Interpreting OCaml GADTs into Coq

Jacques Garrigue, Takafumi Saikawa

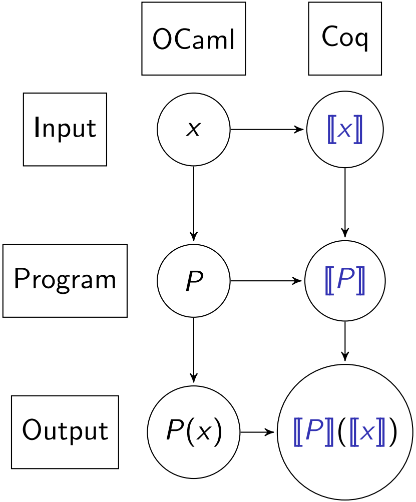
Graduate School of Mathematics, Nagoya University

ML Workshop, September 15, 2022

Starting point : the Coqgen project

- Proving the correctness of the full OCaml type inference is hard
- We can prove it theoretically for subparts, but combining them is complex
- Writing a type checker for the typed syntax tree might help, but still suffers the same difficulties
- Alternative approach: ensure that the generated typed syntax trees enjoys type soundness by translating them into another type system, here Coq

Soundness by translation



If for all $P : \tau \to \tau'$ and $x : \tau$

- P translates to $[\![P]\!]$, and $\vdash [\![P]\!] : [\![\tau \to \tau']\!]$
- x translates to $[\![x]\!]$, and $\vdash [\![x]\!] : [\![\tau]\!]$
- $[\![P]\!]$ applied to $[\![x]\!]$ evaluates to $[\![P(x)]\!]$
- $[\![\cdot]\!]$ is injective (on types)

then the soundness of Coq's type system implies the soundness of OCaml's evaluation

Translating GADTs

Our backend is already able to translate many features

- Core ML : λ-calculus with polymorphism and recursion
- algebraic data types
- references and exceptions
- while and for loops, lazy values, etc...

However, GADTs present specific challenges

- generation of equations between types
- usage of these equations for coercion
- pruning of unreachable branches

In this presentation we study how to do it, using hand-translated code fragments.

A classical example

GADTs allow one to encode the length of vectors.

```
(* encoding of natural numbers into types *)
type zero = Zero
type 'a succ = Succ of 'a
type (_,_) vec =
    Nil  : ('a, zero) vec
  | Cons : 'a * ('a,'n) vec -> ('a, 'n succ) vec

(* map perserves the length *)
let rec map : type a b n. (a -> b) -> (a,n) vec -> (b,n) vec =
  fun f -> function
    | Nil         -> Nil
    | Cons (a, l) -> Cons (f a, map f l)

(* head is exhaustive on vectors of length at least 1 *)
let head : type a n. (a,n succ) vec -> a = function Cons (a, _) -> a
```

The naive translation

We just translate GADTs to Coq inductive types, and rely on Coq's pattern matching construct for equation handling.

```
Inductive zero := Zero.
Inductive succ (n : Type) := Succ of n.
Inductive vec (A : Type) : Type -> Type :=
  | Nil    : vec A zero
  | Cons n : A -> vec A n -> vec A (succ n).

Fixpoint map (A B n : Type) (f : A -> B) (l : vec A n):=
  match l in vec _ n return vec B n with
  | Nil _       => Nil B
  | Cons _ n a l => Cons B n (f a) (map A B n f l)
  end.
```

We only need to annotate the `match` construct explicitly.

What about head ?

```
Fail Definition head A n (l : vec A (succ n)) : A :=
  match l with
  | Cons _ n a l => a
  end.
Non exhaustive pattern-matching: no clause found for pattern
Nil _
```

The naive translation of head fails.

- There is no way to prove that zero is different from succ n inside **Type**.

What about head ?

```
Fail Definition head A n (l : vec A (succ n)) : A :=
  match l with
  | Cons _ n a l => a
  end.
Non exhaustive pattern-matching: no clause found for pattern
Nil _
```

The naive translation of head fails.

- There is no way to prove that zero is different from succ n inside Type.

Yet it works if we define vec with its length index in nat, as Z and S n differ.

```
Inductive vec (A : Type) : nat -> Type := ...
```

The difference between GADTs and inductive types

- GADT indices are simultaneoulsy types and first-order terms.
  Pattern-matching results in unification:
  - generating equations in case of success;
  - pruning unreachable branches in case of failure.

- Indices of inductive types are either types or values of a type.
  Pattern-matching results in:
  - substitution if the `in pattern` syntax was used;
  - pruning of unreachable branches, by discriminating on the head constructor, if the index belongs to an inductive type.

The difference between GADTs and inductive types

- GADT indices are simultaneoulsy types and first-order terms.
  Pattern-matching results in unification:
  - generating equations in case of success;
  - pruning unreachable branches in case of failure.
- Indices of inductive types are either types or values of a type.
  Pattern-matching results in:
  - substitution if the `in pattern` syntax was used;
  - pruning of unreachable branches, by discriminating on the head constructor, if the index belongs to an inductive type.
- Changing an index from `Type` to a specific inductive type is not always sufficient, as GADT indices are essentially both.

Intensional translation

Our solution to eat our cake and keep it, is to give both syntactical and semantical representations to OCaml types.

```
Require Import ssreflect.
Inductive ml_type : Set :=
  | ml_int
  | ml_bool
  | ml_arrow of ml_type & ml_type
  ...
  | ml_zero
  | ml_succ  of ml_type
  | ml_vec   of ml_type & ml_type.
```

Here we use `ssreflect` syntax, closer to ML.

