A Translation of OCaml GADTs into Coq

Anonymous Author(s)

Abstract

Proof assistants based on dependent types are powerful tools for building certified software. In order to verify programs written in a different language, however, a representation of those programs in the proof assistant is required. When that language is sufficiently similar to that of the proof assistant, one solution is to use a *shallow embedding* to directly encode source programs as programs in the proof assistant. One challenge with this approach is ensuring that any semantic gaps between the two languages are accounted for. In this paper, we present *GSet*, a mixed embedding that bridges the gap between OCaml GADTs and inductive datatypes in Coq. This embedding retains the rich typing information of GADTs while also allowing pattern matching with impossible branches to be translated without additional axioms. We formalize this with GADTML, a minimal calculus that captures GADTs in OCaml, and gCIC, an impredicative variant of the Calculus of Inductive Constructions. Furthermore, we present the translation algorithm between GADTML and gCIC, together with a proof of the soundness of this translation. We have integrated this technique into **coq-of-ocaml**, a tool for automatically translating OCaml programs into Coq. Finally, we demonstrate the feasibility of our approach by using our enhanced version of **coq-of-ocaml** to translate a portion of the Tezos code base into Coq.

1 Introduction

Interactive proof assistants based on dependent type theory are powerful tools for program verification. These tools have been used to certify large and complex systems, including compilers [22], operating systems [20, 21], file systems [9], and implementations of cryptographic protocols [3]. While impressive, each of these efforts effectively constructed a new implementation of the system from scratch, as opposed to verifying an existing implementation. This points to an important hurdle to the adoption of proof assistants– in order to use a interactive theorem prover to certify programs written in different languages, users must first encode those programs in the language of the proof assistant.

A key challenge in this scenario is bridging the gap between the language of the source program and that of the proof assistant. In the case that the two are quite different, the standard solution is to employ a *deep embedding*, i.e. representing the abstract syntax trees of source programs as a data type in the proof assistant [4]. While flexible, this strategy demands considerable machinery, including a formalization of the semantics of the language inside the proof assistant, typically accompanied by an additional reasoning mechanism and proof automation, e.g. a program logic [7]. Thus, the formalization of the language semantics become part of the trusted code base (TCB) of any program verified using this approach.

When the languages are semantically similar, e.g. Haskell and Coq, an alternative strategy is to *shallowly embed* source programs in the target language. Recent efforts have shown how to automate this translation [12, 30], reducing user burden. A shallow embedding gives users access to all the built-in verification tooling of the proof assistant, and naturally inherits any further improvements made to the proof assistant. The semantics of translated programs are that of the proof assistant, and do not require extending the TCB. Instead, users rely on the translation itself to preserve the semantics of the source program. Since a key appeal of using an interactive proof assistant to verify programs are their minimal trusted code base, it is vital to ensure that the translation safely bridges any semantic gaps between the source and target languages, e.g. when translating from a partial language to a total one.

Even when the languages are quite similar, though, subtle discrepancies can exist that make a direct translation impossible. As an example, some OCaml functions over generalized algebraic datatypes (GADTs) [18] do not have a direct analogue in Gallina, the functional programming language of Coq [33], despite the fact that Coq's inductive datatypes can be thought of as a generalization of GADTs. To see why, consider the following OCaml program:

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

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

The `udu` datatype is indexed by a type that varies according to the constructor used to build a value, in this case `unit` and `unit*unit`, respectively. The utility of this extra type information can be seen in the subsequent definition of the `unit_twelve` function. Observe that `Double_unit` can never be used to build a value of type `unit udu`, and thus corresponds to an *impossible branch*, i.e. a case that is never encountered at run-time. As a convenience, OCaml allows users to elide patterns for impossible branches. While this particular example is quite simple, GADTs are commonly used to encode rich type information: e.g. embedding type information into the type of syntax trees so that only well-formed expressions can be built.

A naive transliteration of this program into Gallina immediately encounters a problem:

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

Definition unit_twelve (x : udu unit) : nat :=
  match x with
  | Unit ⇒ 12
  end.
```

Coq rejects the definition of `unit_twelve` as missing a case for `Double_unit`, as Gallina requires that `match` statements provide an exhaustive set of patterns. Adding a default pattern does not improve matters,

```
Definition unit_twelve (x : udu unit) : nat :=
  match x with
  | Unit ⇒ 12
  | _ ⇒ _
  end.
```

as Coq now complains that it cannot infer an instantiation of the body for the default case. While we could certainly provide a dummy value for this simple example, constructing a value of a given type in Coq is impossible for many polymorphic functions, e.g. the `get_head : 'a list -> 'a` function. An alternative solution is to equip Coq with the necessary typing information to *prove* that this branch is nonsensical, i.e. to derive a proof of `False` for this case. From here, we can appeal to the principle of explosion[1] to derive a dummy value. One way to do so is to use the convoy pattern [10] to augment the pattern match so that information about the type indices of the discriminee is propagated to each of the branches:

```
Program Definition unit_twelve' (x : udu unit) : nat :=
  match x in udu T return T = unit → nat with
  | Unit ⇒ fun h ⇒ 12
  | Double_unit ⇒ fun h ⇒ _
  end eq_refl.
```

This definition employs the `Program` command [29] so that we can use Coq's interactive proof mode to derive the proof of `False`. Promisingly, the resulting goal includes the assumption `H : unit ∗ unit = unit`, which encodes the desired information about the type index of `x`. Unfortunately, we are no better off than before, as it is impossible to derive `False` from this assumption without additional axioms! The most straightforward way to prove that two types are not equal is via a cardinality argument, i.e. showing that the two types have a different number of elements. This is clearly not the case here, as `unit ∗ unit` and `unit` are both singleton sets containing `(tt, tt)` and `tt` respectively. Moreover, these two types are equivalent [32]: if `unit ∗ unit <> unit` were derivable in Coq, it would imply that the univalence axiom is inconsistent with the underlying type theory, an unwelcome outcome for fans of Homotopy Type Theory [34]. In other words, what was an impossible branch in OCaml could be possible in Coq if we use this straightforward embedding!

Alternatively, we might consider tweaking `unit_twelve` to use dependent pattern matching to allow the type of `match` to vary according to the index of `x`:

```
Definition unit_twelve' (x : udu unit) : nat :=
  match x in udu T return (match T with
                            | unit ⇒ nat
                            | _ ⇒ unit
                            end) with
  | Unit ⇒ 12
  | Double_unit ⇒ tt
  end.
```

The idea here is to have impossible branches return values of a type that is easily inhabited, e.g. `unit`. Unfortunately, Coq also rejects this definition, as case analysis on types is not allowed.

Yet another solution is to implement the missing branches using an axiom of the form: `unreachable_branch: forall {A}, A`. This is the approach previously adopted by **coq-of-ocaml**, a translator from OCaml programs to Coq [12]. While this approach permits `unit_twelve` to be translated, this comes at the cost of admitting an obviously unsound axiom to the trusted code base, relying on the translation to ensure that it is used safely.

In this paper, we propose an alternative approach that does not rely on the use of unsafe axioms. Our solution implements a mixed embedding [11] of GADTs in Gallina using a distinguished universe for GADT indices, which we call `GSet`. At its core, `GSet` is a universe whose members are both injective and disjoint. This could be accomplished by adding a new sort to the Calculus of Inductive Constructions (CIC), similar to the SProp sort that has recently been added to Coq [19], but we adopt a simpler approach of making `GSet` a datatype in `Set` instead, being careful with the translation of GADTs to use `GSet`s in a way that ensures their indices are both injective and disjoint.

In summary, this paper makes the following contributions:

- We present a translation from GADTML, a formalization of OCaml with GADTs, to GCIC, a variant of CIC. Our approach translates impossible branches without any use of axioms.
- We prove that our translation is type-preserving when applied to programs that do not use user-defined type families as indices.
- We have integrated our approach into **coq-of-ocaml**[2]. We evaluate our approach by translating a portion of the Tezos code base, removing a number of axioms required by the previous implementation.

We begin by illustrating our approach with a motivating example of `GSet` in action. Sections 3 and 4 then present a

[1] The principle of explosion states that from falsehood, anything follows.

[2] The implementation is part of the current release of **coq-of-ocaml** and is available at https://github.com/formal-land/coq-of-ocaml.

formalization of our translation and its metatheory, using a minimal functional language with GADTs (GADTML) as the source language, and a variant of the Calculus of Inductive Constructors (CIC) as the target language. Section 5 discusses our implementation of this translation as part of **coq-of-ocaml**[3], and discuss its application to a real-world OCaml codebase. We then conclude with a discussion of related work and future directions.

2 An Overview of GSet

In order to properly translate OCaml clients of GADTs to Coq, we adopt a mixed embedding for the type indices of GADTs which provide similar assurances about impossible branches. The key insight is that while user-defined datatypes are not guaranteed to be disjoint in Coq, the constructors of an inductive datatype are. Thus, by adopting a deep embedding for the type indices of GADTs, we can force them to be distinct:

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

This type identifies three main kinds of OCaml types: an GADT index is either a function type, a tuple, or a labeled base type. The key intuition is that every element of this type is provably unique, modulo disjoint labels. We chose these three types for readability of the generated code and simplicity of the translation. Using `GSet`s for GADT indices, we can finally correctly translate the impossible branch of `unit_twelve`:

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

The case for `Double_unit` now assumes

