Towards a Fully Prolog'ical Typechecker

Jacques Garrigue Takafumi Saikawa

May 28, 2021

> Introduce an abstraction boundary between "the type algebra" and "the type checker" [...] so that the type checker is forced to go through a proper API to access/mutate type nodes. This would make it impossible to "forget" a call to 'repr' and will allow further changes on the internal representation.
>
> — `ocaml/typing/TODO.md`, 2018

1 Introduction

OCaml's type inference constructs a type derivation tree of a given program based on information collected from the abstract syntax tree (AST) of the program. This process can be viewed as a form of logical programming, i.e. the construction of a proof tree for a given proposition, here the property that the program can be typed. Theoretically, the information can be classified into several kinds of "constraint": structural constraints expressing binding and scoping of variables and types in the program, and typing constraints concerning equality, instantiation order, and subtyping order.

In the actual implementation of OCaml, such notions of constraints are not expressed as data structures, but just handled by calls to numerous functions enforcing them. In particular, unification resolves equality constraints by directly modifying types. Because of the lack of abstraction barriers, many such functions freely touch the low-level representation of types. As a result it becomes difficult to still view them in a logical way.

Our goal is to make the logical structure of type inference explicit in the code. This work, without yet changing the handling of the constraint solving, clarifies the fact that types in the typechecker are *logical variables* in the sense of Prolog, by making their interface faithful to their intended semantics. This fact has been buried and hidden in the current heavily effectful implementation, despite the presence of several features which are charactestic of Prolog. According to the `git` log, backtracking was added in 2002 [1], and path-compression in 2015 [2].

A logical variable is not a single concrete object but an equivalence class of objects: the unification algorithm may generate multiple representations of a single type connected by equality relations, which is exactly a logical variable for the typechecker. The handling of such an equivalence class almost always requires a calculation of its canonical representation, especially when comparing two types.

The problem in the implementation, as of OCaml 4.12, is twofold:

1. The data structure representing types is public and mutable at the same time, allowing any part of the code to break invariants.

2. The responsibility to take care of when to perform the canonization is totally left to the programmer.

Because of these issues, a programmer cannot safely regard a type expression as a logical variable, and instead, always has to ponder whether the representation is already canonical or not.

The first part can be solved by making the type of type expressions private [3]. This change makes it easier to pinpoint their mutations. For the second part, we further need to make it an abstract type, introducing an abstract interface that encapsulates all accesses to type expressions [5]. This lets programmers use type expressions as proper logical variables.

2 The problem with type expressions

Unification turns equality constraints into equivalence classes of types. The equivalence classes are grown by a variant of the union-find algorithm, memorizing observed equalities by links between nodes.

The structure of a type expression is described by the type `type_expr`:

```
type type_expr =
  { mutable desc: type_desc;
    mutable level: int;
    mutable scope: int;
    id: int }
and type_desc =
    Tvar of string option
  | Ttuple of type_expr list
  | Tconstr of
      Path.t * type_expr list * abbrev_memo ref
  | Tlink of type_expr
 (*...*)
```

`Tvar`, `Ttuple` and `Tconstr` are concrete nodes that respecively represent type variables, types of tuples and defined types (data types, abbreviations and abstract types). `Tlink` nodes appear only internally, and are used to express sharing introduced by unification. During unification, turning a `Tvar` into a `Tlink` amounts to substitution. Other kinds of nodes are also shared by unification in order to handle recursive types. When the unification of two types succeeds, they enter the same equivalence class.[1]

Because of `Tlink`s, a logical type may have multiple representations, and we need to work on a canonical representation. It is calculated by the function `repr`:

```
val repr: type_expr -> type_expr
```

Programmers need to call `repr` before manipulating a type, typically like this:

```
let rec generalize ty =
  let ty = repr ty in
  if (ty.level > !current_level)
  && (ty.level <> generic_level) then begin
    set_level ty generic_level;
    begin match ty.desc with
      Tconstr (_, _, abbrev) ->
        iter_abbrev generalize !abbrev
    | _ -> ()
    end;
    iter_type_expr generalize ty
  end
```

Every access to fields of `type_expr` assumes that it is already in canonical form. Forgetting `repr`, the programmer might get an unexpected behavior. However, as in the above code, one is often tempted to minimize the number of calls to `repr`. Here the correctness relies on `set_level` not changing the `desc` field of `ty`. In more complex functions, this practice is prone to errors. Moreover, a number of functions expect their argument to be in canonical form, here `iter_type_expr`. One also has to keep in mind such calling conventions.

