

Typechecking of Overloading

in Programming Languages and Mechanized Mathematics

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

Inria

January 28th, 2025

OCaml motivation : overloaded operators

```
(* Without overloading *)  
(x +. 1.5) *. (3.0 *. pi /. 4.0)
```

```
(* With overloading *)  
(x + 1.5) * (3 * pi / 4)
```

OCaml motivation : avoiding module names

```
(* 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])))
```

OCaml motivation : overloaded record fields

```
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 *)
```

Mathematics : overloaded symbols

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

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

Mathematics : overloaded symbols

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

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

Three centuries of mathematics...

but where are the typing rules for overloading resolution???

Work on overloading

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

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

Work on overloading

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

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

- PVS, ADA: resolution guided by arguments and context
→ but no polymorphism, no local inference

Work on overloading

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

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

- PVS, ADA: resolution guided by arguments and context
→ but no polymorphism, no local inference
- Mechanized mathematics
 - ▶ Coq's notation scope → guided by the context only
 - ▶ Typeclasses → adds a logical indirection: `plus Z_plus_inst x y`
 - ▶ Coq's canonical structures → 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.

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 propagation

Static resolution of overloading is intertwined with typechecking:

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

Challenge of propagation

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.

```
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)  
  
let example_higherorder =  
  let a : int = 3 in  
  map (fun x -> 1 + 2 * x) [a; 4]  
  
let example_letfun (f:int->int) (g:int->int) (x:int) : int =  
  let op = (fun n -> n + 42) in  
  f (op (2 * x)) + g (op (3 * x))
```

where literals can be int or float, and + can be on int or float.

Challenge of partial applications

```
__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? *)
```

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

Challenge of partial applications

```
__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? *)
```

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

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. A constraint-based algorithm
3. Theorems
4. A faster, 2-pass algorithm
5. Extensions
6. Demo

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 *)
__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 algorithm

Simple deductions only

Principle of the declarative algorithm:

- gather constraints from ML typechecking
- iteratively try to resolve symbols, in any order.

Assume x is an overloaded symbol with candidate instances:

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

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.

Annotated term $t ::= u{:T}$

Contents of a term $u ::= x_{\text{id}} \mid v \mid t_1(t_2) \mid \text{let } x{:T} = t_1 \text{ in } t_2 \mid \lambda x{:T}. t_1$

Representation of terms and types

In this talk, simplified from ML to STLC.

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

Annotated term $t ::= u{:T}$

Contents of a term $u ::= x_{\text{id}} \mid v \mid t_1(t_2) \mid \text{let } x{:T} = t_1 \text{ in } t_2 \mid \lambda x{:T}. t_1$

Typing environment $E ::= \emptyset \mid E, x : \text{Regular}(T) \mid E, x : \text{Overloaded}(I)$

Set of instances $I ::= \emptyset \mid I, (v : T)$

Representation of terms and types

In this talk, simplified from ML to STLC.

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

Annotated term $t ::= u{:}^T$

Contents of a term $u ::= x_{\text{id}} \mid v \mid t_1(t_2) \mid \text{let } x{:}^T = t_1 \text{ in } t_2 \mid \lambda x{:}^T. t_1$

Typing environment $E ::= \emptyset \mid E, x : \text{Regular}(T) \mid E, x : \text{Overloaded}(I)$

Set of instances $I ::= \emptyset \mid I, (v : T)$

Mutable state $s ::= \emptyset \mid s[\text{id} := v] \mid s[\text{id} := (T, I)]$

Representation of terms and types

In this talk, simplified from ML to STLC.

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

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

The operation $\text{unify}(T_1, T_2)$ 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

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

Symbol resolution

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

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

If

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

Then

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

Theorems—or rather, 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.

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.

