A Translation of OCaml GADTs into Coq

Pedro da Costa Abreu Junior
Purdue University
September 24, 2024

Nomadic Labs

nomadic labs

Tezos

nomadic labs

Tezos

Michelson

nomadic labs

Tezos

Michelson

Coq-Of-Ocaml

nomadic labs

Tezos

Michelson

coq-of-ocaml

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- GADTs

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- GADTs
- Inductive Types (with dependent types)

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- GADTs
- Inductive Types (with dependent types)
- Compiler Correctness

Example of an ADT

```
type term =
  | T_Int : nat -> term
  | T_Bool : bool -> term
  | T_Add : term * term -> term
```

Example of an ADT

```
type term =
  | T_Int : nat -> term
  | T_Bool : bool -> term
  | T_Add : term * term -> term

let get_bool (bexp : term) : bool option = function
  match bexp with
  | T_Bool b -> Some b
  | _ -> None
```

Example of GADT

```
type _ term =
  | T_Int : nat -> nat term
  | T_Bool : bool -> bool term
  | T_Add : nat term * nat term -> nat term
```

Example of GADT

```
type _ term =
  | T_Int : nat -> nat term
  | T_Bool : bool -> bool term
  | T_Add : nat term * nat term -> nat term

let get_bool (bexp : bool term) : bool = function
  match bexp with
  | T_Bool b -> b
```

Example of GADT

```
type _ term =
  | T_Int : nat -> nat term
  | T_Bool : bool -> bool term
  | T_Add : nat term * nat term -> nat term
```

Example of GADT

```
type _ term =
  | T_Int : nat -> nat term
  | T_Bool : bool -> bool term
  | T_Add : nat term * nat term -> nat term

let rec eval (type a) (t : a term) : a =
  match t with
  | T_Int n -> n
  | T_Bool b -> b
  | T_Add (x, y) -> (eval x) + (eval y)
```

Impossible Branches in Coq

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .
```

Impossible Branches in Coq

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Definition get_bool (t : term bool) : bool :=
  match t with
  | T_bool b ⇒ b
  end.
```

Impossible Branches in Coq

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Definition get_bool (t : term bool) : bool :=
  match t with
  | T_bool b ⇒ b
  end.
```

Error: Non exhaustive pattern-matching: no clause found for pattern T_int _

Impossible Branches in Coq

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Axiom unreachable_gadt_branch : forall (A : Type), A.
```

Impossible Branches in Coq

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Axiom unreachable_gadt_branch : forall (A : Type), A.

Definition get_bool (t : term bool) : bool :=
  match t with
  | T_bool b ⇒ b
  | _ ⇒ unreachable_gadt_branch
  end.
```

Dependent Pattern Matching

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Definition get_bool (t : term bool) : bool :=
  match t in term A return A = bool → bool with
  ...
  end eq_refl.
```

Dependent Pattern Matching

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Definition get_bool (t : term bool) : bool :=
  match t in term A return A = bool → bool with
  | T_bool b ⇒ fun (h : bool = bool) ⇒ b
  ...
  end. eq_refl.
```

Dependent Pattern Matching

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Definition get_bool (t : term bool) : bool :=
  match t in term A return A = bool → bool with
  | T_bool b ⇒ fun _ ⇒ b
  | _ ⇒ fun (h : nat = bool) ⇒
    Principle of Explosion
  end eq_refl.
```

Dependent Pattern Matching

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Lemma bnat_neq : nat <> bool. Proof. ... Qed.
```

Dependent Pattern Matching

```
Inductive term : Set → Type :=
| T_int : nat → term nat
| T_bool: bool → term bool
| T_add : term nat → term nat → term nat .

Lemma bnat_neq : nat <> bool. Proof. ... Qed.