Extensional definitions

Type definitions still need to be translated.

```
Inductive zero := Zero.
Inductive succ (n : Type) := Succ of n.
Inductive vec (A : Type) (n : ml_type) :=
  | Nil     of n = ml_zero
  | Cons m of n = ml_succ m & A & vec A m.
```

They differ from the naive translation in that

- Equations are explicit. (Avoids convoy pattern.)
- Type parameters are in `Type` if used for values, in `ml_type` if used in equations, or duplicated if both.

Type interpretation function

The two tranlations are connected by an interpretation function, which must be fully computable.

```
Fixpoint coq_type (T : ml_type) : Type :=
 match T with
 | ml_int          => Int63.int
 | ml_bool         => bool
 | ml_arrow T1 T2 => coq_type T1 -> coq_type T2
 ...
 | ml_zero         => zero
 | ml_succ T1      => succ (coq_type T1)
 | ml_vec T1 T2    => vec (coq_type T1) T2
 end.
```

Note how in the `ml_vec` case only the first type parameter is interpreted.

Translation of terms

The translation is a bit more verbose. (Lots of coq_type needed)

```
Fixpoint map (T1 T2 T3: ml_type) (f: coq_type T1 -> coq_type T2)
         (l: vec (coq_type T1) T3) : vec (coq_type T2) T3 :=
  match l in vec _ n return vec (coq_type T2) n with
  | Nil _ => Nil _
  | Cons _ m a l => Cons _ m (f a) (map T1 T2 _ f l)
  end.
```

Translation of terms

The translation is a bit more verbose. (Lots of coq_type needed)

```
Fixpoint map (T1 T2 T3: ml_type) (f: coq_type T1 -> coq_type T2)
         (l: vec (coq_type T1) T3) : vec (coq_type T2) T3 :=
  match l in vec _ n return vec (coq_type T2) n with
  | Nil _ => Nil _
  | Cons _ m a l => Cons _ m (f a) (map T1 T2 _ f l)
  end.

Definition head (T1 T2 : ml_type) (l : vec (coq_type T1) (ml_succ T2))
  : coq_type T1 :=
  match l with
  | Nil _ _ H => match H with end
  | Cons _ _ _ _ a _ => a
  end.
```

However we are now able to disprove unreachable branches.

Injectivity of type constructors

This translation also supports injectivity, contrary to the naive one.

```
type (_, _) eq = Refl : ('a,'a) eq
let succ_inj : type n1 n2. (n1 succ, n2 succ) eq -> (n1, n2) eq
  = function Refl -> Refl
```

One just needs to apply add hoc projections.

```
Inductive eqw (T1 T2 : ml_type) := Refl of T1 = T2.
Definition eqw_eq [x y] (w : eqw x y) : x = y :=
  match w with Refl _ _ H => H end.

Definition proj_ml_succ defT T :=
  if T is ml_succ T1 then T1 else defT.
Definition succ_inj n1 n2 (w : eqw (ml_succ n1) (ml_succ n2)) : eqw n1 n2 :=
  Refl _ _ (f_equal (proj_ml_succ n1) (eqw_eq w)).
```

Mixed use of existential type variables

While type parameters can be both intensional and extensional, existential type variables create problems with recursion.

```
type _ hlist =         (* Heterogeneous list *)
  | HNil  : zero hlist
  | HCons : 'a * 'b hlist -> ('a * 'b) hlist

Inductive hlist (coq_type : ml_type -> Type) T :=
  | HNil of T = ml_zero
  | HCons T1 T2 of
    T = ml_pair T1 T2 & coq_type T1 & hlist ct T2.
#[bypass_check(guard)]
Fixpoint coq_type (T : ml_type) : Type := ...
  | ml_hlist T1 => hlist coq_type T1
  ...
```

Since hlist depends now on coq_type, we need to bypass the termination check.

Related work

- Guillaume Claret. *Coq of OCaml*. OCaml Workshop, 2014.
- Antal Spector-Zabusky *et al. Total Haskell is reasonable Coq*. CPP, 2018.
- Danil Annenkov *et al. ConCert: a smart contract certification framework in Coq*. CPP, 2020.
- Laila El-Beheiry *et al*. SMLtoCoq: Automated Generation of Coq Specifications and Proof Obligations from SML Programs with Contracts. LFMTP, 2021.
- Matthieu Sozeau *et al. Coq Coq correct! verification of type checking and erasure for Coq, in Coq*, POPL, 2020.
- Pierrick Couderc. *Vérification des résultats de l'inférence de types du langage OCaml.* PhD Thesis, Université Paris-Saclay, 2018.

Towards a type-sound transpiler from OCaml to Coq

Automating the translation of GADTs still requires

- Obtaining a trace of how the compiler generated equations
- And also where and how it uses them
- Neither is currently available in OCaml

Coqgen already supports many features, including side-effects.

- As a result, the translation of ml_arrow is

```
| ml_arrow T1 T2 => coq_type T1 -> M (coq_type T2)
```

  for some monad M, requiring some bootstrapping too.

Many other open problems

- How to represent abstract types, as they may be not injective?

For more information see

http://www.math.nagoya-u.ac.jp/~garrigue/cocti/