```
G_tuple G_unit G_unit = G_unit
```

which contradicts the semantics of inductive datatypes in Coq. Thus, we are able to automatically discharge this branch via `discriminate` using Coq's support for tactics in terms. By carefully propagating equalities on the indices of GADTs indexed by `GSet`, we are able to similarly disregard a large class of impossible branches when using **coq-of-ocaml** to translate OCaml programs. A key intuition underlying our approach is that these equalities can be used to reify the unification algorithm used by OCaml when typing **`match`** expressions.

This is not the whole story, however, as clients of GADTs also make use of the extra typing information to enhance their own typing guarantees. The canonical example of this is having an interpreter vary its return type based on the type index of an expression encoded as a GADT:

```
type _ term =
  | T_Lift : 'a -> 'a term
  | T_Int : int -> int term
  | T_Bool : bool -> bool term
  | T_Add : int term * int term -> int term
  | T_Pair : 'a term * 'b term -> ('a * 'b) term

let rec eval : type a. a term -> a = function
  | T_Lift x -> x
  | T_Int n -> n
  | T_Bool b -> b
  | T_Add (x, y) -> (eval x) + (eval y)
  | T_Pair (t1, t2) -> (eval t1, eval t2)
```

Here, each `term` expression is augmented with its type: the integer literal `T_Int 1` has the type `int term`, for example, while the boolean `T_Bool true` has type `bool term`. In addition to prohibiting nonsensical terms such as `T_Add (T_Bool true) (T_Int 1)`, these indices allow clients of `term` to vary their signature accordingly. Thus, in addition to ensuring that `eval` is only applied to semantically meaningful expressions, it also guarantees that it returns a tuple when applied to an expression of type `(int, bool) term`, for example. In order to provide appropriate types when translating such programs, we need a denotation of a index as a type in Coq.

In order to do so, we utilize the `decodeG` function, which uses the type parameter of a `G_tconstr` to interpret an index in `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.
```

Equipped with this function, we can now produce Coq versions of both `term` and `eval` with the expected types.

```
Inductive term : GSet → Set :=
  | T_Lift : forall {a : GSet}, decodeG a → term a
  | T_Int : int → term G_nat
  | T_Bool : bool → term G_bool
  | T_Add : term G_int → term G_int → term G_int
  | T_Pair : forall {a b : GSet}, term a → term b
        → term (G_tuple a b).

Fixpoint eval {a : GSet} (function_parameter : term a)
  : decodeG a :=
  match function_parameter with
```

[3]The supplementary material includes an in-depth walkthrough of **coq-of-ocaml**.

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

Figure 1. GADTML Syntax

```
  | T_Lift v ⇒ v
  | T_Int n ⇒ n
  | T_Bool b ⇒ b
  | T_Add x y ⇒ Z.add (eval x) (eval y)
  | T_Pair t1 t2 ⇒ ((eval t1), (eval t2))
  end.
```

Note how the pattern for `T_Lift` uses `decodeG`, so that `T_Lift ()` is translated as `T_Lift (a := G_unit) ()`. Relying on `GSet` and `decodeG`, we are able to retain the ability to elide impossible branches when embedding OCaml GADTs into Coq without sacrificing the rich typing information of GADT clients, all while producing Coq programs that are syntactically similar to their OCaml counterparts.

3 GADTML and gCIC

In this section, we present GADTML, a minimal functional language with GADTs, and gCIC, our variant of CIC. The next section uses these calculi to formalize our translation. In a later section, we show how to bridge the gap between the formalism presented in this section and the implementation.

GADTML. GADTML is the source language of our compiler, and its syntax is defined in Figure 1. GADTML extends System F with tuples, user defined ADTs and GADTs, and pattern matching. We write GADTML terms in blue to easily contrast with CIC terms, which are colored in red. We use the notation $e[t]$ for type applications, uppercase lambdas are used for type abstractions, e.g. $\Lambda a.e$, and (e_1, e_2) represents a tuple. Overlines are used to represent sequences, e.g. $\overline{K}$. A GADTML program consists of a sequence of datatype declarations followed by an expression. There are two kinds of datatypes: ADTs, and GADts; ADTs only builds homogeneous datatypes, whereas GADTs allows for a finer-grained polymorphism. Although every ADT can also be written as a GADT, we keep them separate to illustrate the challenges of translating GADTs to Coq. We use G to represent GADTs and T to represent regular ADTs.

The kinding and typing rules for GADTML are presented in Figure 2. These are largely identical to their counterparts in System F, with the addition of an extra context Σ. This context is used to keep track of type constructors for each declared datatype. The type context Γ is a telescope containing type variables $a \in \Gamma$ and mappings of variables to their corresponding types $(x : t) \in \Gamma$. For simplicity, we assume that every type variable introduced in the type context has a fresh name.

$$\boxed{\Sigma; \Gamma \vdash t : *}$$

$$\frac{\text{type } T\ \overline{a}\ := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \to T\ \overline{a}} \in \Sigma \qquad \overline{\Sigma; \Gamma \vdash u : *}}{\Sigma; \Gamma \vdash T\ \overline{u} : *}\ (\text{KAdt})$$

$$\frac{\text{gadt } G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma \qquad \overline{\Sigma; \Gamma \vdash u : *}}{\Sigma; \Gamma \vdash G\ \overline{u} : *}\ (\text{KGAdt})$$

$$\boxed{\Sigma; \Gamma \vdash e : t}$$

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

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

Figure 2. Selected Kinding and Typing Rules for GADTML

The kind system includes the KAdt and KGAdt rules for type constructors, while the type system adds the typing rules TyMatch and TyGMatch for match expressions. The other rules are as expected and therefore elided, the complete definition of the kinding and typing rules can be found in the supplementary material.

The kinding rule KAdt for type constructors $T\ \overline{u}$ states that T must be declared in the context Σ and that each u_i must also be well-kinded. It is implicit in this rule that the length of $\overline{u}$ must agree with the number of declared parameters $\overline{a}$. The kinding rule for GADTs KGAdt behaves similarly, the only difference is that the return type of the constructors can differ, i.e. for an ADT the constructors must always build a $T\ \overline{a}$, whereas GADTs can build $G\ \overline{v}$, for any well typed list of terms $\overline{v}$.