Definition get_bool (t : term bool) : bool :=
  match t in term A return A = bool → bool with
  | T_bool b ⇒ fun _ ⇒ b
  | _ ⇒ fun (h : nat = bool) ⇒
    ltac:(apply False_ind; apply (bnat_neq h))
  end eq_refl.
```

GADTs $\neq$ Inductive Types

```
type _ udu =
  | Unit : unit udu
  | Double_unit : (unit * unit) udu

let unit_twelve (x : unit udu) =
  match x with
  | Unit -> 12
```

GADTs $\neq$ Inductive Types

```
Inductive udu : Set → Type :=
  | Unit : udu unit
  | Double_unit : udu (unit * unit).
```

GADTs ≠ Inductive Types

```
Inductive udu : Set → Type :=
  | Unit : udu unit
  | Double_unit : udu (unit * unit).

Definition unit_twelve (x : udu unit) : nat.
  refine(match x in udu T return T = unit → nat with
  | Unit ⇒ fun h ⇒ 12
  | Double_unit ⇒ fun (h : unit * unit = unit) ⇒ _
  end eq_refl).
```

GADTs $\neq$ Inductive Types

However, `unit*unit` $=$ `unit` in Homotopy Type Theory. Since we know that HTT is consistent with CIC, we cannot discharge this impossible branch.

GADTs $\neq$ Inductive Types

The heart of the problem is that in OCaml, if two types have different declarations, they're automatically considered different from each other.

GADTs $\neq$ Inductive Types

The heart of the problem is that in OCaml, if two types have different declarations, they're automatically considered different from each other.

. . .

GADTs $\neq$ Inductive Types

The heart of the problem is that in OCaml, if two types have different declarations, they're automatically considered different from each other.

...

But that's not necessarily true in Coq.

GADTs $\neq$ Inductive Types

The heart of the problem is that in OCaml, if two types have different declarations, they're automatically considered different from each other.

. . .

But that's not necessarily true in Coq.

The main goal of my MSc Thesis is to bridge this gap!

A Universe for GADTs

We begin by embedding every type constructor used by a GADT into a new type GSet.

A Universe for GADTs

We begin by embedding every type constructor used by a GADT into a new type GSet.

```
Inductive GSet : Set :=
| G_arrow : GSet → GSet → GSet
| G_tuple : GSet → GSet → GSet
| G_tconstr : nat → Set → GSet.
```

A Universe for GADTs

We begin by embedding every type constructor used by a GADT into a new type GSet.

```
Inductive GSet : Set :=
| G_arrow : GSet → GSet → GSet
| G_tuple : GSet → GSet → GSet
| G_tconstr : nat → Set → GSet.

Fixpoint decodeG (s : GSet) : Set :=
  match s with
  | G_tconstr s t ⇒ t
  | G_arrow t1 t2 ⇒ decodeG t1 → decodeG t2
  | G_tuple t1 t2 ⇒ (decodeG t1) * (decodeG t2)
  end.
```

A Universe for GADTs

```
Definition G_unit := G_tconstr 0 unit.

Inductive udu : GSet → Set :=
| Unit : udu G_unit
| Double_unit : udu (G_tuple G_unit G_unit).
```

A Universe for GADTs

```
Definition G_unit := G_tconstr 0 unit.

Inductive udu : GSet → Set :=
| Unit : udu G_unit
| Double_unit : udu (G_tuple G_unit G_unit).

