to resolve symbols, in any order.

How to resolve a symbol?
Assume x is an overloaded symbol with candidate instances:

$$(v_1 : T_1),\ ...,\ (v_n : T_n).$$

An occurrence of x with a type T constrained by the context resolves to v_i if T unifies with T_i but not with T_j for $j \neq i$.

Representation of terms and types

In this talk, simplified from ML to STLC.

Initialization: annotate each AST node with a fresh, unconstrained type.

$$
\begin{array}{llcl}
\text{Annotated term} & t & ::= & u^{:T} \\
\text{Contents of a term} & u & ::= & x_{\mathsf{id}} \mid v \mid t_1(t_2) \mid \mathsf{let}\, x^{:T} = t_1 \,\mathsf{in}\, t_2 \mid \lambda x^{:T}.\, t_1 \\
\\
\text{Typing environment} & E & ::= & \emptyset \;\mid\; E,\, x : \mathsf{Regular}(T) \;\mid\; E,\, x : \mathsf{Overloaded}(I) \\
\text{Set of instances} & I & ::= & \emptyset \mid I, (v : T) \\
\\
\text{Type representation} & T & ::= & \textit{unique identifiers} \\
\text{Type description} & D & ::= & \mathsf{Flexible} \mid \mathsf{Unified}\, T \mid \mathsf{Constr}(C, \vec{T}) \\
\text{Mutable state} & s & ::= & \emptyset \mid s[T := D] \mid s[\mathsf{id} := v] \mid s[\mathsf{id} := (T, I)]
\end{array}
$$

The operation unify(T_1, T_2) refines the state.

Implementation of unification: standard

```
type id = unit ref

type typ = desc ref
and desc =
  | Flexible
  | Unified of typ
  | Constr of id * typ list

let rec unify (t1:typ) (t2:typ) : unit =
  if !t1 != !t2 then
  match !t1, !t2 with
  | Unified t1', _ -> unify t1' t2
  | _, Unified t2' -> unify t1 t2'
  | Flexible, _ -> t1 := Unified t2
  | _, Flexible -> t2 := Unified t1
  | Constr(c1,ts1), Constr(c2,ts2) ->
      if c1 != c2 || List.length ts1 <> List.length ts2 then raise Failure;
      List.iter2 unify ts1 ts2
```

Term constraints: standard except overloaded symbols

Subterm labelled with its type	Operations to apply
$(\mathsf{let}\, x^{:T_0} = {u_1}^{:T_1}\ \mathsf{in}\ {u_2}^{:T_2})^{:T}$	$\mathsf{unify}(T_0, T_1)\ ;\ \mathsf{unify}(T_2, T)$
$({u_0}^{:T_0}({u_1}^{:T_1}))^{:T}$	$\mathsf{unify}(T_0,\, T_1 \to T)$
$(\lambda x^{:T_0}.\, {u_1}^{:T_1})^{:T}$	$\mathsf{unify}(T,\, T_0 \to T_1)$
$v^{:T}$ where v is a literal of type T'	$\mathsf{unify}(T', T)$
$x_{\mathsf{id}}^{:T}$ if x is bound to $\mathsf{Regular}(T')$	$\mathsf{unify}(T', T)$
$x_{\mathsf{id}}^{:T}$ if x is bound to $\mathsf{Overloaded}(I)$	$s' := s[\mathsf{id} := (T, I)]$

Symbol resolution

Consider an occurrence $x_{\mathsf{id}}^{:T}$ of a not-yet-resolved overloaded symbol.

$$s[\mathsf{id}] = (T, I) \quad \text{where} \quad I = (v_1 : T_1), \ldots, (v_n : T_n)$$

If

$$\begin{aligned} & \mathsf{unify}(T_i, T) \text{ would succed} \\ \wedge \quad & \forall j \neq i.\ \mathsf{unify}(T_j, T) \text{ would fail} \end{aligned}$$

Then

$$\begin{aligned} & s' := s[\mathsf{id} := v_i] \\ & \mathsf{unify}(T_i, T) \end{aligned}$$

Implementation

Resolving a overloaded symbol enables the resolution of other symbols. How to avoid a quadratic processing?

