Interpreting OCaml GADTs into Coq

Jacques Garrigue
Nagoya University
Graduate School of Mathematics
Japan

Takafumi Saikawa
Nagoya University
Graduate School of Mathematics
Japan

Abstract

GADTs [1, 9] have become a common feature of strongly typed functional programming languages. They are often presented as a weaker form of the inductive types seen in Coq or Agda. In both cases, constraints generated by pattern-matching allow one to generate equations that can be applied to types. However, there is an important difference: in OCaml or Haskell, unification can be used to refine syntactic equations on types, while this is not the case in Coq or Agda, where one cannot observe that type constructors are syntactically injective. This is a problem when one wants to translate code from a functional programming language to a type-theory based proof assistant, as a literal translation would not allow one to infer the necessary equations. This may be one of the reasons why existing automatic translators provide little or no support for GADTs [2, 7].

In this presentation we show how to avoid this problem by using a two-pronged translation, where OCaml types are mapped to an intensional representation, that preserves the deductive properties, which is itself interpreted into concrete Coq types, that allow computation. We also discuss what is further needed for an automatic translator.

1 Naive translation

Let us consider the standard example for GADTs: a well-typed interpreter, in OCaml syntax.

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

let rec eval : type a. a expr -> a = function
  | Int n -> n
  | Add -> (+)
  | App (f, x) -> eval f (eval x)
```

One can see the expressive power of using types as indices in this trivial example, where one constructs a concrete value matching a type index.

```
let add = eval Add
val add : int -> int -> int = <fun>
```

One can easily translate this GADT into a Coq inductive type, and even `eval` types fine.

```
Require Import Int63.
Inductive expr : Type -> Type :=
  | Int : int -> expr int
  | Add : expr (int -> int -> int)
  | App a b : expr (a -> b) -> expr a -> expr b.

Fixpoint eval (a : Type) (e : expr a) : a :=
  match e in expr a return a with
  | Int n => n
  | Add => Int63.add
  | App b c f x => eval (b -> c) f (eval b x)
  end.
```

One can also translate equality witnesses and the associated `cast` function.

```
type (_,_) eq = Refl : ('a,'a) eq
let cast : type a b. (a, b) eq -> a -> b =
  function Refl -> fun x -> x
```

```
Inductive eqw (A : Type) : A -> A -> Type :=
  | Refl x : eqw A x x.
Definition cast A B (w : eqw Type A B) :=
  match w in eqw _ A B return A -> B
  with Refl _ _ => fun a => a end.
```

However, problems start when one wants to extract an equality through unification.

```
let cast_fst : type a b.
  (a * b, int * bool) eq -> a -> int =
  function Refl -> fun x -> x
```

There is basically no way to do that in Coq, as there is no proof of the injectivity of type parameters for the type of pairs.

A similar problem occurs with the classical encoding of fixed-length vectors.

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

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)

let head : type a n. (a,n succ) vec -> a =
  function Cons (a, _) -> a
```

Note that here the concrete value constructors for `zero` and `succ` do not matter, since we only use these types abstractly.

The function map uses type-level unification to ensure that the length is kept. In head, we again use unification to discard the Nil case as unreachable.

We can translate vec and map without problem.

```
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 (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.
```

However there is no way to translate head as long as zero and succ are translated as types, since discarding Nil requires a proof that zero cannot be equal to succ n, which cannot be done in Coq. More precisely, a proof of semantic inequality would require zero and succ n to have different cardinality [8, Theorem 8.3.2], which is not the case here.

2 Intensional translation

The cause of our problems is that in OCaml datatypes are injective in their type parameters, and can be structurally distinguished in most cases, for instance by value constructors with different names, while in Coq this is not the case.

Our solution is to translate OCaml types to Coq data, and recover Coq types through an interpretation function. This approach is reminiscent of intensional type analysis [3, 6], in the heterogeneous case where we are interpreting OCaml types inside Coq.

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

Here we use the ssreflect syntax for inductive types, which happens to be closer to ML.

Before defining the interpretation, we define our GADTs, using explicit equations for clarity, and easier automation[1].

```
Inductive eqw (T1 T2 : ml_type) :=
  | Refl of T1 = T2.

Inductive expr (T : ml_type) :=
  | Int of T = ml_int & int
  | Add of T = ml_arrow ml_int
               (ml_arrow ml_int ml_int)
  | App T2 of expr (ml_arrow T2 T) & expr T2.

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.

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_pair T1 T2 => coq_type T1 * coq_type T2
 | ml_eqw T1 T2 => eqw T1 T2
 | ml_expr T1 => expr T1
 | ml_zero => zero
 | ml_succ T1 => succ (coq_type T1)
 | ml_vec T1 T2 => vec (coq_type T1) T2
 end.
```

Note in vec how parameters of inductive types may either be Coq types, when they represent concrete values, or ML types, when they are used in equations. The same parameter might even be duplicated if it is used in both ways, with coq_type ensuring that the two version are synchronized. In functions, we will uniformly use ml_type for polymorphism, as we can now use coq_type to interpret it.

