Adding GADTs to OCaml
the direct approach

Jacques Garrigue & Jacques Le Normand

Nagoya University / LexiFi (Paris)

https://sites.google.com/site/ocamlgadt/

Generalized Algebraic Datatypes

- Algebraic datatypes allowing different type parameters for different cases
- Similar to inductive types of Coq *et al.*

```
type _ expr =
  | Int : int -> int expr
  | Add : (int -> int -> int) expr
  | App : ('a -> 'b) expr * 'a expr -> 'b expr

App (Add, Int 3) : (int -> int) expr
```

- Able to express invariants and proofs
- Also provide existential types

Previous work (in OCaml)

Work by Pottier and Régis-Gianas on type inference for GADTs.

Stratified type inference for generalized algebraic data types [POPL06].

- Separate type inference for GADTs from the core of the language
- Uses propagation and constraint resolution for GADTs
- Preliminary implementation of propagation, never merged

We choose a more direct approach.

GADT support

– Many examples require polymorphic recursion
  → available since OCaml 3.12, originally for GADTs

– Pattern matching allows refining types
  → use local abstract types

– Combining the two in a new syntax

```
let rec eval : type t. t expr -> t = function
  | Int n -> n                                   (* t = int *)
  | Add -> (+)                     (* t = int -> int -> int *)
  | App (f, x) -> eval f (eval x)   (* polymorphic recursion *)
val eval : 'a expr -> 'a = <fun>

eval (App (App (Add, Int 3), Int 4));;
- : int = 7
```

Why use local abstract types?

OCaml has two kinds of type variables:

– Unification variables, with scope the whole function

```
let f x (y : 'a) = (x : 'a) + 1
val f : int -> int -> int
```

– Explicitly quantified universal variables, with scope limited to the annotation. Their goal is to allow unification of type schemes.

```
let f : 'a. 'a -> _ = fun x -> (1 : 'a)
val f : 'a -> int
```

Neither of them can be used as universal expression-scoped variable, such as available in Standard ML.

Rather than introduce a 3rd kind of type variables, we choose to reuse local abstract types.

Type annotations creating new types

The syntax

$$\texttt{let rec}\, f : \texttt{type}\ t_1\ \ldots\ t_n.\tau = \mathit{body}$$

is actually a short-hand for

$$\begin{array}{l}\texttt{let rec}\, f : \alpha_1\ \ldots\ \alpha_n.[\alpha_1 ... \alpha_n / t_1 ... t_n]\tau = \\ \quad \texttt{fun}\ (\texttt{type}\ t_1)\ \ldots\ (\texttt{type}\ t_n) \rightarrow (\mathit{body} : \tau)\end{array}$$

It defines a recursively polymorphic function, whose type variables are visible as locally abstract types.

Application to polytypic functions

Intuitively, the following is type sound:

```
let rec neg : 'a. 'a -> 'a = function
  | (n : int)  -> -n
  | (b : bool) -> not b
  | (a, b)     -> (neg a, neg b)
  | x          -> x
val neg : 'a -> 'a
```

For languages showing an early commitment to types, two problems:
(but this is ok for FLP languages)

- Requires runtime type information
- Requires a default case to be exhaustive

GADTs can express both

Tagged encoding

Traditional sum types, but the parameter provides information about the contents.

```
type _ data =
  | Int  : int -> int data
  | Bool : bool -> bool data
  | Pair : 'a data * 'b data -> ('a * 'b) data

let rec neg : type a. a data -> a data = function
  | Int n  -> Int (-n)
  | Bool b -> Bool (not b)
  | Pair (a, b) -> Pair (neg a, neg b)
```

Guarantee that the result is of the same kind as the input.

Tagless encoding

Tags do not need to be inside the data itself.

```
type _ ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tpair : 'a ty * 'b ty -> ('a * 'b) ty

let rec print : type a. a ty -> a -> string = fun t d ->
  match t, d with
  | Tint, n -> string_of_int n
  | Tbool, b -> if b then "true" else "false"
  | Tpair (ta, tb), (a, b) ->
      Printf.sprintf "(%s, %s)" (print ta a) (print tb b)
```

Need to allow left-to-right dependencies in pattern-matching.

Other applications

There is already a large literature of algorithms using GADTs.

- Data structures enforcing invariants
  E.g. balanced trees (c.f. Tim Sheard et al.)

- Typed syntax
  E.g. encodings of lambda-terms and evaluators (ibidem)

- Parsing
  Menhir is supposed to be "GADT ready".
  I.e., one can generate efficient type parsers using GADTs

- DSLs for any kind of application: GUI, database...

