A TRANSLATION OF OCAML GADTS INTO COQ

by

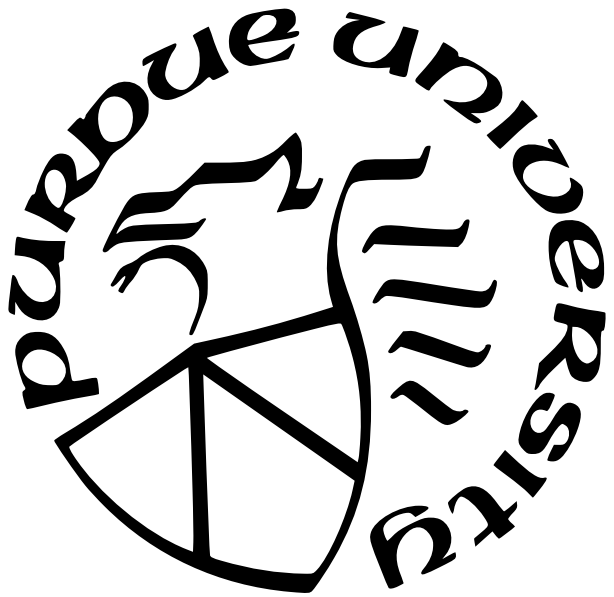
Pedro da Costa Abreu Junior

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science



Department of Computer Science

West Lafayette, Indiana

May 2024

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Benjamin Delaware, Chair

Department of Computer Science

Dr. Tiark Rompf

Department of Computer Science

Dr. Suresh Jagannathan

Department of Computer Science

Approved by:

Dr. Kihong Park

To my mother, the toughest of the roses.

Provider of a constant and firm base of support in my life.

ACKNOWLEDGMENTS

First and foremost I thank and dedicate this thesis to my mother, who has provided me with the constant support of my choices and work, even though my choices often contradicts the yearnings that only the delicate heart of a mother could understand. I thank my therapist Maria Clara Pimenta who has saved me from drowning in the deep sea of anxiety and burnout that constantly lurks in academic life. I deeply thank my advisor, who is the most emotionally resilient person I've ever met. Thank you Ben for all your support, understanding, teachings, meetings, conversations, fundings and support letters. Thank you Anxhelo Xhebraj and Beatrice Belivacqua for all the tea, you've always had my back when I needed most. Thank you Paulo Sérgio and Lucas Nogueira for our weekly esoteric study group (which is often just a weekly manly therapy session). Thank you all friends and colleagues for the insightful conversation, fun and not so fun times. Thank you Murilo, Pratyush, Patrick, Thu, and all of those that came and went at some point of this journey. Thank you my landlords Pat and Cordy Embry for the cheap rent and the care on the 5 out of 6 years I've lived in Lafayette. I thank my girlfriend Rachel Ehrlich for providing me the opportunity to finish my studies in the US in a high note when I was convinced that only the taste of acidity remained at the corner of my mouth.

Thank you Nomadic Labs for partially funding this work. Thank you Guillaume Claret, the work of this thesis would not have been possible without the foundational work of **coq-of-ocaml**.

Thank you Seu Tranca Ruas, Dona Tata, Maria Padilha das Almas, and Capa Preta for your brief and yet surgical teachings, 5 minutes in your presence is worth a whole year of teachings through the sinuous natural means of life. Laroyê!

Thank you Pai Joaquim and Vovó Francisca D'angola for providing me with the resources for completing the last leg of my academic journey in the US.

Finally, thank you all of the people, intelligences, organizations and forces that cannot be named due to reasons of force majeure. Your hidden yet perenne work to bring light unto the world fills my heart with hope for mankind. 770!

PREFACE

When I first came to Purdue I had the goal to be the first in my whole extended family to have a PhD. But life sometimes has its own inexorable plan.

Although I began my PhD at Purdue in 2018, I started working in the topic of this thesis in 2020. I was supposed to fly to Paris for that summer and work on **coq-of-ocaml**. Unironically, I thought 2020 was going to be the best year of my life, but instead, I ended up locked up in my one bedroom apartment working remotely for long hours to the point of burnout.

However, I can't complain much. In many ways, I'm still a yokel kid from small town in Brazil with a population of measly 20,000 people (Silvânia, Goiás), and from a family of farm workers and illiterate grandparents. And now I'm suddenly pursuing the highest of degrees, traveling the world, and making a complete fool out of myself at the dinner table for having no idea on how to use so many forks! This is all beyond my wildest dreams.

Even with all the difficulties, this journey was still largely positive and full of learning. I can't say it was easy. It was very hard. I went through burnout, grief, diseases, and an uncountable amount of failures. This trial by fire brought me incredible growth and personal development. I agree with the buddhist Monk Thich Nhat Hanh when he calls suffering a "Holy Path". Suffering really have the power to show us where we're clinging and getting things wrong in a deeply personal perspective. Finding my clingings allowed me to decide to embrace the (perceived) failure of dropping the PhD program after my second CPP rejection, and write this Master Thesis instead. Honestly, I could have just dropped the PhD without writing this thesis, but I genuinely believe that this work may be of interest of someone else in the future. Or at least I hope so.

This thesis is about the translation between two programming languages. The source language being a multi-purpose functional programming language, whereas the target is an interactive theorem prover. In particular, the translation of Generalized Algebraic Datatypes. When I sat down to write the specification of GADTs and their translation, I noticed that there is a great gap in the literature on the topic. There are papers and abstracts on adding GADTs to a language [1, 2], compiling GADTs [3], impossible branches [4], and alike. And on

the other end of the spectrum there are beautiful papers and thesis [5] on inductive families and dependent pattern matchings [6]. However, there wasn't a solid metatheorical foundation on GADTs and their real relationship with respect to the full power of dependently typed inductive definitions. As such, I had to write much of the theory for GADT from scratch for both of the failed submissions of this work to CPP'21 and CPP'22. Only in the year that I'm writing this thesis (2024) that a POPL paper was accepted on the beautiful and foundational metatheory of GADTs [7].

Finally, this Thesis brings together the research of GADTs, inductive families into the world of compiler correctness, which is also a vibrantly active field [8].

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 thesis, 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 [9], operating systems [10, 11], file systems [12], and implementations of cryptographic protocols [13]. 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 an 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 [14]. 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 [15]. 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 [16, 17], 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.