The typing rule TyMatch for case analysis on ADTs requires that there must be a well-typed branch for each one

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

Figure 3. gCIC Syntax

of the declared constructors K_i in Σ of the expression being analysed. The corresponding rule for GADTs is more interesting: it only requires patterns for those constructors whose signatures are compatible with the type of the expression being analyzed. More precisely, this assumption uses the standard unification [28] algorithm to try to unify the signature of each constructor with the required type: if unification fails, the branch is impossible and can be safely elided; otherwise the resulting unifier σ is used to type the body of the pattern e_i. This rule also relies on the auxiliary function for doing type substitution in contexts, the definition is as expected and provided in the supplementary material. Notice that unification of GADTs is undecidable [17], thus we present a simple algorithm in the supplementary material.

In summary, the typing rule for pattern matching on GADTs states that a *match* expression has type t if:

- The type of the match t is well kinded;
- The discriminee e has type $T\ \overline{u}$, which must be well kinded;
- T must be declared in Σ with constructors $\overline{K}$, each of which constructs a $T\ \overline{v}$;
- Each K_i that can be unified with $T\ \overline{u}$ via a unifier σ_i must appear as a pattern. The body of the corresponding pattern e_i must have type $\sigma_i(t)$ in the context substituted with σ_i.

gCIC. The target language of our translation is gCIC, a variant of CIC equipped with impredicative Set[4] and let bindings. We focus our attention to gCIC's treatment of inductive datatypes; interested readers can see Paulin-Mohring [26] for a more detailed treatment of CIC.

Figure 3 presents the syntax of gCIC, which consists of a single construct for types and terms, and another construct for type family declarations. There is no syntactic distinction between types and expressions, as is standard in dependently typed languages. We use uppercase letters A and T to emphasize that an expression is conceptually a type, and lowercase letters e to emphasize that an object is a term. gCIC expressions include variables a, b, x, y, lambda abstractions, applications, universal quantification, the type of all types Set (including itself), type families $T\ \overline{u}$, let bindings, and case analysis.

Adopting standard practice, we use arrows for non-dependent function types. We use a, b to emphasize when a variable is treated as a type variable, and t, s, τ to emphasize when an expression is conceptually a type. gCIC includes explicit syntax for instantiating inductive datatypes in order to simplify the presentation of our translation. We elide non-dependent motive of match expressions.

Inductive type families consist of a named datatype T and its constructors $\overline{K}$. Each type declaration uses two telescopes [13], Ξ for the non-varying indices– i.e., the parameters– of a type , and Δ for the indices that do vary, i.e., the arity. By convention, we use the letters u and v for the parameters of a type $T\ \overline{u}$; v, in particular, is used for the indices in the return type of a constructor: $K_i : \Delta_i \rightarrow T\ \overline{v}$.

Figure 4 presents the typing rules for gCIC, which are largely standard [26]. The typing rule for match expressions (CTyMatch) requires a pattern for each constructor, in contrast to the TyMatch rule of GADTml, which allows impossible branches to be elided. In addition, this rule uses the supplied motive in $T\ \overline{a}$ return t to ensure that the body of each pattern e_i has the expected type of $t[\overline{u_i}/\overline{a}]$. Motives are required because unification is undecidable in the presence of inductive types [24].

$$\boxed{\Sigma; \Gamma \vdash e : t}$$

$$\frac{\text{Inductive } T\ \Xi : \Delta \rightarrow \text{Set } := \overline{\mid K : \Delta \rightarrow T\ \overline{v}} \in \Sigma}{\Sigma; \Gamma \vdash K_i : \Delta_i \rightarrow T\ \overline{v_i}}\ (\text{CTyKons})$$

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

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

Figure 4. Selected Typing Rules for gCIC

4 Translating GADTml into gCIC

As discussed in Section 2, a sound translation from GADTml to gCIC needs to deal with the semantic mismatches between how each language deals with pattern matching. In contrast to gCIC, GADTml's typing rule for match expressions permit both motives and impossible branches to be elided. This

[4] We use impredicative Set to simplify the translation by avoiding the vexing details of universes. For more details, see the supplementary material.

enables, for example, the following GADTML program to be well-typed[5]:

```
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 int) =>
  match e with
  | T_Lift x -> x
  | T_Int n -> n
```

An embedding of this program in GCIC must supply both an appropriate motive for the match expression and provide bodies for the missing impossible branches. Our solution to both issues is to modify the definition of term to use the type GSet for its indices, instead of Set. This datatype allows us to provide a dependent motive that equips each branch with exactly the typing information provided by unification in the TyGMatch rule. In the case of reachable branches, our translation uses this information to "cast" the body to the expected dependent type. For impossible branches, this information allows us to derive a proof of False; from this proof, we apply the principle of explosion to provide a "default" body for these patterns.

To accomplish this, we have implemented a translation from GADTML to GCIC consisting of three distinct phases:

1. **Transpilation**: First, we generate a potentially ill-typed GCIC program from a GADTML program, gathering information about which types need to be migrated to GSet along the way.
2. **Embedding**: Using the information from the previous phase, we update the intermediate GCIC program to use GSet indices based on the information gathered by the previous phase.
3. **Repair**: Finally, we build the proof terms needed to ensure reachable branches are well-typed and to rule out any impossible branches.

Before diving into the details of each phase, we begin by illustrating the output of each phase on the GADTML program from above.

Transpilation. The first step of our translation produces the following **ill-typed** GCIC term:

```
Inductive term : GSet → Set :=
  | T_Lift : ∀ (a : Set), a → term a
  | T_Int : int → term int
  | T_Bool : bool → term bool
  | T_Pair : ∀ (l : Set),
    forall (r : Set), term l ∗ term r → term (l ∗ r)

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

[5]For simplicity, this example assumes definitions of int and bool.

While quite similar to our input program, we can already observe several key differences. The definition of term, for example, is indexed on GSet, the main match statement now includes a motive with an equality about the type index of the discriminee, and it furthermore includes branches for each constructor of term. We note that the latter two changes rely on some auxiliary definitions. These correspond to items included in the standard library of Coq, namely eq, False, prod, nat, and bool. These definitions are as expected — e.g., eq is the type of equality proofs and has a single constructor eq_refl, while False is an uninhabited datatype — so we elide them from our example. For now, the translation uses False as a signal that later phases need to build the required proof term. We also elide the definition of GSet and its decoding function decodeG, both of which are equivalent to those presented in Section 2. Our translation depends on other functions commonly available in Coq, e.g., the recursion principles for eq (eq_rec) and False (False_rec); our example program elides the (completely standard) definitions of these functions.

The translation also generates the following set of GSet constraints; these track which type variables should live in GSet by marking them with Δ:

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

This information is used by the next phase to help embed each of these type variables into GSet in a well-typed way.

Embedding. From this intermediate program, the next phase produces the following (also ill-typed) term:

```
Inductive term : GSet → Set :=
  | T_Lift : ∀ (a : GSet), decodeG a → term a
  | T_Int : int → term (G_tconstr 0 int)
  | T_Bool : bool → term (G_tconstr 1 bool)
  | T_Pair : ∀ (l : GSet), ∀ (r : GSet),
        term l ∗ term r → term (G_tuple l r)