Type inference

- Obtaining sound type inference for GADTs is not difficult.

- Intuitively, one just needs to use a special kind of unification, able to refine universal type variables, when pattern-matching GADT constructors.

- However, making it complete for some definite specification is more difficult.

Unification for GADTs

– Distinguish normal variables and refinable variables.

– The former are traditional unification variables, the latter can be represented as local abstract types.

– Pattern-matching may instantiate refinable variables.

– Need to proceed left to right, to handle dependencies.

– Outside of pattern-matching, they behave as abstract types.

– Forget this instantiation when moving to the next case.

The difficulties

- Cannot use OCaml type variables
  Luckily, local abstract types were added in 3.12, and adding an equation to an abstract type is easy
- Unification should not share internal nodes
  Sharing might be invalidated when we forget equations
- Cannot handle objects and polymorphic variants
  They both require structural sharing
  We keep compatibility when there are no equations
- Principality/completeness are lost
  Recovered partially by controlling propagation
- Must restrict co-variance and contra-variance
- Exhaustiveness of pattern-matching is harder to check

Variance

– Instantiated parameters are not allowed to have a variance

– If this were allowed, we could have this code

```
type -'a t = C : < m : int > -> < m : int > t

let eval : type a . a t -> a = fun (C x) -> x
val eval : 'a t -> 'a

let a = C (object method m = 5 end)
val a : < m : int > t = <object>

let b = (a :> < m : int ; n : bool > t)
val b : < m : int ; n : bool > t = <object>

let c = eval b
val c : < m : int ; n : bool > = <object>
```

Exhaustiveness

One should be able to omit “impossible” cases.

```
let rec equal : type a. a data -> a data -> bool = fun a b ->
  match a, b with
  | Int m, Int n ->
      m - n = 0
  | Bool b, Bool c ->
      if b then c else not c
  | Pair(a1,a2), Pair(b1,b2) ->
      equal a1 b1 && equal a2 b2
```

Typing guarantees that `a` and `b` are of the same kind, so there is no need to handle other cases.

Done by generating the other cases, and checking whether they may happen. The algorithm is sound but not complete.

Subtleties of exhaustiveness

- Usual type inference checks which types are unifiable
- We need to know which types are incompatible

Unfortunately, some types are neither:

```
type (_,_) eq = Eq : ('a,'a) eq

module M : sig
  type t and u
  val eq : (t,u) eq
end = struct
  type t = int and u = int
  let eq = Eq
end

match M.eq with Eq -> "here t = u !"
```

Incompatibility

Unrelated to unification, we define an incompatibility relation.

Two types are incompatible if:

- they are structurally different (e.g. function vs. tuple)
- for datatype definitions, their representations are incompatible (private types are also in this category)
- for abstract types, both declarations must be either in the initial environment (*i.e.* Pervasives) or in the current module

Without the clause about the initial environment, we wouldn't even be able to distinguish `int` and `bool`!

In patterns, unification does not fail when we cannot prove incompatibility, but equations are only added for refinable variables.

Completeness

As it is well-known, type inference for GADTs is not complete.

In the absence of type annotations, pattern-matching branches have incompatible types.

```
let eval = function
  | Int n -> n
  | Add -> (+)
Error: This pattern matches values of type
       (int -> int -> int) expr but a pattern was expected
       which matches values of type int expr
```

However, we do not want to put type annotations directly on the pattern-matching construct.

Propagation of type information

Our idea is to track the validity of type information through sharing, like we did for first-class polymorphism.

A type node is safe if it is not shared with the environment. Thanks to instantiation, safety is propagated through polymorphic functions.

$$C[\texttt{match}\ e_0\ \texttt{with}\ p_1 \to e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n \to e_n]$$

Type information is propagated bidirectionally:

$\downarrow$ we infer the type for e_0, and keep the safe part to type patterns

For each case $p_i \to e_i$

$\downarrow$ we type e_i using the equations generated by typing p_i

$\downarrow$ we canonicalize the resulting type (and other types shared with the environment), expanding all equations

$\uparrow$ we unify with the safe part of the type inferred for C's hole

Subtle points about canonicalization

```
type _ t = I : int t

let f1 (type a) (x : a t) y =
  let y = (y : a) in                  (* safe type annotation *)
  match x with I -> (y : a)           (* a canonicalized to int *)
val f : 'a t -> 'a -> int
let f2 (type a) (x : a t) y =
  let r = match x with I -> (y : a) in
  ignore (y : a);                     (* y has type int *)
  r
let f3 (type a) (x : a t) y =
  ignore (y : a);                     (* unsafe type annotation *)
  match x with I -> (y : a)
```

`f1` succeeds, but `f2` fails.