1.1 Challenges Translating GADTs to Coq

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) [1] do not have a direct analogue in Gallina, the functional programming language of Coq [18], 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 [19] to augment the pattern match so that information about the type indices of the discriminee is propagated to each of the branches:

```
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 ⇒ _
  end eq_refl).
```

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 [20]: 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 [21]. In

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

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 [16]. 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 thesis, we present an alternative approach that does not rely on the use of unsafe axioms. Our solution implements a mixed embedding [22] 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 [23], 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.

1.2 Contributions and Thesis Organization

In summary, this thesis 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.

This thesis is organized as follows: Chapter 2 we give an overview to the assumed knowledge on Algebraic Datatypes (Section 2.1), Generalized Algebraic Datatypes (Section 2.2), Impossible Branches (Section 2.2.1), Coq (Section 2.3), Inductive Types, and Dependent Pattern Matching (Section 2.3.1).

We begin Chapter 3 by illustrating our approach with a motivating example of `GSet` in action. Section 3.3, then present a 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. Chapter 4 discusses our implementation of this translation as part of **coq-of-ocaml**, and discuss its application to a real-world OCaml codebase. We then conclude at Chapter 6 with a discussion of related work and future directions.

[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.

2. BACKGROUND

2.1 Algebraic Data Types

Algebraic Data Types (ADTs), initially introduced in the HOPE [24] programming language, play a crucial role in functional programming. They are now recognized as one of the cornerstones of this programming paradigm.

ADTs were devised to enable a variety of functionalities, including exhaustive case analysis, an organized syntax for sequencing objects, modularity, and data abstraction. The term "algebraic" refers to the creation of a type along with a set of constructors–rules that define how to operate on this type.

Consider this OCaml example illustrating an ADT:

```
type nat =
| Zero
| Succ of nat
```

In this, `Zero` and `Succ` are constructors for the `nat` type. `Zero` requires no arguments, while `Succ` takes a single `nat` type argument.

With ADTs, it's not only vital to have a mechanism to construct data types, but also a way to deconstruct them. This deconstruction process, known as pattern matching, allows us to discern how an element of a particular type was constructed. In OCaml, pattern matching is performed using the `match e with` construct, where `e` represents the expression being scrutinized.

An example is the `plus_one` function defined below:

```
let rec plus_one (x y: nat) : nat =
    match x with
    | Zero -> y
    | Succ x' -> Succ (plus_one x' y)
    end
```

Here, pattern matching on the variable x determines the function's behavior based on how x was constructed. In this case, x is called the *match discriminee*. Pattern matching can be thought of as a way to conduct case analysis within the language.

Pattern matching is emerging as an essential feature in modern programming languages due to its powerful ability to simplify complex conditional logic. It allows programmers to directly decompose data structures and match their contents against specific patterns, leading to more readable and concise code. This approach is particularly beneficial when dealing with intricate data types, with hierarchical structures. Pattern matching enhances code clarity by avoiding the clutter of nested if-else statements, making the programmer's intent more transparent. Moreover, many languages with pattern matching enforce exhaustiveness checks, ensuring that all possible cases are handled, which significantly reduces the likelihood of runtime errors and unhandled scenarios. As a result, many programming languages, even those that are not traditionally thought of as functional languages, are integrating pattern matching features. This shift reflects a broader recognition of the utility of functional programming concepts in enhancing code safety, maintainability, and developer productivity.

2.1.1 Parametrized ADTs

Algebraic Data Types (ADTs) can also be parametrized, further extending their versatility. Consider the definition of polymorphic lists in OCaml:

```
type 'a list =
| nil
| cons of ('a * 'a list)
```

Parametrized ADTs, such as `list` in this example, introduce the concept of a *type family*. This means that `list` does not refer to a single type, but rather to an infinite family of types, each variant depending on the parameter provided. In this context, `list` acts as a *type constructor*, requiring an argument to specify the exact type of its elements.

This particular ADT has two constructors: `nil` and `cons`. The `nil` constructor represents an empty list and takes no arguments. The `cons` constructor, on the other hand, is used to build a list by taking two arguments: the head (of type `'a`) and the tail (of type `'a list`).

One key aspect of this definition is that it defines homogeneous lists. This means that all elements within a particular list instance must be of the same type, as defined by the parameter `'a`.

2.2 Generalized Algebraic Datatypes

The introduction of Generalized Algebraic Data Types (GADTs) in Haskell and in OCaml, marked a significant evolution in the concept of ADTs. First studied by Xi et al [25], GADTs extends the functionality of ADTs by enabling constructors to create elements of varied types. For instance, the following OCaml GADT defines the AST of a simple expression language:

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

In this example, the `T_Int` constructor generates terms of the type `int term`, while `T_Bool` yields terms of the type `bool term`. This ability for terms to be constructed with different types classifies them as *heterogeneous datatypes*.

GADTs can be perceived as weaker form of *inductive families* found in dependent type languages, such as Coq. The precise relationship between GADTs and inductive families is still under active exploration, and this thesis aims to help clarify that relationship. Concomitantly to the work of this thesis, the paper "The Essence of Generalized Data Types" [7] introduces a core calculus for studying GADTs, and proves important open problems about them, such as soundness of the core calculus and the non macro-expressibility of GADTs in terms of ADTs, effectively showing that GADTs indeed adds expressive power to the programming language.

The additional expressiveness provided by GADTs strengthens the type system, enabling more precise type definitions. This in turn equips the compiler with more detailed information about the runtime behavior of the program. A noteworthy feature enabled by GADTs is the concept of "Impossible Branches", which improves the usability of a language without compromising safety.

2.2.1 Impossible Branches

GADTs simplifies pattern matching by allowing programmers to safely omit branches for patterns that will never occur at runtime. Consider the following example:

```
let rec get_int (t : int term): int =
  match t with
    | T_Int n -> n
    | T_Add (x, y) -> (get_int x) + (get_int y)
  end
```

In the above function, the compiler knows *a priori* that an `int term` can only be built by the constructors `T_int` and `T_Add`. Consequently, it can tell if it is safe to elide branches for other constructors. This optimization gives the programmer a greater confidence of correctness in his code and also enhances its clarity by focusing only on relevant cases.

2.3 Coq

Coq is a dependently typed functional programming language, which in essence means that types can be parametrized over terms. In such a paradigm, types can be viewed as theorems, and a program that inhabits a type serves as a proof of that theorem.