λ (e : term int).
  match e in term c return c = G_tconstr 0 int → int with
  | T_Lift a x → λ (h : a = G_tconstr 0 int). x
  | T_Int n → λ (h : G_tconstr 0 int = G_tconstr 0 int). n
  | T_Bool b → λ (h : G_tconstr 1 bool = G_tconstr 0 int). False
  | T_Pair l r p → λ (h : G_tuple l r = G_tconstr 0 int). False
  end eq_refl
```

Note that all the type variables tagged with Δ in ξ now have the type GSet. Any occurrence of these variables outside of an index of term has been wrapped with a call to the (elided) decodeG function, e.g. as in the first parameter of the T_Lift constructor. Finally, each constructor now produces a term with the right GSet index: T_Int now produces a value of type T (G_tconstr 0 int). The integer argument of G_tconstr uniquely identifies its corresponding type, using the position in the declaration context. In the aforementioned example, 0 marks the position of int in the context Σ while bool is is tagged with 1. After this phase, all the datatypes declarations are well typed, but it still remains to ensure that match expressions are well typed.

Repair. The last phase results in the following well-typed program[6], by either casting the body of a reachable pattern to the appropriate term, or by supplying a proof of False to provide to False_ind when the branch is impossible.

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

The body of the pattern for T_Lift now utilizes the equality provided by the translation of match to "cast" its result to the expected type (via an application of the standard recursion principle for equality eq_rec). Similarly, both T_Bool and T_Pair are impossible branches, so this phase uses the supplied equality to synthesize the required proof of False. This proof relies on two key properties of the constructors of inductive datatypes. First, that they are *injective*, which we abbreviate as $K_{inj} : h : K\ \overline{e_1} = K\ \overline{e_2} \longrightarrow \overline{e_1} = \overline{e_2}$. Second, that they are *disjoint*, which we abbreviate as conflict $: K_i\ \overline{e_1} = K_j\ \overline{e_2}$ (where $K_i \neq K_j$). Our implementation of this translation uses the tactics `inversion` and `discriminate` to construct the proofs of both K_{inj} and conflict on demand. Having seen the results of the three phases of our translation on a simple example, we now proceed to a detailed presentation of each phase.

4.1 Transpilation Phase

In order to translate a program, we must also translate its type. More precisely, to translate a type t from a source language to a type t in a target language, we want an algorithm [1, 5] $\Sigma; \Gamma \vdash t : * \rightsquigarrow t$, meaning that under the declaration context Σ, and under the variable context Γ, the type t is well-kinded in the source language and is transpiled to t in the target language.

When translating programs with GADTs, however, we also need to identify which type variables should be translated into *Set* and which ones should be translated into GSet.

[6] The type declarations are already well-typed after the previous phase, so we elide them here.

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

$$\frac{\Sigma; \Gamma \vdash a : *}{\Sigma; \Gamma \vdash a : * \rightsquigarrow_* a \mid \{a : *\}} \ (\textsc{TyTransVar})$$

$$\frac{\Sigma; \Gamma \vdash a : *}{\Sigma; \Gamma \vdash a : * \rightsquigarrow_\Delta a \mid \{a : \Delta\}} \ (\textsc{TyTransGSetVar})$$

$$\frac{\Sigma; \Gamma, a \vdash t : * \rightsquigarrow_* t \mid \xi}{\Sigma; \Gamma \vdash \forall a.t : * \rightsquigarrow_* \forall (a : \mathrm{Set}), t \mid \xi} \ (\textsc{TyTransAll})$$

$$\frac{\begin{array}{c}\text{gadt } G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \longrightarrow G\ \overline{v}} \in \Sigma \\ \Sigma; \Gamma \vdash u_i : * \rightsquigarrow_\Delta u_i \mid \xi_i, \text{ for each } u_i \in \overline{u} \\ \xi = \bigsqcup \xi_i\end{array}}{\Sigma; \Gamma \vdash G\ \overline{u} : * \rightsquigarrow_g G\ \overline{u} \mid \xi} \ (\textsc{TyTransGADT})$$

$$\frac{\begin{array}{c}\text{type } T\ \overline{a}\ := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \longrightarrow T\ \overline{a}} \in \Sigma \\ \Sigma; \Gamma \vdash u_i : * \rightsquigarrow_* u_i \mid \xi_i, \text{ for each } u_i \in \overline{u} \\ \xi = \bigsqcup \xi_i\end{array}}{\Sigma; \Gamma \vdash T\ \overline{u} : * \rightsquigarrow_g T\ \overline{u} \mid \xi} \ (\textsc{TyTransADT})$$

Figure 5. Selected Type Transpilation Rules

In order to achieve this, we track if we are currently translating an index of a GADT type constructor or not.

Formally, our translation has the form $\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$. The subscript g tracks if we are currently under a GADT type constructor or not, and the GSet constraint ξ tracks which variables should inhabit GSet and which ones should inhabit *Set*. We use the notation $\{a : *\}$, when a should inhabit *Set*, and $\{a : \Delta\}$ when a should inhabit GSet. Analogously, when translating a GADT index, we mark $g = \Delta$, otherwise $g = *$. For example, if gadt $G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \longrightarrow G\ \overline{v}} \in \Sigma$, then $\Sigma; a \vdash G\ a : * \rightsquigarrow_* G\ a \mid \{a : \Delta\}$, since a is used as an index of the GADT, G; and the algorithm tracks this via the constraint $\xi = \{a : \Delta\}$.

We define a join operation $\xi_1 \sqcup \xi_2$ such that $\langle \xi, \sqcup \rangle$ forms a join-semilattice; such that $\{a : *\} \sqcup \{a : \Delta\} = \{a : \Delta\}$, and therefore $\{a : *\} \leq \{a : \Delta\}$. For different variables it behaves as regular set union $\{a : *\} \sqcup \{b : \Delta\} = \{(a : *), (b : \Delta)\}$, This ensures that all type variables used as GSet will be appropriately marked as such.

Figure 5 lists a subset of the rules defining out the translation of types; the complete set of rules can be found in the supplementary material. The heart of the translation are the rules TyTransGADT and TyTransADT. The latter states that in order to translate a type $G\ \overline{u}$, where G is declared as a GADT, we first translate each index u_i at GSet, i.e. we set $g = \Delta$. The translation of each u_i yields a translated u_i and GSet constraint ξ_i. The final result of translating $G\ \overline{u}$ is $G\ \overline{u}$ and the join of all the sets of set of GSet constraints $\xi = \bigsqcup \xi_i$.

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

$$\frac{\Sigma;\Gamma \vdash x : t \qquad \Sigma;\Gamma \vdash t \rightsquigarrow_* t \mid \xi}{\Sigma;\Gamma \vdash x : t \rightsquigarrow x \mid \xi}\ (\textsc{TransVar}) \qquad \frac{\Sigma;\Gamma, a \vdash e : t \rightsquigarrow e \mid \xi}{\Sigma;\Gamma \vdash \Lambda a.e : \forall a.t \rightsquigarrow \lambda(a : Set).e \mid \xi}\ (\textsc{TransKLam})$$

$$\frac{\begin{array}{c}\xi = \xi_e \sqcup \xi_t \\ \Sigma;\Gamma, x : t_1 \vdash e : t \rightsquigarrow e \mid \xi_e \\ \Sigma;\Gamma \vdash t_1 : * \rightsquigarrow_* t_1 \mid \xi_t\end{array}}{\Sigma;\Gamma \vdash \lambda(x : t_1).e : t_1 \to t \rightsquigarrow \lambda(x : t_1).e \mid \xi}\ (\textsc{TransLam}) \qquad \frac{\begin{array}{c}\xi = \xi_1 \sqcup \xi_2 \\ \Sigma;\Gamma \vdash e_1 : t_2 \to t_1 \rightsquigarrow e_1 \mid \xi_1 \\ \Sigma;\Gamma \vdash e_2 : t_2 \rightsquigarrow e_2 \mid \xi_2\end{array}}{\Sigma;\Gamma \vdash e_1 e_2 : t_1 \rightsquigarrow e_1 e_2 \mid \xi}\ (\textsc{TransApp})$$

$$\frac{\begin{array}{c}\xi = \xi_1 \sqcup \xi_2 \\ \Sigma;\Gamma \vdash e : \forall a.t \rightsquigarrow e \mid \xi_1 \\ \Sigma;\Gamma \vdash s : * \rightsquigarrow_{\xi_1(a)} s \mid \xi_2\end{array}}{\Sigma;\Gamma \vdash e[s] : t[s/a] \rightsquigarrow e\ s \mid \xi}\ (\textsc{TransTApp}) \qquad \frac{\begin{array}{c}\text{type } T\ \overline{a} := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \to T\ \overline{a}} \in \Sigma \\ \Sigma;\Gamma \vdash e : T\ \overline{u} \rightsquigarrow e \mid \xi_e \\ \{\ \Sigma;\Gamma, \overline{a}, \overline{b}, \overline{x_i : t_i} \vdash e'_i : t \rightsquigarrow e'_i \mid \xi_i\ \}_{K_i} \\ \xi = (\bigsqcup \xi_i) \sqcup \xi_e\end{array}}{\begin{array}{c}\Sigma;\Gamma \vdash \text{match } e \text{ with } \overline{\mid K\ \overline{x} \to e} \text{ end} : t \rightsquigarrow \\ \text{match } e \text{ with } \overline{\mid K\ \overline{x}\ \Rightarrow e'} \text{ end} \mid \xi\end{array}}\ (\textsc{TransMatch})$$

