Typechecking of Overloading

in Programming Languages and Mechanized Mathematics

Arthur Charguéraud

Inria

October 17th, 2024

Context

Team CAMUS in Strasbourg: compilation, optimizations, verification.

OptiTrust: user-guided, source-to-source transformations, to produce high-performance code with formal guarantees.

Overloading:

1. for more concise program specifications
2. for programming languages
3. for mechanized mathematics

Overloading in Mathematics

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

Motivation: improve conciseness and readability.

- In on-paper mathematics $\rightarrow$ no explicit algorithm!
- In mechanized mathematics:
  - Notation scope $\rightarrow$ guided by the context only
  - Typeclasses $\rightarrow$ adds a logical indirection
  - Canonical structures $\rightarrow$ complicated, scalability issues
- In mathematical formulae that appear in program specifications

Overloading in Programming Languages

- Java, Javascript, Python, etc: dynamic resolution
- Haskell: typeclasses, also with runtime overheads
  $\rightarrow$ we are interested in static resolution

- OCaml: no overloading; recently for constructors and fields
  $\rightarrow$ fragile resolution, dependent on the order of unifications
- C++: resolution guided by arguments only
  $\rightarrow$ never guided by context; thus no overloading for constants
- PVS, ADA: resolution guided by arguments and context
  $\rightarrow$ but no polymorphism, no local inference

Contribution

This work presents the first typechecking algorithm:

- guided by function arguments *and* by expected type
- with support for polymorphic types.

Moreover, it supports type inference like traditional ML typecheckers (no inference of polymorphism yet).

Challenges

Static resolution of overloading is intertwined with typechecking:

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

Motivating example

Assume literals can be `int` or `float`.
Assume + can be on `int` or `float`.
Try to typecheck:

```
let example =
  let x = (0:int) in
  let y = 1 + 2 in
  (x + y)
```

```
let harder_example =
  let x = 0 in
  let y = 1 in
  let z = (2 + x) + (3 + y) in
  (4 + x) + (5:int)
```

Two-pass algorithm

Our proposal: two passes over the AST, using recursive functions.

First pass:

- propagate expected type downwards
- retreive information from subterms

Second pass:

- propagate expected type downwards (possibly a more refined type)

An overloaded function is attempted to be resolved up to 3 times:

- on the way down in the first pass
- on the way back up in the first pass
- on the way down in the second pass

We accept the idea of rejecting programs that need more propagation.

Local inference

Can infer the type of a local variable based either on its definition or its occurrences.

Typechecking of "$\text{let}\, x = t_1 \,\text{in}\, t_2$" as follows:

1. first pass in t_1
2. first pass in t_2
3. second pass in t_2
4. second pass in t_1

Example:

```
                          1st pass          2nd pass
let example =
  let x = (0:int) in      [x:int]           [x:int]
  let y = 1 + 2 in        [y:Unresolved]    [1:int], [2:int]
  (x + y)                 [x+y:int]         [y:int]
```

Local inference for functions

```
let exlet1 (f:int->int) (g:int->int) (x:int) : int =
  f (2*x + 42) + g (3*x + 42)

let exlet2 (f:int->int) (g:int->int) (x:int) : int =
  let op = (fun n -> n + 42) in
  f (op (2*x)) + g (op (3*x))
```

Resolution of overloaded constants

Guided only by expected type.
If cannot resolve, assign type `Unresolved`.
After second pass, all types should be resolved.

At each pass, count instances that would unify with expected type.

- If zero, then typing error.
- If several, then `Unresolved`.
- If one, then resolution succeds; unify instance with expected type.

Resolution of overloaded functions

Consider $t_0(t_1, .., t_n)$. Let T_r be the expected type.

First pass:

1. Typecheck t_0 with expected type $T_1 \to .. \to T_n \to T_r$ for fresh T_i
2. Typecheck each t_i with expected type T_i
3. Try resolve t_0 if its type is `Unresolved`
4. Save T_r as type for the call

Second pass:

1. Let T be the type saved for the call and T_i for the arguments
2. Unify T with T_r
3. Typecheck t_0 with expected type $T_1 \to .. \to T_n \to T_r$
4. Typecheck each t_i with expected type T_i

Partial applications

Partial applications add ambiguities,
hence decrease the interest of overloading.

Proposal: use a dedicated syntax instead.

```
#(f 3 _)          fun y -> f 3 y
#(f _ 4)          fun x -> f x 4
```

Opaque vs Transparent Types

If `t` unifies with `u`, then instances `u -> int` and `t -> int` overlap.

If `t` is an abstract type, it can be used to discriminate.

Overloaded record fields

```
type t = { mutable f : int; mutable g : int }
```

Encodings:

```
r.f                          __get_f r
r.f <- 3                     __set_f r 3
{ f = 3; g = 4 }             __make_f_g 3 4
{ r with f = 3 }             __with_f r 3
{ r with f = 3; g = 4 }      __with_g (__with_f r 3) 4
```

Examples with overloaded records

```
type t = { f : int; mutable g : int }
type u = { f : int; mutable g : float }
type v = { f : int; mutable g : float; h : bool }

let r1 (r:t) = r.f (* resolves [f] to be a field of [t] *)

let r2 : t = { f = 3; g = 2 } (* [2] resolves as [int] *)

let r3 = { f = 3; g = (2:float) } (* resolves [r3] to [u] *)

let r4 = { f = 3; g = 2; h = true } (* resolves [r4] to [v] *)

let r5 = r2.g <- 2 (* [r2] has type [t], thus [2] resolves to [int] *)

let r6 = { f = 2; g = 3 } (* rejected: ambiguous *)
```