Definition unit_twelve (x : udu G_unit) : nat :=
  match x in udu s0 return s0 = G_unit → nat with
   | Unit ⇒ fun eq0 ⇒ 12
   | _ ⇒ fun (neq : G_tuple G_unit G_unit = G_unit) ⇒
     ltac:(discriminate)
  end eq_refl.
```

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- ✓ GADTs
- ✓ Inductive Types (with dependent types)
- ■ Compiler Correctness

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- ✓ GADTs
- ✓ Inductive Types (with dependent types)
- Compiler Correctness
  - Specification of the Syntaxes
  - Specification of the Type Systems
  - Specification of the Translation
  - Proof of Type-Preservation

GADTml Syntax

$$
\begin{array}{llll}
s & ::= & \forall a.s \mid t & \textit{Types} \\
t, u & ::= & a \mid t \rightarrow t \mid t * t \mid T\ \overline{t} & \textit{Monotype} \\
e & ::= & x \mid \lambda x : t.e \mid e\ e & \textit{Expression} \\
 & \mid & \Lambda a.e \mid e[t] \mid (e, e) & \\
 & \mid & \text{match } e \text{ with } \overline{\mid K\ \overline{x} \rightarrow e'} & \\
dcl & ::= & \text{type } T\ \overline{a}\ := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \rightarrow T\ \overline{a}} & \textit{ADT Declaration} \\
 & \mid & \text{gadt } G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \rightarrow G\ \overline{v}} & \textit{GADT Declaration} \\
p & ::= & \overline{dcl}; e & \textit{Program}
\end{array}
$$

Figure: GADTml Syntax

GADTml Typing

$$\frac{\Sigma;\Gamma \vdash e : T\ \overline{u} \qquad \Sigma;\Gamma \vdash t : * \qquad \text{type } T\ \overline{a} := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \to T\ \overline{a}} \in \Sigma \qquad \left\{ \Sigma;\Gamma, \overline{a, b, x_i : t_i} \vdash e'_i : t \right\}_{K_i}}{\Sigma;\Gamma \vdash \text{match } e \text{ with } \overline{\mid K_i\ \overline{x_i} \to e'} : t} \ (\textsc{TyMatch})$$

$$\frac{\Sigma;\Gamma \vdash e : G\ \overline{u} \qquad \Sigma;\Gamma \vdash t : * \qquad \text{gadt } G\ \overline{a} := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma \qquad \left\{ \begin{array}{c} \Sigma;\sigma_i(\Gamma, \overline{b, x_i : t_i}) \vdash e'_i : \sigma_i(t) \\ \sigma_i \equiv \text{unifies}(\overline{u}, \overline{v_i}) \not\equiv \bot \end{array} \right\}_{K_i}}{\Sigma;\Gamma \vdash \text{match } e \text{ with } \overline{\mid K_i\ \overline{x_i} \to e'} : t} \ (\textsc{TyGMatch})$$

GADTml Unification

$$
\begin{aligned}
\text{unifies}([\,],[\,]) &\triangleq [\,]\\
\text{unifies}(x;\ \overline{t}, s;\ \overline{s}) &\triangleq [s/x];\ \text{unifies}(\overline{t}[s/x], \overline{s}[s/x])\\
\text{unifies}(t;\ \overline{t}, x;\ \overline{s}) &\triangleq [t/x];\ \text{unifies}(\overline{t}[t/x], \overline{s}[t/x])\\
\text{unifies}(T\ \overline{u}; \overline{t},\ \ T\ \overline{v}; \overline{s}) &\triangleq \text{unifies}(\overline{u}; \overline{t},\ \ \overline{v}; \overline{s})\\
\text{unifies}(t_1 \rightarrow t_2; \overline{t},\ \ s_1 \rightarrow s_2; \overline{s}) &\triangleq \text{unifies}(t_1; t_2; \overline{t},\ \ s_1; s_2; \overline{s})\\
\text{unifies}(\_, \_) &\triangleq \bot
\end{aligned}
$$

gCIC Syntax

$$
\begin{array}{lcll}
T, e & ::= & x \mid \lambda x : A.e \mid e\ e \mid T\ \overline{v} & \textit{Expressions} \\
 & \mid & \forall(a : A), t \mid Set & \\
 & \mid & \text{let } (x : t) = e \text{ in } e & \\
 & \mid & \text{match } e \text{ in } T\ \overline{a} \text{ return } t \text{ with} & \\
 & & \quad \overline{\mid K\ \overline{x} \Rightarrow e' } \text{ end} & \\
decl & ::= & \text{Inductive } T\ \Xi : \Delta \to \text{Set} \ := & \textit{Inductive Types} \\
 & & \overline{\mid K : \Delta \to T\ \overline{v}} & \\
prog & ::= & \overline{decl}; e & \textit{Program}
\end{array}
$$

gCIC Typing