$$\frac{\begin{array}{c}\text{gadt } G\ \overline{a} := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma \\ \Sigma;\Gamma \vdash e : G\ \overline{u} \rightsquigarrow e \mid \xi_e \qquad \Sigma;\Gamma \vdash G\ \overline{u} \rightsquigarrow_* G\ \overline{u} \mid \xi_u \\ \Sigma;\Gamma \vdash t : * \rightsquigarrow_* t \mid \xi_t \qquad \Sigma;\Gamma, \overline{a}, \overline{b} \vdash \overline{v : *} \rightsquigarrow_\Delta \overline{v} \mid \xi_v \\ \xi = (\bigsqcup \xi_i) \sqcup \xi_e \sqcup \xi_u \sqcup \xi_v \\ \left\{ \begin{array}{c|c} \begin{array}{c}\Sigma; \sigma_i(\Gamma, \overline{a}, \overline{b}, \overline{x_i : t_i}) \vdash e'_i : \sigma_i(t) \rightsquigarrow e'_i \mid \xi_i \\ \text{if } \sigma_i \equiv \text{unifies}(\overline{u}, \overline{v_i}) \not\equiv \bot\end{array} & \begin{array}{c} e'_i = \text{False} \\ \text{if unifies}(\overline{u}, \overline{v_i}) \equiv \bot\end{array}\end{array} \right\}_{K_i}\end{array}}{\Sigma;\Gamma \vdash \text{match } e \text{ with } \overline{\mid K\overline{x} \to e'} \text{ end} : t \ \rightsquigarrow \left.\begin{array}{l}\text{match } e \text{ in } G\ \overline{c} \text{ return } \overline{(c = u)} \to t \text{ with} \\ \overline{\mid K\ \overline{x}\ \Rightarrow \lambda(\overline{h : v = u}).e'} \\ \text{end eq\_refl}\end{array}\right| \ \xi}\ (\textsc{TransGMatch})$$

Figure 6. Expression Transpilation

The rule TyTransADT is similar, but instead we translate the indices at Set, i.e. $g = *$.

The remaining rules are largely as expected. If a variable is being translated at Set then the TyTransVar rule records that $\{a : *\}$. Otherwise the rule TyTransGSetVar applies, and the constrait $\{a : \Delta\}$ is recorded. Finally, type abstractions are always translated into Set, as can be seen in the TyTransAll rule. A later phase will update this to reflect the information gathered by the GSet constraint. To see why we wait to update this term until a later phase consider the following type: $\forall a.T a \to Ga$, the proper translation should be $\forall(a : GSet), T\ (\text{decode } a) \to G\ a$, however, we would have to first translate the right-hand-side of the arrow (i.e. G a) to know that $\{a : \Delta\}$ and be able to translate $T\ a$ into $T\ (\text{decode } a)$.

4.1.1 Expression Transpilation.

Figure 6 defines the type directed translation of GADTml expressions into gCIC expressions. Most of the rules follow the typing rules, with some additional tracking of set of GSet constraints. The most interesting rules are TransTApp, TransMatch, and TransGMatch,

The TransTApp rule translates type applications $e[s]$, where e has type $\forall a.t$. It first translates e and the constraint associated a will be used to determine if s should be translated at GSet or at Set.

The TransMatch rule translates match expressions of the form match e with $\overline{\mid K\ \overline{x} \to e_i}$ end, where the type of the discriminee e is $T\ \overline{u}$ and T is an ADT. It first translates the discriminee e into e, then it proceeds to translate every branch expression e_i into their respective e_i, with the proper constructor variables in the context $\overline{a}, \overline{b}, \overline{x_i}$. It also returns the join of all of the generated sets of GSet constraints ξ.

The TransGMatch rule translates matching expressions of the form match e with $\overline{\mid K\ \overline{x} \to e'}$ when the discriminee e has type $G\ \overline{u}$ and G is declared as a GADT. It translates the discriminee into e, its type $G\ \overline{u}$ into $G\ \overline{u}$, and the return type t into t.

In order to capture the information provided by unification in the TyGMatch rule, TransGMatch exposes the equalities between the indices of the discriminee and the indices of the constructors $\overline{v = u}$, in each branch of the match. This is accomplished by using the motive in $G\ \overline{c}$ return $\overline{(c = u)} \to t$, and having each branch take the equalities $\lambda(\overline{h : v = u}).e_i$ as arguments. These proofs are provided at the end of the match with the appropriate number of eq_refls, ensuring that the type of the translated match is t, as expected.

Impossible branches are elided in GADTML, but they must be provided in gCIC. To achieve this, the translation checks if each branch is possible by unifies($\overline{u}$,$\overline{v_i}$). If this fails then we set the body of the branch as False, signaling to a later phase that a proof of False is required. If it succeeds, then we translate a unified version of the body e_i. As always, this rule returns the join of the constraints generated by each subexpression.

4.2 Embedding Phase

The embedding phase uses the embedding function ${}^{g}[-]^{\Gamma}_{\xi}$ that is defined in Figure 7. This phase takes as input a gCIC term and returns another gCIC term, with type variables translated into GSet when necessary. As before, g tracks if the embedding is being done inside a GADT type constructor, Γ tracks the variable types necessary for the next phase, and ξ stores the set of GSet constraints produced by the first phase of the translation.

Similar to in the type translation, $g = \vartriangle$ denotes that the embedding is being performed inside a GADT type constructor, and $g = *$ otherwise. In other words, g flips into $\vartriangle$ when embedding the indices of a GADT type constructor, i.e. ${}^{*}[G\ \overline{u}]^{\Gamma}_{\xi} = G\ {}^{\vartriangle}[\overline{u}]^{\Gamma}_{\xi}$ and flips back to $*$ when embedding indices of ADT type constructors, i.e. ${}^{\vartriangle}[T\ \overline{u}]^{\Gamma}_{\xi} = T\ {}^{*}[\overline{u}]^{\Gamma}_{\xi}$.

Formally, embedding is a partial function defined over the structure of gCIC terms. It is only applied after the transpilation phase, and hence is only defined on the range of transpilation, which is a subset the gCIC language. As one example, $T\ (\lambda(x : t).e)$ can never be generated by the transpilation, and therefore embedding is not defined on this term.

To embed an ADT type constructor at Set, i.e. ${}^{*}[T\ \overline{u}]^{\Gamma}_{\xi}$, we simply embed its indices: $T\ {}^{*}[\overline{u}]^{\Gamma}_{\xi}$. On the other hand, to embed this type at GSet, we use G_tconstr, and record its position in the declaration signature $\#\Sigma(T)$. Embedding a GADT is similar, with the only difference that the indices will be embedded at GSet, i.e. $G\ {}^{\vartriangle}[\overline{u}]^{\Gamma}_{\xi}$. Assigning a unique key to each type constructor is paramount for ensuring injectivity and disjointness of type constructors, which is crucial to the next phase.

To embed a match expression, we embed both the discriminee and return of the motive. To finish translating the branches of the match, the next phase will use information from the typing context Γ to repair the body of each match to have the correct type.

The other rules are largely as expected. Arrows and tuples are also embedded into GSet when necessary. The indices of equations are always translated with $g = \vartriangle$ because they are only generated by the transpiler to compare GADT indices. Universally quantified variables are now migrated to GSet when they are marked in the set of GSet constraints. The context Γ is also extended when embedding lambda terms and universal quantifiers, as this information will be necessary by the repair phase.

4.3 Repair Phase