Coq is both a programming language and a proof assistant, enabling verification of programs. Programming with dependent types can become complex and unwieldy, especially when proving theorems programmatically. To enhance the language's usability, the Coq system incorporates three distinct languages: Vernacular, Gallina, and Ltac.

Vernacular is the user-facing language, enabling direct interaction with the Coq system. It is used for adding various definitions to the context, such as functions (`Definition`), recursive functions (`Fixpoint`), theorems (`Theorem`), and lemmas (`Lemma`). It also provides commands to check the type of terms (`Check`), print the definition of terms (`Print`), introduce axioms (`Axiom`), and more.

Gallina is the core language of Coq, implementing the Calculus of Constructions. It underlies all Coq code, for example, `fun (x : Type) ⇒ x`, illustrating the language's functional and type-theoretical nature. Ltac is the language of *tactics*, or strategies for constructing

proofs. Ltac leverages the rich type context of Coq, allowing users to interactively build Gallina terms, making Ltac an essential tool for working within the Coq environment.

2.3.1 Inductive Types

Inductive types, introduced in Coq in the early 90s [26], introduced ADTs as a first class concept to the dependently typed framework of the Calculus of Constructions. This integration endows ADTs with both a computational and a logical interpretation. Logically, inductive types facilitate proving properties about data. Computationally, they allow defining functions via primitive recursion. The transition to the Calculus of Inductive Constructions (CIC) was made possible by Pfenning and Christine [27], who demonstrated that incorporating induction principles into CoC for inductively defined types could be done soundly.

Consider Coq's representation of Peano's natural numbers as an example:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

Following this definition, Coq automatically generates several induction and recursion principles, such as `nat_rect`, `nat_ind`, `nat_rec`, and `nat_sind`. Examining the `nat_ind` principle reveals a properly derived realizer for induction:

```
nat_ind =
fun (P : nat → Prop) (f : forall n : nat, P n → P (S n)) (f0 : P O) ⇒
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | S n0 ⇒ f n0 (F n0)
  | O ⇒ f0
  end
     : forall P : nat → Prop,
       (forall n : nat, P n → P (S n)) → P O → forall n : nat, P n
```

This principle exemplifies how Coq's type system, through inductive types, elegantly merges the realms of logic and computation, thereby enhancing the language's power for both programming and theorem proving.

Parameters vs Indices

Revisiting lists from Section 2.1, Coq's approach to lists illustrates the flexibility of dependent types:

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A
```

In this definition, the `list` datatype is parametrized over an arbitrary type `A`, with constructors that ensure that all the elements of this type are `list A`. The real intrigue begins when we explore dependent types' capability to have types depend on data. An illustrative example is the length-indexed list [19]:

```
Inductive ilist (A : Type) : nat → Type :=
| inil : ilist A O
| icons : forall (n : nat), A → ilist A n → ilist A (S n).

Arguments inil {_}.
Arguments icons {_ _} _ _.
```

Here, the list type carries information about its length. An `ilist` is parametrized on `A` and an indexed on it's length of type `nat`, distinguishing between parameters (before the colon) and indices (after the colon) in its type signature. You should note that indices vary (a la GADTs), parameters do not. The constructors `inil` and `icons` demonstrate the flexibility of indices similar to the capabilities showed for GADTs in Section 2.2: `inil` constructs an `ilist` of length zero (`ilist O`), while `icons` adds an element to an `ilist`, incrementing its length by one (`ilist (S n)`). The use of the `Arguments` command makes constructor arguments implicit when their types can be inferred, simplifying the syntax for creating lists.

An illustrative example of the power of indices is a type-safe implementation of a head function:

```
Fixpoint ihd {A} {n} (l : ilist A (S n)) : A :=
  match l with
  | icons a l' ⇒ a
  | inil ⇒ ... end.
```

This function can only be called on indexed lists that has at least one element. As a result, it can directly return an element without the need to employ a "maybe" type wrapper, which is typically used in programming to handle cases where a value might be absent. In order to implement the nonsensical `inil` branch, we will use dependent pattern matching. The rest of this chapter will slowly introduces the various details of dependent pattern matching; once all the pieces are in place, we will provide the complete implementation of `ihd`.

Dependent Pattern Matching

Consider the following append function for `ilists`:

```
Fixpoint append {A} {n1 n2: nat} (l1 : ilist A n1) (l2 : ilist A n2) : ilist A (n1 + n2) :=
match l1 with
| inil ⇒ l2
| icons h ls1 ⇒ icons h (append ls1 l2)
end.
```

More recent Coq versions will automatically annotate this code with the following elaborated version

```
Fixpoint append {A} {n1 n2: nat} (l1 : ilist A n1) (l2 : ilist A n2) : ilist A (n1 + n2) :=
match l1 in (ilist _ n1') return ilist A (n1' + n2) with
| inil ⇒ l2
| icons n h ls1 ⇒ icons (n + n2) h (append ls1 l2)
end.
```

Immediately, we notice that Coq's pattern matching uses the strange keywords `match ... in ... return ...`. This is due to the dependently typed nature of the language, and the need for fine grain directives in order for Coq to properly type check pattern matching. This is necessary because pattern matching in the presence of indices requires higher order unification, which is undecidable in general [28].

Ultimately, each pattern of the match produce a value of the expected type (in this case, the return type of `append`: `ilist A (n1 + n2)`). However, when `l1` is `inil` returning `l2` builds a list of type `ilist A n2` instead of the expected `ilist A (n1 + n2)`. In order to bridge this gap, we use the syntax `in (ilist _ n1')` to bind the indices of the discriminee, in this case `n1'`. Now

it is used in the `return` keyword to show how those indices are in the expected return type `return ilist A (n1' + n2)`. Finally, in the `inil` case, the type checker will unify `n1'` with the concrete value produced by the constructor declaration, and the final type expected for the `inil` branch will be `ilist A (0 + n2)`, and this trivially reduces to `ilist A n2` via β-reduction. A similar line of reasoning applies for type checking the `icons` case.

Finally, the type checker will ensure that the entire match has the desired type by unifying the type of the function `ilist A (n1 + n2)` with the return statement of the dependent pattern matching, `ilist A (n1' + n2)`. This unification will succeed, since originally `l1 : ilist A n1`, and therefore `n1'` concretized as `n1`, allowing Coq to accept that the return type of the match as expected type that the function returns.

Due to the homogeneous nature of type parameters, pattern matching should only bind occurrences of indices, and not parameters. Concretely, this means that inductive types parameters never changes with respect to each constructor. Therefore, in the extended pattern matching syntax, all parameters must be left as a wildcard `_`. In the above example, writing `match ... in ilist A n1' ...` would yield an error.

Convoy Pattern Rescues

Let's now revisit the impossible branch example from Section 2.2.1 in the context of Coq, showing how to handle cases that can be safely elided. This approach is illustrated through the example `get_nat` function designed to operate on a custom `term` type, which encapsulates natural numbers, boolean values, and addition operations:

```
Inductive term : Type → Type =
  | T_Int : nat → term nat
  | T_Bool : bool → term bool
  | T_Add : term nat → term nat → term int
```

The implementation of `get_nat` utilizes the refine keyword, enabling the function to temporarily leave holes (`_`) where the logic is incomplete or where type-checker feedback is desired:

```
Fixpoint get_nat (t: term nat) : nat.
    refine (match t with
```

```
| tNat tn ⇒ tn
| tAdd tn1 tn2 ⇒ get_nat tn1 + get_nat tn2
| tBool tb ⇒ _
end).
```

Just as before, we know that the case for `tBool` is nonsensical, because we know that `t` has type `term nat`, and `tBool` only builds terms of type `term bool`. Ideally, we should be able to use the information that `nat <> bool` to avoid writing this case.

Unfortunately, our typing context does not include the information that we are comparing `nat` and `bool`. In order to recover this information, we use a design pattern known as the *convoy pattern* [19].

```
Fixpoint get_nat (t: term nat) : nat.
  refine (match t in term a return a = nat → nat with
  | tNat tn ⇒ fun (eq : nat = nat) ⇒ tn
  | tAdd tn1 tn2 ⇒ fun (eq : nat = nat) ⇒ get_nat tn1 + get_nat tn2
  | tBool tb ⇒ fun (eq : bool = nat) ⇒ _
  end eq_refl).
```

With the convoy pattern we transform the shape of the branches of the pattern match to be a function, and this allows us to strengthen the context of each pattern with additional information. In particular, in the last pattern are able to add to our context an equality `eq : bool = nat` showing that in this branch, `nat` and `bool` are equals.

We also note that since the pattern match has a functional type `a = nat → nat`, the context for `get_nat` still expects just a `nat`, so we need to discharge the argument of this function. Since in this point of the context `t` has type `term nat`, we know that the pattern matching has type `nat = nat → nat`, and hence we can apply `eq_refl` to this pattern matching and discharge this argument.

To proceed defining this function, Coq now requires a proof that `bool` and `nat` cannot be equal, which follows by a simple cardinality argument.

Theorem 2.3.1. *$Bool \neq Nat$*

Proof. Assume by contradiction that $Bool = Nat$, therefore there exists a bijection $f : Bool \to Nat$.

By surjectivity of f, there exists b_1 and b_2 and b_3 such that

$$f(b_1) = 0 \tag{2.1}$$

$$f(b_2) = 1 \tag{2.2}$$

$$f(b_3) = 2 \tag{2.3}$$

Assume, without loss of generality that $b_1 = true$ and $b_2 = false$. If $b_3 = true$ then by injectivity $2 = 0$, this is a contradiction by the non-confusion properties of the constructors. If $b_3 = false$ then $2 = 1$, which is also a contradiction.

Therefore, there cannot exist a bijection between $Bool$ and Nat, and this suffices to show that $Bool \neq Nat$.

□

To use the above proof in our function we call this theorem `beq_neq_nat : bool = nat → False`.

Finally all the elements are present in order to build this impossible branch by the principle of explosion. Remember that the principle of explosion is realized in Coq by the function `False_rect` that has type `forall P : Type, False → P`, this can be literally read as "from false anything follows".

```
Fixpoint get_nat (t: term nat) : nat.
  refine (match t in term a return a = nat → nat with
  | tNat tn ⇒ fun _ ⇒ tn
  | tAdd tn1 tn2 ⇒ fun _ ⇒ get_nat tn1 + get_nat tn2
  | tBool tb ⇒ fun (eq : bool = nat) ⇒ False_rect nat (bool_neq_nat eq)
  end eq_refl).
```

Finally, all the pieces are in place to give a final implementation to `ihd` program from Section 2.3.1, given that `O_S` is a lemma of type `forall n, O <> S n`:

```
Fixpoint ihd {A} {n} (l : ilist A (S n)) : A :=
  match l in ilist _ n' return n' = S n → A with
  | icons a l' ⇒ fun _ ⇒ a
  | inil ⇒ fun eq ⇒ False_rect A (O_S n eq)
  end eq_refl.
```

More recently, Mathieau Sozeau has incorporated the Cockx algorithm [6] in the Equations plugin [29], greatly simplifying dependent pattern matching from the perspective of the user. This thesis was also greatly influenced by the work of Cockx, which gives a deep accounting into the essence of dependent pattern matching.

2.4 Coq-of-Ocaml

coq-of-ocaml [16] is a publicly available, open source translator from OCaml to Coq. The overriding design principle of **coq-of-ocaml** is the generation of an idiomatic and human readable shallow embedding of OCaml programs in Coq. The primary goal of this translation is to enable formal proofs on OCaml programs via their embeddings in Coq, but it can also be used to port existing OCaml libraries to Coq. The tool targets a subset of OCaml that covers most common programs while producing programs that are similar to the original. To this end, the selection of features is driven by specific use cases of the tool. The reasons for generating similar programs are twofold: firstly, this enables users to easily validate translated programs by comparing them to the original. Secondly, this results in code which is stable with respect to the changes made on the OCaml input, so that any Coq proofs about programs under active development can be more easily maintained.

3. TRANSLATING GADTml INTO gCIC

In this chapter we discuss the technical details of our translation.

3.1 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 `GSets` 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
  | 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.

3.2 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.

3.2.1 GADTml

GADTml is the source language of our compiler, and its syntax is defined in Section 3.2.1. 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 $\mathrm{e}[t]$ for type applications, uppercase lambdas are used for type abstractions, e.g. $\Lambda a.\mathrm{e}$, and $(\mathrm{e}_1, \mathrm{e}_2)$ 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. To differentiate between GADT type constructors and regular type constructors we use G for GADTs and T for regular ADTs.

The kinding and typing rules for GADTml are presented in Figures 3.2 and 3.3. 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.

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

Figure 3.1. GADTML Syntax

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

$$\frac{\Sigma; \Gamma, a \vdash s : *}{\Sigma; \Gamma \vdash \forall a.s : *}\ (\textsc{KAll}) \qquad \frac{\Sigma; \Gamma \vdash t_1 : * \qquad \Sigma; \Gamma \vdash t_2 : *}{\Sigma; \Gamma \vdash t_1 \to t_2 : *}\ (\textsc{KArr})$$

$$\frac{a \in \Gamma}{\Sigma; \Gamma \vdash a : *}\ (\textsc{KVar}) \qquad \frac{\Sigma; \Gamma \vdash t_1 : * \qquad \Sigma; \Gamma \vdash t_2 : *}{\Sigma; \Gamma \vdash t_1 * t_2 : *}\ (\textsc{KTup})$$

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

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

Figure 3.2. Kinding 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.

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 $\overline{v}$.

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

$$\frac{a \in \Gamma}{\Sigma; \Gamma \vdash a : *} \text{ (TyKVar)} \qquad \frac{\Sigma; \Gamma, x : t_1 \vdash u : t_2}{\Sigma; \Gamma \vdash \lambda x : t_1.\mathrm{e} : t_1 \to t_2} \text{ (TyAbs)}$$

$$\frac{\Gamma(x) = t}{\Sigma; \Gamma \vdash x : t} \text{ (TyVar)} \qquad \frac{\Sigma; \Gamma, a \vdash \mathrm{e} : t}{\Sigma; \Gamma \vdash \Lambda a.\mathrm{e} : \forall a.t} \text{ (TyTAbs)}$$

$$\frac{\Sigma; \Gamma \vdash \mathrm{e}_1 : t' \to t \qquad \Sigma; \Gamma \vdash \mathrm{e}_2 : t'}{\Sigma; \Gamma \vdash \mathrm{e}_1 \mathrm{e}_2 : t} \text{ (TyApp)}$$

$$\frac{\Sigma; \Gamma \vdash \mathrm{e} : \forall a.t \qquad \Sigma; \Gamma \vdash t' : *}{\Sigma; \Gamma \vdash \mathrm{e}[t'] : t[t'/a]} \text{ (TyTApp)}$$

$$\frac{\Sigma; \Gamma \vdash \mathrm{e}_1 : t_1 \qquad \Sigma; \Gamma \vdash \mathrm{e}_2 : t_2}{\Sigma; \Gamma \vdash (\mathrm{e}_1, \mathrm{e}_2) : t_1 * t_2} \text{ (TyTup)}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathrm{e} : T\ \overline{u} \qquad \Sigma; \Gamma \vdash t : * \\ \mathsf{type}\ T\ \overline{a}\ := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \to T\ \overline{a}} \in \Sigma \\ \{\ \Sigma; \Gamma, \overline{a, b, x_\mathrm{i} : t_\mathrm{i}} \vdash \mathrm{e}'_\mathrm{i} : t\ \}_{K_\mathrm{i}}\end{array}}{\Sigma; \Gamma \vdash \mathrm{match\ e\ with}\ \overline{\mid K_\mathrm{i}\ \overline{x_\mathrm{i}} \to \mathrm{e}'} : t} \text{ (TyMatch)}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathrm{e} : G\ \overline{u} \qquad \Sigma; \Gamma \vdash t : * \\ \mathsf{gadt}\ G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma \\ \left\{ \begin{array}{c} \Sigma; \sigma_\mathrm{i}(\Gamma, \overline{b, x_\mathrm{i} : t_\mathrm{i}}) \vdash \mathrm{e}'_\mathrm{i} : \sigma_\mathrm{i}(t) \\ \sigma_\mathrm{i} \equiv \mathsf{unifies}(\overline{u}, \overline{v_\mathrm{i}}) \not\equiv \bot \end{array} \right\}_{K_\mathrm{i}}\end{array}}{\Sigma; \Gamma \vdash \mathrm{match\ e\ with}\ \overline{\mid K_\mathrm{i}\ \overline{x_\mathrm{i}} \to \mathrm{e}'} : t} \text{ (TyGMatch)}$$