$$\frac{\text{Inductive } T\ \Xi : \Delta \to \text{Set} \ := \overline{\mid K : \Delta \to T\ \overline{v}} \in \Sigma \qquad \Sigma; \Gamma \vdash \overline{u} : \Xi \qquad \Sigma; \Gamma \vdash \overline{v} : \Delta}{\Sigma; \Gamma \vdash T\ \overline{u\ v} : \text{Set}} \ (\text{CTyTyFam})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash e : T\ \overline{u} \\ \Sigma; \Gamma, \overline{a} : \Delta \vdash t : s \\ \text{Inductive } T\ \Xi : \Delta \to \text{Set} \ := \overline{\mid K : \Delta \to T\ \overline{v}} \in \Sigma \\ \{\ \Sigma; \Gamma, \overline{x_i} : \Delta_i \vdash e'_i : t[\overline{u_i}/\overline{a}]\ \}_{K_i}\end{array}}{\Sigma; \Gamma \vdash \text{match } e \text{ in } T\ \overline{a} \text{ return } t \text{ with } \overline{\mid K\ \overline{x} \ \Rightarrow\ e'} \text{ end} : t[\overline{u}/\overline{a}]} \ (\text{CTyMatch})$$

Translation

The translation process is divided in three phases:

Translation

The translation process is divided in three phases:

1. Transpilation
   - First translation into gCIC
   - Gathers information about GSet variables into a mapping ξ

Translation

The translation process is divided in three phases:

1. Transpilation
   - First translation into gCIC
   - Gathers information about GSet variables into a mapping ξ
2. Embedding
   - Moves necessary variables and declarations into GSet

Translation

The translation process is divided in three phases:

1. Transpilation
   - First translation into gCIC
   - Gathers information about GSet variables into a mapping ξ
2. Embedding
   - Moves necessary variables and declarations into GSet
3. Repair
   - Builds proof terms for casts and impossible branches

Transpilation Rules

Datatype Tranpilation

$$\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma$$

Variable Context Transpilation

$$\Sigma; \Delta \vdash \Gamma \rightsquigarrow \Gamma$$

Type Transpilation

$$\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$$

Expression Transpilation

$$\Sigma; \Gamma \vdash e : t \rightsquigarrow e \mid \xi$$

Type Transpilation

Type Transpilation

$$\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$$

Type Transpilation

$$\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$$

- Σ Map of datatype declarations
- Γ Map of variable types
- t Well-Kinded type being translated into t
- $\rightsquigarrow_g$ Points under which context the translation is happening.
  - Δ if GSet
  - $*$ otherwise
- ξ GSet Context

Type Variable Transpilation

$$\frac{\Sigma; \Gamma \vdash a : *}{\Sigma; \Gamma \vdash a : * \leadsto_{*} a \mid \{a : *\}}$$

$$\frac{\Sigma; \Gamma \vdash a : *}{\Sigma; \Gamma \vdash a : * \leadsto_{\Delta} a \mid \{a : \Delta\}}$$

GADT Pattern Matching Transpilation

$$
\dfrac{
\begin{array}{c}
\mathsf{gadt}\ G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma \\
\Sigma; \Gamma \vdash \mathrm{e} : G\ \overline{u} \rightsquigarrow \mathrm{e} \mid \xi_{\mathrm{e}} \qquad \Sigma; \Gamma \vdash G\ \overline{u} \rightsquigarrow_{*} G\ \overline{u} \mid \xi_{u} \\
\Sigma; \Gamma \vdash t : * \rightsquigarrow_{*} t \mid \xi_{t} \qquad \Sigma; \Gamma, \overline{a}, \overline{b} \vdash \overline{v : *} \rightsquigarrow_{\Delta} \overline{v} \mid \xi_{v} \\
\xi = (\bigsqcup \xi_{\mathrm{i}}) \sqcup \xi_{\mathrm{e}} \sqcup \xi_{u} \sqcup \xi_{v} \\
\left\{ \begin{array}{c|c}
\begin{array}{c} \Sigma; \sigma_{\mathrm{i}}(\Gamma, \overline{a}, \overline{b}, \overline{x_{\mathrm{i}} : t_{\mathrm{i}}}) \vdash \mathrm{e}'_{\mathrm{i}} : \sigma_{\mathrm{i}}(t) \rightsquigarrow \mathrm{e}'_{\mathrm{i}} \mid \xi_{\mathrm{i}} \\ \text{if } \sigma_{\mathrm{i}} \equiv \mathsf{unifies}(\overline{u}, \overline{v_{\mathrm{i}}}) \not\equiv \bot \end{array}
&
\begin{array}{c} \mathrm{e}'_{\mathrm{i}} = \mathsf{False} \\ \text{if } \mathsf{unifies}(\overline{u}, \overline{v_{\mathrm{i}}}) \equiv \bot \end{array}
\end{array} \right\}_{K_{\mathrm{i}}}
\end{array}
}{
\Sigma; \Gamma \vdash \mathsf{match}\ \mathrm{e}\ \mathsf{with}\ \overline{\mid K \overline{x} \to \mathrm{e}'}\ \mathsf{end} : t \rightsquigarrow
\left.\begin{array}{l}
\mathsf{match}\ \mathrm{e}\ \mathsf{in}\ G\ \overline{c} \\
\quad \mathsf{return}\ (\overline{c = u}) \to t\ \mathsf{with} \\
\quad \overline{\mid K\ \overline{x}\ \Rightarrow \lambda(h : v = u).\mathrm{e}'} \\
\mathsf{end}\ \overline{\mathsf{eq\_refl}}
\end{array}\right| \xi
}
\ (\textsc{TransGMatch})
$$

Running Example - Transpilation