We are currently investigating two possible routes.

1. Use advanced data structures to efficiently find the set of symbols impacted by the resolution of one symbol.
2. Restrict the set of programs that can be handled by processing the symbols in a very specific order (top-down, bottom-up, top-down).

Conjectures

Successful typechecking

Typechecking: ML-typechecking followed by iterated symbol resolution.

Extracted program: the program obtained by replacing overloaded symbols with the values they resolved to.

Theorem (Type soundness)

If a program successfully typechecks, then the extracted program is well-typed in ML.

Theorem (Non-ambiguity)

If a program successfully typechecks, then no other instantiation of the overloaded symbols extracts to a well-typed ML program.

Unsuccessful typechecking

If a program does not typecheck, then:

- either no instantiation of overloaded symbols makes the extracted program well-typed in ML,
- or several distinct instantiations make the extracted program well-typed in ML,
- or there is exactly one possible instantiation, yet it cannot be deduced by a sequence of simple deduction (resolution) steps.

Derived instances

Derived instance, example of sum over arrays

Register an instance of `sum` on `'a array`, assuming + and `zero` on `'a`.

```
let sum (type a) ((+) : a -> a -> a) (zero : a) : a array -> a =
  __instance (fun s -> Array.fold (fun acc v -> acc + v) zero s)

let r1 = sum ([| 4; 5; 6 |] : int array) (* infers [r1 : int] *)
let r2 = sum ([| 4; 5; 6 |] : float array) (* infers [r2 : float] *)
```

Same with packaging of plus and zero.

```
(* Structure to represent additive monoids *)
type 'a monoid = { op : 'a -> 'a -> 'a ; neutral : 'a }

(* Register an instance of [addmonoid] for types with a [(+)] and [zero]. *)
let addmonoid (type a) ((+) : a -> a -> a) (zero : a) : a monoid =
  __instance ({ op = (+); neutral = zero })

(* Register [sum] on ['a array] assuming a monoid on ['a] *)
let sum (type a) (addmonoid as m : a monoid) : a array -> a =
  __instance (fun s -> Array.fold (fun acc v -> m.op acc v) m.neutral s)

let r1 = sum ([| 4; 5; 6 |] : int array)
(* --> relies on a resolution of [addmonoid : int monoid] *)
```

Generalization to sum over containers

```
(* Example instances of the fold operator *)
  __instance fold : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a
  __instance fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

( * Register an instance of [mapreduce f m s] assuming [fold] *)
let mapreduce (type t) (type a) (type x)
  (fold : (a -> x -> a) -> a -> t -> a)
  : (x -> a) -> a monoid -> t -> a =
  __instance (fun f m s -> fold (fun acc x -> m.op acc (f x)) m.neutral s)

(* Register an instance of [sum s] assuming [mapreduce] and [addmonoid] *)
let sum (type t) (type a)
  (mapreduce : (a -> a) -> a monoid -> t -> a)
  (addmonoid as m : a monoid)
  : t -> a =
  __instance (fun s -> mapreduce (fun x -> x) m s)

(* Example usage *)
let r1 = sum ([| 4; 5; 6 |] : int array)
```

Application to mathematical formulae

$$\sum_{x \in s} f(x)$$

```
let bigsum (type t) (type a) (type x) (* instance for [bigsum s f] *)
    (addmonoid as m : a monoid)
    (mapreduce : (a -> a) -> a monoid -> t -> a)
    : t -> (x -> a) -> a =
    __instance (fun s f -> mapreduce f m s)
```

$$\sum_{d \in \{i, 2i\}} \sum_{k \in -6..7} 3 \cdot e^{\frac{d \cdot \pi}{8}} \cdot M^{2 \cdot k^2} \cdot N$$

```
let demo (m:complex matrix) (n:complex matrix) =
  bigsum [i; 2*i] (fun d ->
    bigsum (int_range (-6) 7) (fun k ->
      3 * (e ^ (d * pi / 8)) * (m ^ (2*k^2)) * n))
```