The last translation step repairs the body of match expressions so that they are well-typed. It does so via the repair function $\vdash_s$ defined in Figure 8. This function takes as input a term e, a target type t and a context Γ, and outputs a term $e^{\dagger}$ that has type t under the context Γ (Lemma 1). It recurses over the tail of the typing context Γ, and terminates when it either reaches a non-equation in Γ, or when a contradiction found.

The first two rules of $\vdash_s$ perform a type cast when they find an equation on a variable x. The rules behave similarly, the only difference being that if x is found on the left of the equation the symmetry property of equations is first applied via the function eq_sym. To perform this cast, the algorithm first gathers all variables $\overline{z : u}$ in Γ in which x appears in u_i. It then casts the body by recursively applying $\vdash_s$ to e with all occurrences of x substituted by τ, including occurrences in the target type t and in the context Γ. The cast is built using the function eq_rec, which has the type:

$$\begin{aligned}\Sigma; \vdash \text{eq\_rec} : \forall (A : Set)(x : A)(P : A \longrightarrow Set),\\ Px \longrightarrow \forall (y : A), x = y \longrightarrow Py\end{aligned}$$

Building this substitution recursively is possible because the previous type marks the exact positions at which the rewrite will happen. This can be seen in the third argument supplied to eq_rec: $\lambda\ (y : A).\ (\overline{u} \rightarrow\ t)[x/y]$, which indicates that the expression being cast has type $\overline{u}[x/y] \rightarrow t[x/y]$. This allows the recursive call to access to each $z_0 : \overline{u}[x/\tau]$, after y is instantiated to τ. Substitution then replaces each z in Γ with its respective z_0, i.e. $\Gamma[\overline{z_0/z}]$. Since $\overline{z}$ captures all variables that mentions x, and they have been substituted by z_0, which doesn't mention x, we can safely remove it from the context, via $\Gamma[\overline{z_0/z}] - \{x\}$.

When an equation over constructors is encountered, the function first check if they are the same constructor. If this is the case, the repair algorithm uses K_{inj}, the injectivity rule for constructors, and continues the type substitution recursively. If the constructors are not the same, it has reached a contradiction, and the term can be replaced by the aforementioned conflict term. The repair function also introduces function variables $\lambda(x : t)$ into the context, and furthermore ignores trivial equations.

Concretely, this algorithm behaves as an inverse function of the substitution of type variables. We can see this in Lemma 1, which states that for any term e that has type $t[\tau/x]$ under a context with the same substitution $\Gamma[\tau/x]$, applying the repair algorithm using the equality $h : \tau = x$ to the e yields a well-typed term without the substitution.

Lemma 1 (Repair step is the inverse of substitution)**.** *If* $\Sigma; \Gamma[\tau/x] \vdash e : t[\tau/x]$ *and* $\Gamma, h : x = \tau \vdash_s e : t = e^{\dagger}$ *then*

$$
\begin{aligned}
&{}^{*}[Set]^{\Gamma}_{\xi} = Set \\
&{}^{\Delta}[Set]^{\Gamma}_{\xi} = GSet \\
&{}^{*}[a]^{\Gamma}_{\xi} = \begin{cases} \text{decode } a & \text{if } (a : \Delta) \in \xi \\ a & \text{otherwise} \end{cases} \\
&{}^{\Delta}[a]^{\Gamma}_{\xi} = a, \quad \text{if } (a : \Delta) \in \xi \\
&{}^{\Delta}[t_1 \rightarrow t_2]^{\Gamma}_{\xi} = \text{G\_arrow } {}^{\Delta}[t_1]^{\Gamma}_{\xi}{}^{\Delta}[t_2]^{\Gamma}_{\xi} \\
&{}^{*}[t_1 \rightarrow t_2]^{\Gamma}_{\xi} = {}^{*}[t_1]^{\Gamma}_{\xi} \rightarrow {}^{*}[t_2]^{\Gamma}_{\xi} \\
&{}^{*}[t_1 * t_2]^{\Gamma}_{\xi} = {}^{*}[t_1]^{\Gamma}_{\xi} * {}^{*}[t_2]^{\Gamma}_{\xi} \\
&{}^{\Delta}[t_1 * t_2]^{\Gamma}_{\xi} = \text{G\_tuple } {}^{\Delta}[t_1]^{\Gamma}_{\xi}{}^{\Delta}[t_2]^{\Gamma}_{\xi}
\end{aligned}
$$

$$
\begin{aligned}
&{}^{*}[T\ \overline{u}]^{\Gamma}_{\xi} = T\ {}^{*}[\overline{u}]^{\Gamma}_{\xi} \\
&{}^{\Delta}[T\ \overline{u}]^{\Gamma}_{\xi} = \text{G\_tconstr } (\#\Sigma(T))\ (T\ {}^{*}[\overline{u}]^{\Gamma}_{\xi}) \\
&{}^{*}[G\ \overline{u}]^{\Gamma}_{\xi} = G\ {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi} \\
&{}^{\Delta}[G\ \overline{u}]^{\Gamma}_{\xi} = \text{G\_tconstr } (\#\Sigma(G))\ (G\ {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi}]) \\
&{}^{*}[\overline{u} = \overline{v}]^{\Gamma}_{\xi} = {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi} = {}^{\Delta}[\overline{v}]^{\Gamma}_{\xi} \\
&{}^{*}[\forall(a : Set), t]^{\Gamma}_{\xi} = \forall(a : GSet), {}^{*}[t]^{\Gamma,(a\,:\,GSet)}_{\xi} \quad \text{, if } (a : \Delta) \in \xi \\
&{}^{*}[\forall(a : Set), t]^{\Gamma}_{\xi} = \forall(a : Set), {}^{*}[t]^{\Gamma,(a\,:\,Set)}_{\xi} \quad \text{, otherwise} \\
&{}^{*}[\lambda(x : t_1), t]^{\Gamma}_{\xi} = \lambda(x : {}^{*}[t_1]^{\Gamma}_{\xi}), {}^{*}[t]^{\Gamma,(x\,:\,{}^{*}[t_1]^{\Gamma}_{\xi})}_{\xi} \\
&{}^{g}[e_1\ e_2]^{\Gamma}_{\xi} = {}^{g}[e_1]^{\Gamma}_{\xi}\ {}^{g}[e_2]^{\Gamma}_{\xi}
\end{aligned}
$$

$$
{}^{*}\left[\begin{array}{l} \text{match } e \text{ in } T\ \overline{b} \text{ return } t \text{ with} \\ \overline{\mid K\ \overline{x} \Rightarrow e' \text{ end}} \end{array} \right]^{\Gamma}_{\xi} = \frac{\text{match } {}^{*}[e]^{\Gamma}_{\xi} \text{ in } T\ \overline{b} \text{ return } {}^{*}[t]^{\Gamma}_{\xi \sqcup \{\overline{b\,:\,*}\}} \text{ with}}{\mid K\ \overline{x} \Rightarrow \Gamma, (\overline{x} : {}^{*}[\Delta]^{\Gamma}_{\xi}) \vdash_s {}^{*}[e']^{\Gamma,(\overline{x}\,:\,{}^{*}[\Delta]^{\Gamma}_{\xi})}_{\xi} : {}^{*}[t]^{\Gamma}_{\xi} \text{ end}}
$$

Figure 7. Embedding Function

$\Gamma, h : x = \tau \vdash_s e : t \triangleq$
  take all $(\overline{z : u}) \in \Gamma$, s.t $x \in u$,
  $\text{eq\_rec } A\ \tau\ (\lambda\ (y : A).\ (\overline{u} \rightarrow t)[x/y])$
  $(\lambda\ (\overline{z_0 : u[\tau/x]}).\ \Gamma[\overline{z_0/z}] - \{x\} \vdash_s e[\overline{z_0/z}] : t[\tau/x])$
  $x\ (\text{eq\_sym } h)\ \overline{z}$

$\Gamma, h : x = \tau \vdash_s e : t \triangleq$
  take all $(\overline{z : u}) \in \Gamma$, s.t $x \in u$,
  $\text{eq\_rec } A\ \tau\ (\lambda\ (y : A).\ (\overline{u} \rightarrow t)[x/y])$
  $(\lambda\ (\overline{z_0 : u[\tau/x]}).\ \Gamma[\overline{z_0/z}] - \{x\} \vdash_s e[\overline{z_0/z}] : t[\tau/x])$
  $x\ h\ \overline{z}$

$$
\begin{aligned}
\Gamma, h : K\ \overline{x} = K\ \overline{y} \vdash_s e : t \quad &\triangleq \quad \text{let } (\overline{h : x = y}) := K_{inj}\ h \text{ in} \\
&\qquad \Gamma, (\overline{h : x = y}) \vdash_s e : t \\
\Gamma, h : K_1\ \overline{x} = K_2\ \overline{y} \vdash_s e : t \quad &\triangleq \quad \text{if } K_1 \neq K_2, \\
&\qquad \text{False\_ind (conflict h)} \\
\Gamma \vdash_s \lambda(x : t').\ e : t' \rightarrow t \quad &\triangleq \quad \Gamma, (x : t') \vdash_s e : t \\
\Gamma, h : \tau = \tau \vdash_s e : t \quad &\triangleq \quad \Gamma \vdash_s e : t \\
\Gamma \vdash_s e : t \quad &\triangleq \quad e \quad \text{, if the head of } \Gamma \text{ is not an equation}
\end{aligned}
$$

Figure 8. Repair Function

$\Sigma; \Gamma, h : x = \tau \vdash e^{\dagger} : t$, *assuming that* Γ *contains no equations.*

Proof. Direct from the definition of $\vdash_s$ and the type of eq_rec. See the supplementary material for the full proof. □

4.4 Soundness of the Translation

In order to show that our translation is sound, we prove both that type translation preserves kinding and that expression translation preserves typing.

Theorem 1 establishes that if a type t is a well-kinded type in GADTml, then its translation t is also well-kinded under a translated context, after the embedding and repair phases. For this it is also necessary to define the translation of contexts: $\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma$ and $\Sigma \vdash \Gamma \rightsquigarrow \Gamma$, these definitions are straightforward and elided from our presentation due to space constraints, but they can be found in the supplementary material.