```
gadt term a =
  | T_Int : int -> term int
  | T_Bool : bool -> term bool
  | T_Pair : forall l r.
    term l * term r -> term (l * r)

λ (e : term nat) =>
  match e with
  | T_Int n -> n
```

Running Example - Transpilation

```
Inductive term : GSet → Set :=
  | T_Int : nat → term nat
  | T_Bool : bool → term bool
  | T_Pair : ∀ (l :  Set),
    forall (r : Set), term l * term r → term (l * r)

λ (e : term nat).
  match e in term c return c = nat → nat with
  | T_Int n → λ (nat = nat). n
  | T_Bool b → λ (bool = nat). False
  | T_Pair l r p → λ (l * r = nat). False
  end eq_refl
```

Running Example - Transpilation

```
Inductive term : GSet → Set :=
  | T_Int : nat → term nat
  | T_Bool : bool → term bool
  | T_Pair : ∀ (l : Set),
    forall (r : Set), term l * term r → term (l * r)
```

$$\xi_{\Sigma} = [(\text{T\_Int}, \emptyset); (\text{T\_Bool}, \emptyset); (\text{T\_Pair}, \{(l : \Delta), (r : \Delta)\})]$$

Running Example - Transpilation

```
λ (e : term nat).
  match e in term c return c = nat → nat with
  | T_Int n → λ (nat = nat). n
  | T_Bool b → λ (bool = nat). False
  | T_Pair l r p → λ (l * r = nat). False
  end eq_refl
```

$$\xi \quad = \quad \{(l : \Delta), (r : \Delta)\}$$

ξ is a join-semilattice

We define a join operation $\xi_1 \sqcup \xi_2$
$\{a : *\} \sqcup \{a : \Delta\} = \{a : \Delta\}$, and therefore $\{a : *\} \leq \{a : \Delta\}$.

For different variables it behaves as regular set union
$\{a : *\} \sqcup \{b : \Delta\} = \{(a : *), (b : \Delta)\}$

Transpilation Lemma

Transpilation of expressions subsumes context of types

Lemma

If $\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi_t$ *and* $\Sigma; \Gamma \vdash e : t \rightsquigarrow e \mid \xi_e$ *then* $\xi_t \leq \xi_e$

Translation

1. Transpilation ✓
2. Embedding
   - Moves necessary variables and declarations into GSet
3. Repair

Embedding Phase

$$g\left[\textcolor{red}{—}\right]_{\xi}^{\Gamma}$$

Embedding Function

$$ {}^{*}[\mathit{Set}]^{\Gamma}_{\xi} = \mathit{Set} $$
$$ {}^{\Delta}[\mathit{Set}]^{\Gamma}_{\xi} = \mathit{GSet} $$
$$ {}^{*}[a]^{\Gamma}_{\xi} = \begin{cases} \text{decodeG } a & \text{if } (a : \Delta) \in \xi \\ a & \text{otherwise} \end{cases} $$

Embedding Phase

$$^{*}[T\ \overline{u}]^{\Gamma}_{\xi} = T\ ^{*}[\overline{u}]^{\Gamma}_{\xi}$$
$$^{\Delta}[T\ \overline{u}]^{\Gamma}_{\xi} = \text{G\_tconstr}\ (\#\Sigma(T))\ (T\ ^{*}[\overline{u}]^{\Gamma}_{\xi})$$

$$^{*}[G\ \overline{u}]^{\Gamma}_{\xi} = G\ ^{\Delta}[\overline{u}]^{\Gamma}_{\xi}$$
$$^{\Delta}[G\ \overline{u}]^{\Gamma}_{\xi} = \text{G\_tconstr}\ (\#\Sigma(G))\ (G\ ^{\Delta}[\overline{u}]^{\Gamma}_{\xi}])$$

Running Example - Embedding

$$* \left[\begin{array}{l} \texttt{Inductive term : GSet} \to \texttt{Set :=} \\ \quad \texttt{| T\_Int : nat} \to \texttt{term nat} \\ \quad \texttt{| T\_Bool : bool} \to \texttt{term bool} \\ \quad \texttt{| T\_Pair : } \forall \texttt{ (l : Set),} \\ \quad\quad \texttt{forall (r : Set), term l * term r} \to \texttt{term (l * r)} \end{array} \right]_{\xi}^{\Gamma}$$

$$=$$