Since there is no propagation in `f3`, it must fail too.

How to canonicalize properly

– Canonicalization of types depends on where they were defined

  ◦ `a` canonicalizes to `int` only if we had the equation `a = int` at the annotation point

  ◦ but canonicalization may occur in another context

– Implementation using OCaml's type level mechanism

  ◦ levels grow when we enter binding constructs
  ◦ for every equation in the environment, remember its level
  ◦ only use an equation if the level of the type is at least the level of the equation
  ◦ generalized types get duplicated at use sites
  ◦ canonicalize a type when we lower its level (already done for local modules)

How principal?

- Due to our use of canonicalization, we cannot hope for real principality

- If we require some derivations to be minimal (*i.e.* infer the most general type), then we can recover principality.
  (Similar to first-class polymorphism with value restriction)

- Good symmetry properties: changing the order of subexpressions should not change the outcome of type inference.

- Small drawback: due to canonicalization, type annotations inside a branch of pattern-matching do not help typing its result.

Inference power

- Up to now, all examples could be typed without adding annotations inside functions.
- Could type almost all examples in the Omega Tutorial [1], adding only function types, and omitting all impossible cases. (Since we do not have Omega's type level functions, some examples cannot be expressed)

[1] Tim Sheard and Nathan Linger: *Programming In Omega*. Notes from the 2nd Central European Functional Programming School, 2007.

Comparison to Wobbly Types (i.e. GHC 6)

Glasgow Haskell had already GADTs for 7 years. Wobbly types are the original approach.

Come in two versions.

- the original version is very close to what we do, but described in terms of unification and substitution rather than sharing. Propagation is weaker in the basic system, but gets stronger with "smart application", so the power seems close

- [POPL 06] version is simpler, wobbliness being a property of terms rather than types, but this makes propagation weaker

Using sharing makes possible a somehow cleaner specification.

Comparison to OutsideIn (i.e. GHC 7)

GHC 7 uses a constraint-based approach to inference, which allows good properties.

- Proceeds by first typing the function ignoring all match cases, then propagating external information to type them.
- The types they infer are always principal in the naive type system (but not complete).
  This is not our case, since canonicalization may choose between two different types (in a deterministic way).
- Like us, complete with respect to a specification.
- In some cases, able to infer types without any type annotation.

This seems very powerful, but requires a reimplementation of type inference.

Non-principal example

The following example is taken from OutsideIn.

```
type _ t =
  | T1 : int -> bool t
  | T2 : 'a t

let test (type a) (x : a t) r =
  match x with
  | T1 n -> n > 0
  | T2   -> r
val test : 'a t -> bool -> bool
```

The type we infer here is not principal since the following type, which is not comparable, would also be valid:

```
val test : 'a t -> 'a -> 'a
```

Note that while OutsideIn rightly rejects this example,
GHC 7 accepts it for practical reasons :-)

What is principality about

Principality has two roles.

- When a solution is principal, this guarantees that there is no ambiguity about its choice.
  Important for Haskell, but OCaml has untyped semantics.

- When we also have weakening with respect to hypotheses, this allows modular type inference.

$$\Gamma \vdash e : \tau \text{ and } \Gamma' \leq \Gamma \text{ implies } \Gamma' \vdash e : \tau$$

- However, inference systems for GADTs lack weakening.

- If we don't care about ambiguity, principality modulo some minimal derivations is not really worse than full principality.

Future improvements

- **Allow giving names to freshly introduced existential types**
  Currently there is no way to give names locally.

- **Make the typing more principal**
  An idea (with Didier Rémy) is to keep track of which equations have been used for a type.

  - See inferred types as sets of related types.
  - Do not allow an inferred type to escape to an environment where it would be incoherent.

  This recovers principality with respect to the naive type system, and could permit sharing for objects and variants.

Enjoy

- About the implementation:
  `https://sites.google.com/site/ocamlgadt/`
- Code is already in the trunk, for the next release of OCaml:
  `svn checkout http://caml.inria.fr/svn/ocaml/trunk`
- Examples: `testsuite/tests/typing-gadts`
  - `test.ml` : basic cases
  - `omega07.ml` : examples translated from Omega
- Do not forget to use `ocaml -principal` for predictable behavior!