Also note the contexts can also be embedded using the generated set of GSet constraints, i.e. $[\Sigma]_{\xi_\Sigma}; [\Gamma]_{\xi}$. The definition of these embeddings are a straightforward application of the embedding algorithm to contexts. The translation of declaration contexts $\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma$ generates its own set of GSet constraints since the variable information on each datatype constructor is local.

Theorem 1 (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 $\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$. □

Our second theorem establishes that the translation of a well-typed GADTml term produces well-typed gCIC term.

We note here that the repair function isn't strong enough to handle nested user-defined type constructors of the form $G\ (T\ \overline{u})$, as the current formulation of GSet can only embed one level of type constructors with a unique key, as seen in G_tconstr.

Theorem 2 (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 $[\Sigma]_{\xi_\Sigma}; [\Gamma]_\xi \vdash {}^*[e]^\Gamma_\xi : {}^*[t]^\Gamma_\xi$, assuming that e doesn't have pattern matchings over datatypes that uses user-defined types as indices*

Proof. By induction over the derivation of the transpilation of expressions $\Sigma;\Gamma \vdash e : t \rightsquigarrow e \mid \xi$. The more interesting case is that of the TransGMatch rule; the proof of this case is included in the supplementary material. □

5 Implementation and Evaluation

We have implemented our translation of GADTs in **coq-of-ocaml**, a source-to-source compiler from OCaml to Coq. Our implementation closely follows the algorithm presented above, although there are two discrepancies worth mentioning. First, our translation supports mixing datatype and function declarations, in contrast to the algorithm, which requires type declarations to appear at the beginning of a program. In order to ensure that embedded types are unique, our implementation uses strings instead of numbers for identifiers, and uses the name of a type for this argument. Second, as described in Section 1, the repair phase of the algorithm inserts uses of the `discriminate` and `subst` tactics for the bodies of branches.

Notably, **coq-of-ocaml** handles a considerably larger subset of OCaml than GADTml, including many features that use type parameters, e.g. parametrized records, parametrized type synonyms and "grabbing" of existential variables. All of these represent another use of type variables that our implementation also carefully tracks and migrates to GSet when necessary. In addition, our implementation handles native types (e.g. `int`, `bool`, and `list`) and translates them as their equivalent counterpart in Coq's standard library. As the treatment of these base types is orthogonal to the translation of GADTs, we have opted to elide them from our formalization.

We have developed a set of micro-benchmarks showcasing each of the features needed to support GSet-indexed GADTs in **coq-of-ocaml**. They can be found in the folder *tests* of the supplementary material[7], they are:

- *GSet_term.ml*: impossible branches and casts;
- *GSet_record.ml*: embedded records with parameters that are used as GADT indices;
- *GSet_existential.ml*: existential variables used as GADT indices;

[7]The micro-benchmarks are also included in the current version of **coq-of-ocaml**, but organized differently.

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**

Table 1. Size of translated Operation_Repr functions

- *GSet_record.ml*: regular records and irrefutable patterns;
- *GSet_ex_grab.ml*: grabbing of existential variables that are marked as GSet;

To evaluate the effectiveness of our approach, we have also used our extended version of **coq-of-ocaml** to translate a portion the Michelson interpreter, which is part of Tezos' code base. Michelson is a smart contract language that uses GADTs to ensure that operations are always applied to arguments of the expected type. Since Michelson can be used to manage real money, it is paramount that its interpreter is bug-free and reliable.

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*

Before the implementation of the presented translation, **coq-of-ocaml** would translate impossible branches via a use of the axiom `gadt_unreachable_branch`. Using our translation, the updated version of **coq-of-ocaml** eliminated all uses of the `gadt_unreachable_branch` axiom in the five functions shown in Table 1. While the updated translation increased the size of the translated functions, e.g. by inserting type equalities in `match` statements, the small increase in code size has a clear benefit in terms of reducing the trusted code base by eliminating the use of axioms.

6 Related Work

The source language of our translation, GADTml, is similar to $\lambda_{2,G\mu}$, first presented in Xi et al. [35]. That calculus is an extension of System F with all the features of GADTml and more (e.g. fixpoints and let bindings). Although $\lambda_{2,G\mu}$ does not include GADTs directly, the authors show they can be derived from a surface language. While GADTml explicitly uses unification to type check pattern matching, $\lambda_{2,G\mu}$ instead solves constraints maintained in the type variable context. The authors also never discuss impossible branches. We argue that the alternative design choices of GADTml enable a cleaner presentation of our translation.

In a similar vein, Sulzmann et al. [31] presents System Fc, an extension of System F with type equality coercions. The authors show that their calculus can encode a plethora of interesting language features, including GADTs. That work also presents a multi-step constraint-translation of a source language with GADTs into System Fc, and also shows that their translation is type-preserving. In addition to the different target languages, the key difference between our two approaches is that Sulzmann et al. [31] does not need to construct proof terms witnessing type casts and the infeasibility of impossible branches.

There have been a number of efforts using interactive proof assistants to verify programs written in mainstream functional programming languages. The most closely related example is **hs-to-coq** [30], a source-to-source translator from Haskell to Coq. Like **coq-of-ocaml**, **hs-to-coq** produces a shallow embedding of source programs in Coq. The tool is able to capture many advanced language features of Haskell, including typeclasses, records and guarded pattern matching, and it has been used to translate and verify several textbook examples as well as significant portions of Haskell's containers library [6]. **hs-to-coq** provides a best-effort approach to supporting GADTs in translated programs. For some datatypes, users can provide a specification file marking which arguments should be translated as indices, simplifying type argument inference. They then translate the indices directly, so they are forced to use an axiom to handle impossible branches, similar to the use of `unreachable_gadt_branch` in **coq-of-ocaml**. This axiom is also used to support incomplete patterns in Haskell functions. In principle, our translation algorithm is general enough to be incorporated into **hs-to-coq**, eliminating the need for this axiom when used for impossible branches.

CFML translates OCaml programs to Coq via characteristic formulae [8]. The key idea in this approach is to capture program behaviors via invariants expressed as higher-order formulae, which can then be expressed directly in the logic of Coq. Since this approach does not generate functions in Coq, it is capable of faithfully capturing the behaviors of non-terminating programs. The cost of this flexibility is that the translation loses much of the structure of the original program. Following the structure of the source program is an important design decision behind **coq-of-ocaml**.

OCaml-to-PVS Equivalence Validation (**OPEV**) [2] is a tool to validate translations between OCaml and PVS [25] programs by automatically generating a large number of test cases and automatically discharging them using PVS. This approach could help to address a different gap in the current implementation of **coq-of-ocaml**, as it currently relies on users to validate that the translated code matches the intended semantics of the OCaml source programs.