```
Inductive term : GSet → Set :=
  | T_Int : nat → term (G_tconstr 0 nat)
  | T_Bool : bool → term (G_tconstr 1 bool)
  | T_Pair : ∀ (l : GSet), ∀ (r : GSet),
              term l * term r → term (G_tuple l r)
```

Running Example - Embedding

$$
* \left[\begin{array}{l}
\lambda\ (\texttt{e} : \texttt{term nat}). \\
\quad \texttt{match e in term c return c} = \texttt{nat} \to \texttt{nat with} \\
\quad \mid \texttt{T\_Int n} \to \lambda\ (\texttt{nat} = \texttt{nat}).\ \texttt{n} \\
\quad \mid \texttt{T\_Bool b} \to \lambda\ (\texttt{bool} = \texttt{nat}).\ \texttt{False} \\
\quad \mid \texttt{T\_Pair l r p} \to \lambda\ (\texttt{l} * \texttt{r} = \texttt{nat}).\ \texttt{False} \\
\quad \texttt{end eq\_refl}
\end{array} \right]^{\Gamma}_{\xi}
$$

$$=$$

$$
\begin{array}{l}
\lambda\ (\texttt{e} : \texttt{term}\ (\texttt{G\_tconstr}\ 0\ \texttt{nat})). \\
\quad \texttt{match e in term c return c} = \texttt{G\_tconstr}\ 0\ \texttt{nat} \to \texttt{nat with} \\
\quad \mid \texttt{T\_Int n} \to \lambda\ (\texttt{h} : \texttt{G\_tconstr}\ 0\ \texttt{nat} = \texttt{G\_tconstr}\ 0\ \texttt{nat}).\ \texttt{n} \\
\quad \mid \texttt{T\_Bool b} \to \lambda\ (\texttt{h} : \texttt{G\_tconstr}\ 1\ \texttt{bool} = \texttt{G\_tconstr}\ 0\ \texttt{nat}).\ \texttt{False} \\
\quad \mid \texttt{T\_Pair l r p} \to \lambda\ (\texttt{h} : \texttt{G\_tuple l r} = \texttt{G\_tconstr}\ 0\ \texttt{nat}).\ \texttt{False} \\
\quad \texttt{end eq\_refl}
\end{array}
$$

Translation

1. Transpilation ✓
2. Embedding ✓
3. Repair
   - Builds proof terms for casts and impossible branches

Repair

Injective and Conflict Properties

Constructors are injective

$K_{inj} : K\ \overline{e_1} = K\ \overline{e_2} \to \overline{e_1 = e_2}$.

Injective and Conflict Properties

Constructors are injective

$K_{inj} : K\ \overline{e_1} = K\ \overline{e_2} \rightarrow \overline{e_1 = e_2}$.

Implemented by the `inversion` tactic in Coq

Injective and Conflict Properties

Constructors are injective
$K_{inj} : K\ \overline{e_1} = K\ \overline{e_2} \rightarrow \overline{e_1 = e_2}$.
Implemented by the `inversion` tactic in Coq

Constructors are disjoint
$\text{conflict} : K_i\ \overline{e_1} = K_j\ \overline{e_2} \rightarrow \mathit{False}$ (where $K_i \neq K_j$)

Injective and Conflict Properties

Constructors are injective

$K_{inj} : K\ \overline{e_1} = K\ \overline{e_2} \rightarrow \overline{e_1 = e_2}.$

Implemented by the `inversion` tactic in Coq

Constructors are disjoint

$\text{conflict} : K_i\ \overline{e_1} = K_j\ \overline{e_2} \rightarrow \mathit{False}$ (where $K_i \neq K_j$)

Implemented by the `discriminate` tactic in Coq

Repair Function

$$\Gamma, h : K\ \overline{x} = K\ \overline{y} \vdash_s e : t \triangleq \text{let } \overline{(h : x = y)} := K_{inj}\ h \text{ in } \Gamma, \overline{(h : x = y)} \vdash_s e : t$$

$$\Gamma, h : K_1\ \overline{x} = K_2\ \overline{y} \vdash_s e : t \triangleq \text{if } K_1 \neq K_2, \text{False\_ind (conflict h)}$$

Running Example - Repair