While this translation is more verbose, as we are now using equations explictly, no expressive power is lost.

```
Fixpoint eval (T : ml_type) (e : expr T)
  : coq_type T :=
  match e with
  | Int _ H n => eq_rect _ _ n _ (eq_sym H)
  | Add _ H => eq_rect _ _ Int63.add _ (eq_sym H)
  | App _ H f x => eval _ f (eval _ x)
  end.

Definition eqw_eq [x y] (w : eqw x y) : x = y :=
  match w with Refl _ _ H => H end.

Definition cast T1 T2 (w : eqw T1 T2)
  (x : coq_type T1) : coq_type T2 :=
  eq_rect T1 coq_type x T2 (eqw_eq w).
```

Moreover, it becomes possible to properly extract equations through injectivity.

```
Definition proj_ml_pair_1 T0 T :=
  match T with ml_pair T1 _ => T1 | _ => T0 end.

Definition cast_fst (A B : ml_type)
  (w : eqw(ml_pair A B)(ml_pair ml_int ml_bool))
```

[1] A style closer to the naive translation, leaving equations implicit, is also possible, but arguably less predictable.

```
  (x : coq_type A) : int :=
  eq_rect A coq_type x ml_int
    (f_equal (proj_ml_pair_1 A) (eqw_eq w)).
```

One can also discard impossible cases by proving contradictions. In simple cases, an empty `match` statement suffices.

```
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.
```

3 Bad recursion?

Some type definitions use type variables both intensionally and as Coq types.

For type parameters themselves, this can be handled through duplication, `coq_type` keeping the copies synchronized.

```
type 'a result = Res of 'a expr * 'a
```

```
Inductive result (T : ml_type) (A : Type) :=
  | Result of expr T & A.
Fixpoint coq_type (T : ml_type) : Type := ...
  | ml_result T1 => result T1 (coq_type T1)
  ...
```

However, this trick does not work with existential type variables. Let us conside for instance the type `hlist` of heterogeneous lists.

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

The translation to Coq requires using `coq_type` inside the definition itself, as `T1` is used concretely, but can only be obtained by using the injectivity of `ml_pair`.

```
Inductive hlist (ct : ml_type -> Type) T :=
  | HNil of T = ml_zero
  | HCons T1 T2 of
    T = ml_pair T1 T2 & ct T1 & hlist ct T2.
```

We can still define `coq_type`, but we need to bypass the termination check.

```
#[bypass_check(guard)]
Fixpoint coq_type (T : ml_type) : Type := ...
  | ml_hlist T1 => hlist coq_type T1
  ...
```

Note that this recursion is rather special, as the recursive call to `coq_type` is delayed until we actually access the contents of the `hlist`.

4 Towards a translation of OCaml

While the translation of GADTs is still experimental, we have already used this combination of intensional representation and its interpretation to build a working translator for the core features of OCaml, including polymorphism, state, exceptions, and polymorphic comparison [4]. In that case, the translation of `ml_arrow T1 T2` is not `coq_type T1 → coq_type T2`, but rather `coq_type T1 → M (coq_type T2)` for a carefully crafted monad `M`. This part of the work was presented at TYPES [5].

We intend to use this type preserving translation to prove emprirically the soundness of OCaml type inference, by relying on Coq's subject reduction. Proving properties of translated programs is a secondary goal.

Many issues are still open. For instance, how should one represent abstract types, which may be non-injective, inside `ml_type`. The encoding of non-positive recursive types and equi-recursive types is also challenging.

Aknowledgements. We thank the anonymous reviewers for their insightful comments. This research is supported by grants from the Tezos Foundation and the Japanese Society for the Promotion of Science (KAKENHI number 22K11902).

References

[1] James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, January 2003.

[2] Guillaume Claret. Coq of OCaml. In *OCaml Users and Developers Meeting*, August 2014.

[3] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming*, 12(6):567–600, 2002.

[4] Jacques Garrigue. OCaml in Coq. GitHub PR, 2022. https://github.com/COCTI/ocaml/pull/3.

[5] Jacques Garrigue and Takafumi Saikawa. Validating OCaml soundness by translation into Coq. In *Proc. 28th International Conference on Types for Proofs and Programs*, June 2022.

[6] Bratin Saha, Valery Trifonov, and Zhong Shao. Intensional analysis of quantified types. *ACM Trans. Program. Lang. Syst.*, 25(2):159–209, March 2003.

[7] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In *Proc. International Conference on Functional Programming*, pages 14—-27, New York, NY, USA, 2018.

[8] Théo Winterhalter. *Formalisation and meta-theory of type theory*. PhD thesis, Université de Nantes, 2020.

[9] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proc. 30th Symposium on Principles of Programming Languages*, POPL '03, page 224–235, New York, NY, USA, 2003. Association for Computing Machinery.