Cameleer [15, 27] takes as input OCaml programs annotated with specifications in the GOSPEL language and outputs verification conditions. These conditions are then discharged by Why3 [16], a deductive verification toolchain that interfaces with several SMT solvers. In contrast to **coq-of-ocaml**, **Cameleer** relies on automated theorem provers to certify the correctness of OCaml programs.

Coq provides a mechanism to extract code [23] to OCaml, Haskell, Scheme and JSON. This is, in some sense, the inverse of the problem we address here, as we go from from a language with less expressive types to one with richer types. Thus, the extraction mechanism is tasked with safely *erasing* information, including any proof terms, as opposed to faithfully *preserving* type information. Spector-Zabusky et al. [30] proposes that extracting translated code and then testing its equivalence with the original program could greatly increase confidence in their results. Automatically validating the equivalence of roundtrip translations of translating code is an interesting direction for future work.

7 Future Work and Conclusion

One potential direction for future work is to provide a proof that the runtime behavior of the translated GADTml term is equivalent to the behavior of the generated gCIC term. In addition, the present definition of GSet is not expressive enough to translate some mutually recursive GADTs, due to positivity constraints in Coq. The current formulation of GSet is also currently not expressive enough to solve equations generated by GADT indexed by other user-defined type constructors, since these type constructors are injective in OCaml but not necessarily in CIC. As mentioned in the introduction, this could be more directly addressed by making GSet a new universe, similar to SProp, with is equipped with axioms encoding that all inhabitants of this new universe respects injectivity and disjointness of type constructors. A key technical challenge is developing restrictions that make this new universe sound, as injectivity of type constructors is known to be unsound in the presence of inductive types [14].

In this paper, we have presented *GSet*, a mixed embedding that bridges the gap between OCaml GADTs and inductive datatypes in Coq. This embedding retains the rich typing information of GADTs while also allowing case statements with impossible branches to be translated without additional axioms. We presented GADTml, a calculus that captures the essence of GADTs in OCaml and described a sound translation from GADTml to gCIC, a variant of CIC. We have implemented this technique in **coq-of-ocaml**, a tool for automatically translating OCaml programs into Coq. We have used this enhanced version of **coq-of-ocaml** to translate a portion of the OCaml interpreter for Michelson, the smart contract language of Tezos, into Coq, removing five axioms that were generated by previous versions of the tool.

References

[1] Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. *SIGPLAN Not.* 43, 9 (sep 2008), 157–168. https://doi.org/10.1145/1411203.1411227

[2] Xiaoxin An, Amer Tahat, and Binoy Ravindran. 2020. A Validation Methodology for OCaml-to-PVS Translation. In *NASA Formal Methods*, Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou (Eds.). Vol. 12229. Springer International Publishing, -, 207–221. https://doi.org/10.1007/978-3-030-55754-6_12 Series Title: Lecture Notes in Computer Science.

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. 2016. Implementing and Proving the TLS 1.3 Record Layer. Cryptology ePrint Archive, Report 2016/1178. https://ia.cr/2016/1178.

[4] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*. North-Holland Publishing Co., NLD, 129–156.

[5] William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-Preserving CPS Translation of Σ and Π Types is Not Not Possible. *Proc. ACM Program. Lang.* 2, POPL, Article 22 (dec 2017), 33 pages. https://doi.org/10.1145/3158110

[6] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-Coq to Real-World Haskell Code (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 89 (July 2018), 16 pages. https://doi.org/10.1145/3236784

[7] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, J. Dodds, and A. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61 (2018), 367–422.

[8] Arthur Charguéraud. 2010. Program Verification through Characteristic Formulae. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 321–332. https://doi.org/10.1145/1863543.1863590

[9] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. *Using Crash Hoare Logic for Certifying the FSCQ File System*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

[10] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, -.

[11] Adam Chlipala. 2021. Skipping the Binder Bureaucracy with Mixed Embeddings in a Semantics Course (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 94 (Aug. 2021), 28 pages. https://doi.org/10.1145/3473599

[12] Guillaume Claret. 2021. Coq of Ocaml. https://github.com/clarus/coq-of-ocaml. Accessed: 2021-09-09.

[13] Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Unifiers as Equivalences: Proof-Relevant Unification of Dependently Typed Data. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 270–283. https://doi.org/10.1145/2951913.2951917

[14] Leonardo de Moura. 2022. Coq of Ocaml. https://github.com/leanprover/lean/issues/654. Accessed: 2022-09-21.

[15] Jean-Christophe Filliâtre, Léon Gondelman, Cláudio Lourenço, Andrei Paskevich, Mário Pereira, Simão Melo de Sousa, and Aymeric Walch. 2020. A Toolchain to Produce Verified OCaml Libraries. (Jan. 2020). https://hal.archives-ouvertes.fr/hal-01783851 working paper or preprint.

[16] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128.

[17] Jacques Garrigue and Jacques Le Normand. 2017. GADTs and Exhaustiveness: Looking for the Impossible. *Electronic Proceedings in Theoretical Computer Science* 241 (Feb. 2017), 23–35. https://doi.org/10.4204/EPTCS.241.2

[18] Jacques Garrigue and Jacques Le Normand. 2011. Adding GADTs to OCaml the direct approach. (2011), 29.

[19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL, Article 3 (jan 2019), 28 pages. https://doi.org/10.1145/3290316

[20] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. *SIGPLAN Not.* 50, 1 (Jan. 2015), 595–608. https://doi.org/10.1145/2775051.2676975

[21] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[22] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[23] Pierre Letouzey. 2008. Extraction in coq: An overview. In *Conference on Computability in Europe*. Springer, 359–369.

[24] Conor McBride. 2000. Dependently typed functional programs and their proofs. (2000).

[25] Sam Owre, John M Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *International Conference on Automated Deduction*. Springer, 748–752.

[26] Christine Paulin-Mohring. 2015. Introduction to the calculus of inductive constructions.

[27] Mário Pereira and António Ravara. 2021. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 677–689.

[28] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

[29] Matthieu Sozeau. 2007. Subset Coercions in Coq. In *Types for Proofs and Programs*, Thorsten Altenkirch and Conor McBride (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–252.

[30] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3167092

[31] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

[32] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.* 2, ICFP, Article 92 (July 2018), 29 pages. https://doi.org/10.1145/3236787

[33] The Coq Development Team. 2021. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.4501022

[34] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.

org/book, Institute for Advanced Study.

[35] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150