```
λ (e : term nat).
  match e in term c return c = G_tconstr 0 nat → nat with
  | T_Int n → λ (h : G_tconstr 0 nat = G_tconstr 0 nat). n
  | T_Bool b → λ (h : G_tconstr 1 bool = G_tconstr 0 nat).
    let (h1 : 1 = 0); (h2 : bool = nat) := K_inj h
    in False_ind (conflict h1)
  | T_Pair l r p → λ (h : G_tuple l r = G_tconstr 0 nat).
    False_ind (conflict h)
  end eq_refl
```

Kinding Preservation

Theorem (Type Translation Preserves Kinding)

If $\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$ *and* $\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma$ *and* $\Sigma \vdash \Gamma \rightsquigarrow \Gamma$ *then* $[\Sigma]_{\xi_\Sigma}; [\Gamma]_\xi \vdash {}^g[t]^\Gamma_\xi : {}^g[Set]^\Gamma_\xi$

Proof.

By induction on the derivation of the type transpilation $\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$. □

Type Preservation

Theorem (Expression Translation Preserves Typing)

If $\Sigma; \Gamma \vdash e : t \rightsquigarrow e \mid \xi$ *and* $\Sigma; \Gamma \vdash t : * \rightsquigarrow_* t \mid \xi_t$ *and* $\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma$ *and* $\Sigma \vdash \Gamma \rightsquigarrow \Gamma$ *then* $[\Sigma]_{\xi_\Sigma}; [\Gamma]_\xi \vdash {}^*[e]_\xi^\Gamma : {}^*[t]_\xi^\Gamma$.
Assuming that e *doesn't have pattern matchings over datatypes that uses other GADTs as indices*

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- ✓ GADTs
- ✓ Inductive Types (with dependent types)
- ■ Compiler Correctness

Problem

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- ✓ GADTs
- ✓ Inductive Types (with dependent types)
- Compiler Correctness
  - ✓ Specification of the Syntaxes
  - ✓ Specification of the Type Systems
  - ✓ Specification of the Translation
  - ✓ Proof of Type-Preservation

Results

Table 3.1. Size of translated Operation_Repr functions

Function Name	OCaml LOC	Coq LOC
`reveal_case`	10	25
`transaction_case`	36	65
`origination_case`	30	47
`delegation_case`	11	31
`register_global_constant_case`	12	40
Total	**99**	**208**

In order to evaluate our implementation, we picked a representative GADT from the Michelson interpreter, namely manager_operation. This datatype is responsible for managing some operations performed by the nodes and smart contracts of the Tezos protocol, and its definition can be found in **operation_repr.ml**.

Implementation Caveats

We had to also implement how this translation interacts with other OCaml features, such as parametrized records and existentials

GADTs meets Records

```
type _ exp =
  | E_Int : nat -> nat exp

type 'a my_record = {
  x : 'a exp;
  y : nat
}
```

GADTs meets Records

```
Inductive exp : GSet → Set :=
| E_Int : int → exp (t_constr 1 nat).