Figure 3.3. Typing Rules for GADTML

The typing rule TyMatch for case analysis on ADTs requires that there must be a well-typed branch for each one 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 [30] 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. Notice that unification of GADTs is undecidable [4], thus we present a best-effort algorithm in Figure 3.4.

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

Figure 3.4. Rules for unification on the Source Language

$$
\begin{array}{rcll}
\sigma() & \triangleq & () & \\
\sigma(a\Gamma) & \triangleq & \sigma\Gamma & \text{if } a \in dom(\sigma) \\
\sigma(a\Gamma) & \triangleq & a\sigma(\Gamma) & a \notin dom(\sigma) \\
\sigma((\text{e} : t)\Gamma) & \triangleq & (\text{e} : \sigma(t))\sigma\Gamma &
\end{array}
$$

Figure 3.5. Mapping Type Substitution on the Type Context of the Source Language

It is also necessary to define how to apply the type substitution to a context Γ, and this is shown in Figure 3.5. Context substitution is defined by induction over Γ by erasing each $a \in dom(\sigma)$ and applying the substitution σ to each $t \in codom(\Gamma)$.

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 $\sigma_\text{i}(t)$ in the context substituted with σ_i.

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

Figure 3.6. GCIC Syntax

3.2.2 gCIC

The target language of our translation is GCIC, a variant of CIC equipped with impredicative Set[3] and let bindings. We focus our attention to GCIC's treatment of inductive datatypes; interested readers can see [31] for a more detailed treatment of CIC.

Figure 3.6 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 [32], 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

[3] We use impredicative Set to simplify the translation by avoiding the vexing details of universes

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

$$\frac{(x : A) \in \Gamma}{\Sigma; \Gamma \vdash x : A} \qquad \frac{\Sigma; \Gamma \vdash f : \forall (x : a), B \qquad \Sigma; \Gamma \vdash a : A}{\Sigma; \Gamma \vdash f\,a : B[a/x]}$$

$$\frac{\Sigma; \Gamma, x : A \vdash t : B}{\Sigma; \Gamma \vdash \lambda x : A.t : \forall (x : A), B} \qquad \frac{\Sigma; \Gamma, x : A \vdash B : \mathit{Set} \qquad \Sigma; \Gamma \vdash A : \mathit{Set}}{\Sigma; \Gamma \vdash \forall (x : A), B : \mathit{Set}}$$

$$\frac{\Sigma; \Gamma \vdash \mathrm{e}' : t' \qquad \Sigma; \Gamma, x : t' \vdash \mathrm{e} : t}{\Sigma; \Gamma \vdash \mathrm{let}\ (x : t') = \mathrm{e}'\ \mathrm{in}\ \mathrm{e} : t}$$

$$\frac{\mathsf{Inductive}\ T\ \Xi : \Delta \to \mathrm{Set}\ := \overline{\mid K : \Delta \to T\ \overline{v}} \in \Sigma}{\Sigma; \Gamma \vdash K_\mathrm{i} : \Delta_\mathrm{i} \to T\ \overline{v_\mathrm{i}}} \quad (\textsc{CTyKons})$$

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

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathrm{e} : T\ \overline{u} \\ \Sigma; \Gamma, \overline{a} : \Delta \vdash t : s \\ \mathsf{Inductive}\ T\ \Xi : \Delta \to \mathrm{Set}\ := \overline{\mid K : \Delta \to T\ \overline{v}} \in \Sigma \\ \{\ \Sigma; \Gamma, \overline{x_\mathrm{i}} : \Delta_\mathrm{i} \vdash \mathrm{e}'_\mathrm{i} : t[\overline{u_\mathrm{i}}/\overline{a}]\ \}_{K_\mathrm{i}}\end{array}}{\Sigma; \Gamma \vdash \mathsf{match}\ \mathrm{e}\ \mathsf{in}\ T\ \overline{a}\ \mathsf{return}\ t\ \mathsf{with}\ \overline{\mid K\ \overline{x}\ \Rightarrow\ \mathrm{e}'}\ \mathsf{end} : t[\overline{u}/\overline{a}]} \quad (\textsc{CTyMatch})$$

Figure 3.7. Typing Rules for GCIC

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: $K_\mathrm{i} : \Delta_\mathrm{i} \to T\ \overline{v}$.

Figure 3.7 presents the typing rules for GCIC, which are largely standard. 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 $t[\overline{u_\mathrm{i}}/\overline{a}]$. Motives are required because unification is undecidable in the presence of inductive types [33].

We refer the reader to [31] for a detailed description of CIC.

3.3 The Translation

As discussed in Section 3.1, 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 enables, for example, the following GADTML program to be well-typed[4]:

```
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:

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

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.

3.3.1 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
```

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. Note that the definitions of GSet and its decoding function decodeG, were presented in Section 3.1. 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:

$$\begin{aligned} \xi_\Sigma \quad = \quad & [(\mathsf{T\_Lift}, \{a : \Delta\}); \\ & (\mathsf{T\_int}, \emptyset); \\ & (\mathsf{T\_bool}, \emptyset); \\ & (\mathsf{T\_pair}, \{(l : \Delta), (r : \Delta)\})] \end{aligned}$$

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

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

3.3.2 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 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.

3.3.3 Repair

The last phase results in the following well-typed program[5], 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
```

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

```
  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 $K_{inj} : h : K\ \overline{e_1} = K\ \overline{e_2} \to \overline{e_1 = e_2}$. Second, that they are *disjoint*, which we abbreviate as conflict : $K_i\ \overline{e_1} = K_j\ \overline{e_2} \to False$ (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.

3.3.4 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 [34, 35] $\Sigma; \Gamma \vdash t : * \rightsquigarrow t$, 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. 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. Analo-

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

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

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

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash t_1 : * \rightsquigarrow_g t_1 \mid \xi_1 \\ \Sigma; \Gamma \vdash t_2 : * \rightsquigarrow_g t_2 \mid \xi_2 \\ \xi = \xi_1 \sqcup \xi_2\end{array}}{\Sigma; \Gamma \vdash t_1 \to t_2 : * \rightsquigarrow_g t_1 \to t_2 \mid \xi} \quad (\textsc{TyTransArr})$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash t_1 : * \rightsquigarrow_g t_1 \mid \xi_1 \\ \Sigma; \Gamma \vdash t_2 : * \rightsquigarrow_g t_2 \mid \xi_2 \\ \xi = \xi_1 \sqcup \xi_2\end{array}}{\Sigma; \Gamma \vdash t_1 * t_2 : * \rightsquigarrow_g t_1 * t_2 \mid \xi} \quad (\textsc{TyTransTup})$$

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

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

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

Figure 3.8. Type Transpilation Rules

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

$$\frac{\Sigma; \Gamma \vdash x : t \qquad \Sigma; \Gamma \vdash t \rightsquigarrow_* t \mid \xi}{\Sigma; \Gamma \vdash x : t \rightsquigarrow x \mid \xi} \text{ (TransVar)}$$

$$\frac{\Sigma; \Gamma, a \vdash \mathrm{e} : t \rightsquigarrow \mathrm{e} \mid \xi}{\Sigma; \Gamma \vdash \Lambda a.\mathrm{e} : \forall a.t \rightsquigarrow \lambda(a : Set).\mathrm{e} \mid \xi} \text{ (TransKLam)}$$

$$\frac{\begin{array}{c}\xi = \xi_\mathrm{e} \sqcup \xi_t \\ \Sigma; \Gamma, x : t_1 \vdash \mathrm{e} : t \rightsquigarrow \mathrm{e} \mid \xi_\mathrm{e} \\ \Sigma; \Gamma \vdash t_1 : * \rightsquigarrow_* t_1 \mid \xi_t\end{array}}{\Sigma; \Gamma \vdash \lambda(x : t_1).\mathrm{e} : t_1 \to t \rightsquigarrow \lambda(x : t_1).\mathrm{e} \mid \xi} \text{ (TransLam)}$$

$$\frac{\begin{array}{c}\xi = \xi_1 \sqcup \xi_2 \\ \Sigma; \Gamma \vdash \mathrm{e}_1 : t_2 \to t_1 \rightsquigarrow \mathrm{e}_1 \mid \xi_1 \\ \Sigma; \Gamma \vdash \mathrm{e}_2 : t_2 \rightsquigarrow \mathrm{e}_2 \mid \xi_2\end{array}}{\Sigma; \Gamma \vdash \mathrm{e}_1\mathrm{e}_2 : t_1 \rightsquigarrow \mathrm{e}_1\mathrm{e}_2 \mid \xi} \text{ (TransApp)}$$

$$\frac{\begin{array}{c}\xi = \xi_1 \sqcup \xi_2 \\ \Sigma; \Gamma \vdash \mathrm{e} : \forall a.t \rightsquigarrow \mathrm{e} \mid \xi_1 \\ \Sigma; \Gamma \vdash s : * \rightsquigarrow_{\xi_1(a)} s \mid \xi_2\end{array}}{\Sigma; \Gamma \vdash \mathrm{e}[s] : t[s/a] \rightsquigarrow \mathrm{e}\ s \mid \xi} \text{ (TransTApp)}$$

$$\frac{\begin{array}{c}\mathsf{type}\ T\ \overline{a}\ := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \to T\ \overline{a}} \in \Sigma \\ \Sigma; \Gamma \vdash \mathrm{e} : T\ \overline{u} \rightsquigarrow \mathrm{e} \mid \xi_\mathrm{e} \\ \{\ \Sigma; \Gamma, \overline{a}, \overline{b}, \overline{x_\mathrm{i} : t_\mathrm{i}} \vdash \mathrm{e}'_\mathrm{i} : t \rightsquigarrow \mathrm{e}'_\mathrm{i} \mid \xi_\mathrm{i}\ \}_{K_\mathrm{i}} \\ \xi = (\bigsqcup \xi_\mathrm{i}) \sqcup \xi_\mathrm{e}\end{array}}{\begin{array}{l}\Sigma; \Gamma \vdash \mathsf{match}\ \mathrm{e}\ \mathsf{with}\ \overline{\mid K\ \overline{x} \to \mathrm{e}}\ \mathsf{end} : t \rightsquigarrow \\ \qquad \mathsf{match}\ \mathrm{e}\ \mathsf{with}\ \overline{\mid K\ \overline{x}\ \Rightarrow \mathrm{e}'}\ \mathsf{end} \mid \xi\end{array}} \text{ (TransMatch)}$$

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

Figure 3.9. Expression Transpilation

gously, when translating a GADT index, we mark $g = \Delta$, otherwise $g = *$. For example, if $\mathsf{gadt}\ G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma$, 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 3.8 lists the rules defining out the translation of types. $\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi$ means that well kinded type t under the context Γ is translated to a type t with the set of constraints ξ. The variable g can be seen as a flag that signals when the translation is happening in the universe of a GSet by setting $g = \Delta$, otherwise $g = *$. 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 $\xi = \bigsqcup \xi_i$. 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: $\forall a.Ta \to Ga$, where T is an ADT and G is a GADT the proper translation should be $\forall (a : GSet), T\ (\mathsf{decode}\ a) \to G\ a$, 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 $T\ (\mathsf{decode}\ a)$.

Expression Transpilation

Figure 3.9 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 $\mathrm{e}[s]$, 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 $\mathsf{match}\ \mathrm{e}\ \mathsf{with}\ \overline{\mid K\ \overline{x} \to \mathrm{e}_\mathrm{i}}\ \mathsf{end}$, 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 $\overline{a}, \overline{b}, \overline{x_\mathrm{i}}$. 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 $\overline{\mid K\ \overline{x} \to \mathrm{e}'}$ 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 $\mathsf{in}\ G\ \overline{c}\ \mathsf{return}\ (\overline{c = u}) \to\ t$, and having each branch take the equalities $\lambda(\overline{h : v = u}).\mathrm{e}_\mathrm{i}$ 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 $\mathsf{unifies}(\overline{u}, \overline{v_\mathrm{i}})$. 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.

Lemma 1 (Transpilation of expressions subsumes context of types)**.** *If* $\Sigma; \Gamma \vdash t : * \rightsquigarrow_g t \mid \xi_t$ *and* $\Sigma; \Gamma \vdash \mathrm{e} : t \rightsquigarrow \mathrm{e} \mid \xi_\mathrm{e}$ *then* $\xi_t \leq \xi_\mathrm{e}$