3 Solution

We solve this problem by having all accesses to `type_expr` use `repr`. This is achieved by making `type_expr` abstract. The original interface is kept as a private type `transient_expr` for some exceptional, low-level uses.

```
type type_expr
type type_desc = (*...*)

type transient_expr = private
  { mutable desc: type_desc;
    mutable level: int;
    mutable scope: int;
    id: int }

val get_desc: type_expr -> type_desc
(*...*)

module Transient_expr : sig
  (* Operations on transient types *)
  val create: type_desc -> level: int -> scope: int ->
              id: int -> transient_expr
  val set_desc: transient_expr -> type_desc -> unit
  val repr: type_expr -> transient_expr
  (*...*)
end
```

Using this interface, `generalize` is modified as follows:

```
let rec generalize ty =
  let level = get_level ty in
  if (level > !current_level)
  && (level <> generic_level) then begin
    set_level ty generic_level;
    begin match get_desc ty with
      Tconstr (_, _, abbrev) ->
        iter_abbrev generalize !abbrev
    | _ -> ()
    end;
    iter_type_expr generalize ty
  end
```

The `level` and `desc` fields are now accessed through accessor functions, which always issue a call to `repr`. `iter_type_expr` is also using `get_desc` internally. This makes all the accesses safe.

One might be concerned about the performance overhead due to the change. The number of calls to `repr` increases by a factor of about 3. The execution time of `ocamlc.opt` increases by about 2 – 4 %. For large projects compiled by `ocamlopt.opt`, the overhead is even smaller: in the case of Coq, we observe no overhead.

4 Conclusion

To summarize, we cleanly encapsulated the low-level representation of type expressions into an abstract type, which can now be used safely. This should benefit future compiler development, in a practical way by avoiding bugs, in a theoretical way by helping understand the compiler's semantics, and in a social way by allowing more potential developers to join the development of the compiler.

This is of course not the end of the road. First `type_expr` is not the only kind of logical variable in the compiler, and we need to enforce the same invariant for several other data structures. Second, as stated in the introduction, our ultimate goal is to make the logical structure of type inference fully explicit. This can be

[1] This explanation is slightly incorrect when one of the types is an abbreviation. In that case, the link to the shared type does not override the node itself, but is added to the `abbrev_memo`, ensuring that subsequent expansions of this type will be shared. As a result, the node itself is not in the equivalence class, but its expansion is.

done at several levels: separating structural constraints from typing constraints during inference [4], and introducing a concrete language for typing constraints, which we view as the next major move.

References

[1] GitHub commit, Jacques Garrigue, Nov., 21, 2002, "add unification backtracking", `https://github.com/ocaml/ocaml/commit/65c80f8ae14fdcec297d43efeb64cb11318ba137`

[2] GitHub commit, Jacques Garrigue, Nov., 16, 2015, "Do path compression, and undo it in case of unification error", `https://github.com/ocaml/ocaml/commit/5d8397a9376a79490520e6a8c5e14ca85f3c165f`

[3] GitHub pull request #9994, Jacques Garrigue and Takafumi Saikawa, "Make type_expr private", `https://github.com/ocaml/ocaml/pull/9994`

[4] GitHub pull request #10311, Jacques Garrigue and Takafumi Saikawa, "Separate constraint-solving (unification) part of type_pat into solve_*", `https://github.com/ocaml/ocaml/pull/10311`

[5] GitHub pull request #10337, Jacques Garrigue and Takafumi Saikawa, "Normalize type_expr nodes on access", `https://github.com/ocaml/ocaml/pull/10337`