2-pass algorithm: avoid a quadratic blow-up

Two-pass algorithm

First recursive pass:

- propagate expected type downwards
- retrieve type information from subterms

Second recursive pass:

- propagate a possibly more refined expected type downwards

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

Two specificities: treatment of let-bindings and of functions calls

Propagation on let-bindings

Purpose: infer the type of a local variable based either on its definition or its occurrences. Typechecking of “let $x = t_1$ in t_2 ” proceeds 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

Propagation on let-bindings

Purpose: infer the type of a local variable based either on its definition or its occurrences. Typechecking of “let $x = t_1$ in t_2 ” proceeds 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

```
let example_failure = (* requires more than 2 passes *)
  let x = 0 in
  let y = 1 + x in
  let z = 2 in
  let t = (3 + y) + (4 + z) in
  (5 + z) + (6:float)
```

```
let example_success = (* typechecks with our algorithm *)
  let x = 0 in
  let y = 1 + x in
  let z = 2 in
  let t : float = (3 + y) + (4 + z) in
  (5 + z) + 6
```

Propagation on overloaded function calls

Consider $t_0(t_1, \dots, t_n)$ annotated with type T . Let T_i be the type of t_i .

First pass:

1. Recursively process t_0 with expected type $T_1 \rightarrow \dots \rightarrow T_n \rightarrow T$
2. Recursively process each t_i
3. If t_0 is an overloaded symbol still unresolved, attempt its resolution

Second pass:

1. Recursively process t_0
2. Recursively process each t_i

In total, up to 3 attempts at resolving an overloaded function.

Extensions: modes, records, derived instances

Higher-order iterators

```
__instance map : 'a list -> ('a -> 'b) -> 'b list  
__instance map : 'a array -> ('a -> 'b) -> 'b array
```

```
let example_higherorder =  
  let a : int = 3 in  
  map (fun x -> 1 + 2 * x) [a; 4]
```

Two passes do not suffice to typecheck the subterm 1.

Higher-order iterators

```
__instance map : 'a list -> ('a -> 'b) -> 'b list
__instance map : 'a array -> ('a -> 'b) -> 'b array

let example_higherorder =
  let a : int = 3 in
  map (fun x -> 1 + 2 * x) [a; 4]
```

Two passes do not suffice to typecheck the subterm 1.

We could swap the arguments of map.

But typability should not depend on the order of arguments.

What is the mathematicians' intuition?

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

Input and output arguments

For each symbol, optionally specify *input* and *output* arguments.
By default, arguments are *input*.

An *output* argument is not processed before the function is resolved.

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

let example_higherorder = (* typechecks in 2 passes *)
  let a : int = 3 in
  map (fun x -> 1 + 2 * x) [a; 4]
```

For higher-order iterators, always set the functional argument as *output*.

Encoding record operations as function calls

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

Example encodings:

```
r.x                __get_x r  
r.x <- 3           __set_x r 3  
{ x = 3; y = 4 }   __make_x_y 3 4  
{ r with x = 3 }   __with_x r 3  
{ r with x = 3; y = 4 } __with_y (__with_x r 3) 4
```

Instances include:

```
__instance __get_x : point2D -> int -> int  
__instance __set_x : point2D -> int -> int -> unit  
__instance __make_x_y : int -> int -> point2D  
__instance __with_x : point2D -> int -> point2D
```

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] *)
```

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)
```

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))
```

Demo

`https://chargueraud.org/`

in the menu, typing overloading:

draft and proto.