Record my_record {a : GSet} : Set := Build {
    x : exp a;
    y : int
}.
```

Future Work

How to Correctly Translate OCaml GADTs as Coq Inductive Datatypes?

- Compiler Correctness
  - ✓ Specification of the Syntaxes
  - ✓ Specification of the Type Systems
  - ✓ Specification of the Translation
  - ✓ Proof of Type-Preservation
  - Specification of the Semantics
  - Specification of the cross-language relation
  - Proof of Semantics Preservation

Review

- We have implemented a translation of GADTs to Inductive Datatypes in Coq

Review

- We have implemented a translation of GADTs to Inductive Datatypes in Coq
- We have formalized a type system for a subset of OCaml (GADTml) and Coq (gCIC)

Review

- We have implemented a translation of GADTs to Inductive Datatypes in Coq
- We have formalized a type system for a subset of OCaml (GADTml) and Coq (gCIC)
- We proved that the translation of well typed expression in GADTml remains well typed in gCIC

Review

- We have implemented a translation of GADTs to Inductive Datatypes in Coq
- We have formalized a type system for a subset of OCaml (GADTml) and Coq (gCIC)
- We proved that the translation of well typed expression in GADTml remains well typed in gCIC
- We used our translation to remove all GADT-related axioms of a GADT datatype in the Michelson interpreter

Review

Injective TCs + EM $\to \bot$

[Agda] Agda with excluded middle is inconsistent

Thorsten Altenkirch txa at Cs.Nott.AC.UK
Thu Jan 7 11:30:41 CET 2010

- Previous message: [Agda] Agda with excluded middle is inconsistent
- Next message: [Agda] Agda with excluded middle is inconsistent
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Dear Chung,

congratulations! I didn't know about this problem and I think it is a serious issue indeed. May

Surely, type constructors should not be injective in general. A definition of the form

        data I : (Set -> Set) -> Set where

should be expandable by an annonymous declaration

        I : (Set -> Set) -> Set
        I F = data {}

in an analogous way a named function definition can be expanded by definition and a lambda abst
```

https://lists.chalmers.se/pipermail/agda/2010/001530.html

Repair Rule for Type Cast

$\Gamma, h : \tau = x \vdash_s e : t \triangleq$

take all $(\overline{z : u}) \in \Gamma$, s.t $x \in u$,

$\text{eq\_rec}\ A\ \tau\ (\lambda\ (y : A).\ (\overline{u} \to t)[x/y])$

$(\lambda\ (\overline{z_0 : u[\tau/x]}).\ \Gamma[\overline{z_0/z}] - \{x\} \vdash_s e[\overline{z_0/z}] : t[\tau/x])$

$x\ h\ \overline{z}$

Compiled Example with Typecast

```
gadt term a =
  | T_Lift : forall a. a -> term a
  | T_Int : int -> term int
  | T_Bool : bool -> term bool
  | T_Pair : forall l r.
    term l * term r -> term (l * r)

λ (e : term nat) =>
  match e with
  | T_Lift x -> x
  | T_Int n -> n
```

Example with Type Cast

```
λ (e : term nat).
  match e in term c return c = G_tconstr 0 nat → nat with
  | T_Lift a x → λ (h : a = G_tconstr 0 nat).
    eq_rec A (G_tconstr 0 nat) (λ y ⇒ decodeG y → nat)
    (λ (z : decodeG (G_tconstr 0 nat)) ⇒ z) a (eq_sym h) x
  | T_Int n → λ (h : G_tconstr 0 nat = G_tconstr 0 nat). n
  | T_Bool b → λ (h : G_tconstr 1 bool = G_tconstr 0 nat).
    let (h1 : 1 = 0); (h2 : bool = nat) := K_inj h
    in False_ind (conflict h1)
  | T_Pair l r p → λ (h : G_tuple l r = G_tconstr 0 nat).
    False_ind (conflict h)
  end eq_refl
```