Overloaded data constructors

```
type t = Var of string | Let of string * t * t | Load of t
type u = Var of string | Let of string * u * u | Load of string
```

```
let rec norm (e:t) : u =
  match e with
  | Var x -> Var x
  | Let (x, t1, t2) -> Let (x, norm t1, norm t2)
  | Load t1 ->
      match t1 with
      | Var x -> Load x
      | _ -> let x = generate_var_fresh_from t1 in
             Let (x, norm t1, Load x)
```

Typechecking of pattern matching

Desirable equivalence:

```
match t0 with x -> t1        let x = t0 in t1
```

Typechecking with type `T` of:

```
match t0 with
| p1 -> t1
| p2 -> t2
```

1. Typecheck `t0`, obtain a type `T_0`.
2. Typecheck `p1` and `p2`, with expected type `T_0`.
3. Typecheck `t1` and `t2`, with expected type `T`.
4. Typecheck again `t1` and `t2`, with expected type `T`.
5. Typecheck again `p1` and `p2`, with expected type `T_0`.
6. Typecheck again `t0`, with expected type `T_0`.

Advanced matching

```
type t = A of t | B of int | C of int
type u = A of u | B of float

let f v =
  match v with
  | A _ -> ()
  | B _ -> ()
  | C _ -> () (* resolves [v:t] on 1st pass *)

let g v = (* resolves [v:t] on 2nd pass *)
  match v with
  | A (B x) -> ()
  | A (B x) -> ignore (x:int)
  | _ -> ()

type 'a p = P of 'a * 'a

let h v =
  match v with
  | P (A y, B x) -> (x:int) (* would need 3 passes to resolve [A] *)
```

Higher-order iterators

```
val List.map : 'a list -> ('a -> 'b) -> 'b list
val Array.map : 'a array -> ('a -> 'b) -> 'b array

let map = __instance Array.map
let map = __instance List.map

let d : float list = [3.2; 4.5]
let ex12 = map (fun x -> 2 * x + 1) d
```

Fails to typecheck unless swapping arguments of `map` or adding a feature.

What is the mathematicians' intuition?

$$\sum_{x \in E} (2x + 1)$$

Input and output arguments

Additional feature: possibility specify the *input* and *output* arguments.

```
let map = __overload [Out; In]
```

Unless specified otherwise, all arguments are *input*.

Output-mode arguments are typechecked only after the overloaded function is resolved.

If resolution happens during the second pass, output arguments are typechecked both in first pass and second pass.

Derived instance

```
val matrix_add : ('a -> 'a -> 'a) -> 'a matrix -> 'a matrix -> 'a matrix


(* Register an instance for [+] on the type ['a matrix], for every type
   ['a] for which there exists an instance of [+] on the type ['a]. *)

let (+) (type a) ((+) : a -> a -> a) : a matrix -> a matrix -> a matrix =
  __instance (fun m1 m2 -> matrix_add (+) m1 m2)

(* Register an instance of [sum] for arrays with [+] and [zero]. *)

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

Packing arguments

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

(* Register an instance of the additive monoid on [int] *)
let addmonoid : int monoid = __instance { op = (+); neutral = 0 }

(* Register an instance of [sum] for arrays whose elements are equipped
  with the additive monoid. *)
let sum (type a) (m : a monoid) : a array -> a =
  __instance (fun s -> Array.fold (fun acc v -> m.op acc v) m.neutral s)

(* Example usage *)
let result1 = sum ([| 4; 5; 6 |] : int array)
```

Sum operator over containers

```
(* 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 })

(* Example instances of fold operators *)
 let fold : ('a -> 'x -> 'a) -> 'a -> 'x array -> 'a = Array.fold_left
 let fold : ('a -> 'x -> 'a) -> 'a -> 'x list -> 'a = List.fold_left

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

(* Example usage *)
let result2 = sum ([| 4; 5; 6 |] : int array)
```

Example mathematical formula

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

```
let bigsum (type t) (type a) (type x)
    (addmonoid : a monoid)
    (mapreduce : (a -> a) -> a monoid -> t -> a)
    : t -> (x -> a) -> a =
    __instance (fun s f -> mapreduce f addmonoid s)

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

Design choices with packing

1. **Without packing:**
   - define `plus` and `zero` on `int`
   - derive `sum` from `plus` and `zero`

     → too many arguments when operating on fields

2. **With upfront packing:**
   - define `ring` on `int`
   - derive `addmonoid` from `ring`
   - derive `plus` and `zero` from `addmonoid`
   - derive `sum` in terms of `addmonoid`

3. **With last-minute packing:**
   - define `plus` and `zero` on `int`
   - derive `addmonoid` from `plus` and `zero`
   - derive `sum` in terms of `addmonoid`

Challenges with packing

Algebraic hierarchy: derive instances for operations, and for properties

Overlapping instances: reject? accept if convertible solutions?

Efficiency: caching of resolved instances?

Future work: extensions

1. Printing expressions with overloaded symbols wherever possible.
2. On-the-fly introduction of instances during quantification.
   $\rightarrow$ e.g. assume a commutative group $G(0, +)$
3. Interaction with coercions.
   $\rightarrow$ additional sources of ambiguities
4. Interaction with dependent types (?)

Future work: applications

1. Apply at scale in ML programming, with OCaml extraction.
2. Apply at scale in mechanized mathematics, with Coq parser.
3. Experiment with overloading of lemma names.
   $\rightarrow$ for example `rewrite plus_comm`.

Conclusion

A simple, efficient, practical, bidirectional typechecking algorithm for ML code with overloaded symbols. Maybe soon for a (subset of) Coq?

Paper: JFLA submission on my webpage.

Implementation: prototype available will be made public soon.

Thanks!