Appendix

Formalization

$$\begin{array}{c}
\frac{E(x) = \text{Regular}(S) \quad s_1 \vdash \text{unifyScheme}(S, T) \dashv s_2}{E; s_1 \vdash_{\text{fst}} x_{\text{id}}^{:T} \dashv s_2} \qquad \frac{E(x) = \text{Overloaded}(I) \quad s_1 \vdash_{\text{res}} (x_{\text{id}}^{:T}) \in I \dashv s_2}{E; s_1 \vdash_{\text{fst}} x_{\text{id}}^{:T} \dashv s_2} \\
\\
\frac{s_1 \vdash \text{checkScheme}(S, T_1) \dashv s_2 \quad s_2 \vdash \text{unify}(T_2, T) \dashv s_3 \quad E; s_3 \vdash_{\text{fst}} (u_1^{:T_1}) \dashv s_4 \quad E, x : \text{Regular}(T_1); s_4 \vdash_{\text{fst}} (u_2^{:T_2}) \dashv s_5}{E; s_1 \vdash_{\text{fst}} (\text{let } x^{:S} = (u_1^{:T_1}) \text{ in } (u_2^{:T_2}))^{:T} \dashv s_5} \\
\\
\frac{s_1 \vdash \text{unify}(T, T_0 \rightarrow T_1) \dashv s_2 \quad E, x : \text{Regular}(T_0); s_2 \vdash_{\text{fst}} (u_1^{:T_1}) \dashv s_3}{E; s_1 \vdash_{\text{fst}} (\lambda x^{:T_0}. u_1^{:T_1})^{:T} \dashv s_3} \\
\\
\frac{s_1 \vdash \text{unify}(T_0, T_1 \rightarrow T) \dashv s_2 \quad E; s_2 \vdash_{\text{fst}} (u_0^{:T_0}) \dashv s_3 \quad E; s_3 \vdash_{\text{fst}} (u_1^{:T_1}) \dashv s_4 \quad \forall x. \forall \text{id}. \forall I'. \neg (u_0 = x_{\text{id}} \wedge s_4[\text{id}] = I')}{E; s_1 \vdash_{\text{fst}} (u_0^{:T_0} (u_1^{:T_1}))^{:T} \dashv s_4} \\
\\
\frac{s_1 \vdash \text{unify}(T_0, T_1 \rightarrow T) \dashv s_2 \quad E; s_2 \vdash_{\text{fst}} (u_0^{:T_0}) \dashv s_3 \quad E; s_3 \vdash_{\text{fst}} (u_1^{:T_1}) \dashv s_4 \quad u_0 = x_{\text{id}} \quad s_4[\text{id}] = I' \quad s_4 \vdash_{\text{res}} (x_{\text{id}}^{:T_0}) \in I' \dashv s_5}{E; s_1 \vdash_{\text{fst}} (u_0^{:T_0} (u_1^{:T_1}))^{:T} \dashv s_5} \\
\\
\frac{\text{id} \notin \text{dom } s}{s \vdash_{\text{snd}} x_{\text{id}}^{:T} \dashv s} \qquad \frac{s[\text{id}] = v}{s \vdash_{\text{snd}} x_{\text{id}}^{:T} \dashv s} \qquad \frac{s_1[\text{id}] = I' \quad s_1 \vdash_{\text{res}} (x_{\text{id}}^{:T}) \in I' \dashv s_2 \quad s_2[\text{id}] = v}{s_1 \vdash_{\text{snd}} x_{\text{id}}^{:T} \dashv s_2} \\
\\
\frac{s_1 \vdash_{\text{snd}} t_2 \dashv s_2 \quad s_2 \vdash_{\text{snd}} t_1 \dashv s_3}{s_1 \vdash_{\text{snd}} (\text{let } x = t_1 \text{ in } t_2)^{:T} \dashv s_3} \qquad \frac{s_1 \vdash_{\text{snd}} t_1 \dashv s_2}{s_1 \vdash_{\text{snd}} (\lambda x^{:T_0}. t_1)^{:T} \dashv s_2} \qquad \frac{s_1 \vdash_{\text{snd}} t_0 \dashv s_2 \quad s_2 \vdash_{\text{snd}} t_1 \dashv s_3}{s_1 \vdash_{\text{snd}} (t_0(t_1))^{:T} \dashv s_3}
\end{array}$$

OCaml motivation : overloaded 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)
```

Opaque vs Transparent Types

If t unifies with u , then instances $u \rightarrow \text{int}$ and $t \rightarrow \text{int}$ overlap.

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

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 *)
```

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 t_0 , obtain a type T_0 .
2. Typecheck p_1 and p_2 , with expected type T_0 .
3. Typecheck t_1 and t_2 , with expected type T .
4. Typecheck again t_1 and t_2 , with expected type T .
5. Typecheck again p_1 and p_2 , with expected type T_0 .
6. Typecheck again t_0 , 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)
  | _ -> ()
```

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] *)
```