$$
\begin{array}{ll}
{}^{*}[Set]^{\Gamma}_{\xi} = Set & {}^{\Delta}[t_1 * t_2]^{\Gamma}_{\xi} = \mathsf{G\_tuple}\ {}^{\Delta}[t_1]^{\Gamma}_{\xi}\,{}^{\Delta}[t_2]^{\Gamma}_{\xi} \\
{}^{\Delta}[Set]^{\Gamma}_{\xi} = GSet & {}^{*}[\lambda(x : t_1), t]^{\Gamma}_{\xi} = \lambda(x :{}^{*}[t_1]^{\Gamma}_{\xi}),{}^{*}[t]^{\Gamma,(x\ :\ {}^{*}[t_1]^{\Gamma}_{\xi})}_{\xi} \\
{}^{*}[a]^{\Gamma}_{\xi} = \begin{cases} \mathsf{decode}\ a & \text{if } (a : \Delta) \in \xi \\ a & \text{otherwise} \end{cases} & {}^{*}[T\ \overline{u}]^{\Gamma}_{\xi} = T\ {}^{*}[\overline{u}]^{\Gamma}_{\xi} \\
{}^{\Delta}[a]^{\Gamma}_{\xi} = a, \quad \text{if } (a : \Delta) \in \xi & {}^{\Delta}[T\ \overline{u}]^{\Gamma}_{\xi} = \mathsf{G\_tconstr}\ (\#\Sigma(T))\ (T\ {}^{*}[\overline{u}]^{\Gamma}_{\xi}) \\
{}^{\Delta}[t_1 \to t_2]^{\Gamma}_{\xi} = \mathsf{G\_arrow}\ {}^{\Delta}[t_1]^{\Gamma}_{\xi}\,{}^{\Delta}[t_2]^{\Gamma}_{\xi} & {}^{*}[G\ \overline{u}]^{\Gamma}_{\xi} = G\ {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi} \\
{}^{*}[t_1 \to t_2]^{\Gamma}_{\xi} = {}^{*}[t_1]^{\Gamma}_{\xi} \to {}^{*}[t_2]^{\Gamma}_{\xi} & {}^{\Delta}[G\ \overline{u}]^{\Gamma}_{\xi} = \mathsf{G\_tconstr}\ (\#\Sigma(G))\ (G\ {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi}) \\
{}^{*}[t_1 * t_2]^{\Gamma}_{\xi} = {}^{*}[t_1]^{\Gamma}_{\xi} * {}^{*}[t_2]^{\Gamma}_{\xi} & {}^{*}[\overline{u} = \overline{v}]^{\Gamma}_{\xi} = {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi} = {}^{\Delta}[\overline{v}]^{\Gamma}_{\xi} \\
 & {}^{g}[\mathrm{e}_1\ \mathrm{e}_2]^{\Gamma}_{\xi} = {}^{g}[\mathrm{e}_1]^{\Gamma}_{\xi}\ {}^{g}[\mathrm{e}_2]^{\Gamma}_{\xi}
\end{array}
$$

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

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

Figure 3.10. Embedding Function

Proof. Straightforward induction over the structure of the translation of expressions, as $\xi_1 \leq \xi_1 \sqcup \xi_2$, for all ξ_1, ξ_2, and join is commutative. □

3.3.5 Embedding Phase

The embedding phase uses the embedding function ${}^{g}[-]^{\Gamma}_{\xi}$ that is defined in Figure 3.10. 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, $g = \Delta$ denotes that the embedding is being performed inside a GADT type constructor, and $g = *$ otherwise. In other words, g flips into Δ when

embedding the indices of a GADT type constructor, i.e. ${}^{*}[G\ \overline{u}]^{\Gamma}_{\xi} = G\ {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi}$ and flips back to $*$ when embedding indices of ADT type constructors, i.e. ${}^{\Delta}[T\ \overline{u}]^{\Gamma}_{\xi} = T\ {}^{*}[\overline{u}]^{\Gamma}_{\xi}$.

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, $T\ (\lambda(x : t).\mathrm{e})$ 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. $G\ {}^{\Delta}[\overline{u}]^{\Gamma}_{\xi}$. 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 $g = \Delta$ 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.

3.3.6 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 3.11. This function takes as input a term e, a target type t and a context Γ, and outputs a term $\mathrm{e}^{\dagger}$ that has type t under the context Γ (Lemma 2). 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.

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

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

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

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

$$\Gamma \vdash_s \lambda(x : t').\ \mathrm{e} : t' \to t \quad \triangleq \quad \Gamma, (x : t') \vdash_s \mathrm{e} : t$$

$$\Gamma, h : \tau = \tau \vdash_s \mathrm{e} : t \quad \triangleq \quad \Gamma \vdash_s \mathrm{e} : t$$

$$\Gamma \vdash_s \mathrm{e} : t \quad \triangleq \quad \mathrm{e} \quad \text{, if the head of } \Gamma \text{ is not an equation}$$

Figure 3.11. Repair Function

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:

$$\Sigma; \vdash \mathsf{eq\_rec} : \forall (A : Set)(x : A)(P : A \to Set),$$
$$Px \to \forall (y : A), x = y \to Py$$

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: $\lambda\ (y : A).\ (\overline{u} \to\ t)[x/y]$, which indicates that the expression being cast has type $\overline{u}[x/y] \to t[x/y]$. 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 is the inverse of substition. We can see at Lemma 2 by showing that the repair algorithm concretizes the type substitution of x for τ by generating the proper eq_rec term.

Lemma 2 (Repair step is the inverse of substitition)**.** *If* $\Sigma; \Gamma[\tau/x] \vdash \mathrm{e} : t[\tau/x]$ *and* $\Gamma, h : x = \tau \vdash_s \mathrm{e} : t = \mathrm{e}^{\dagger}$ *then* $\Sigma; \Gamma, h : x = \tau \vdash \mathrm{e}^{\dagger} : t$*, given that* Γ *has no equations.*

Proof. By applying the definition of $\vdash_s$ we have that

$$
\begin{aligned}
\mathrm{e}^{\dagger} = \ & \text{for } (\overline{z : u}) \in \Gamma \text{ s.t. } x \in u \\
& \mathsf{eq\_rec}\ A\ \tau(\lambda y.\ (\overline{u} \to t)[y/x]) \\
& (\lambda(\overline{z_0} : \overline{u}[\tau/x]).\ \Gamma[\overline{z_0/z}] - \{x\} \vdash_s \mathrm{e}[\overline{z_0/z}] : t[\tau/x]) \\
& x\ (\mathsf{eq\_sym}\ \mathrm{h})\ \overline{z}
\end{aligned}
$$

By the type signature of eq_rec, this application has the desired type t if all of it's arguments are properly typed. All arguments are straightforward to show that has the expected type, the only interesting one is the fourth with the recursive call to $\vdash_s$.

We want to show that

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

Since Γ has no equations, $\Gamma[\overline{z_0/z}] - \{x\} \vdash_s \mathrm{e}[\overline{z_0/z}] : t[\tau/x]) = \mathrm{e}[\overline{z_0/z}]$, after applying the typing rule for lambdas, it suffices to show that

$$\Sigma; \Gamma, (\overline{z_0} : \overline{u}[\tau/x]) \vdash \mathrm{e}[\overline{z_0/z}] : t[\tau/x]$$

Notice that since $\overline{z}$ are all the variables that mentions x in it's type, we are substituting all of those with a corresponding $z_0 : u[\tau/x]$, which has the same effect as typing e against $\Gamma[\tau/x]$. So it suffices to show that:

$$\Sigma; \Gamma[\tau/x] \vdash \mathrm{e} : t[\tau/x]$$

Which we have by assumption, and this finishes the proof.

□

Lemma 3 states that if we succeed to unify indices $\overline{u}, \overline{v}$ as σ, and we have a term e that is has type $\sigma(t)$, then we can repair the translation of e to be well-typed under the translation of t.

Lemma 3 (Repair Function is the Inverse of Unification)**.** *If all of the following holds:*

- $\sigma \equiv \mathit{unifies}(\overline{u},\ \overline{v}) \not\equiv \bot$;
- $\Sigma; \sigma(\Gamma) \vdash \mathrm{e} : \sigma(t) \rightsquigarrow \mathrm{e} \mid \xi_{\mathrm{e}}$
- $\Sigma; \Gamma \vdash t : * \rightsquigarrow_{*} t \mid \xi_t$
- $\Sigma; \Gamma \vdash \overline{u : *} \rightsquigarrow_{\Delta} \overline{u} \mid \xi_{\overline{u}}$
- $\Sigma; \Gamma \vdash \overline{v : *} \rightsquigarrow_{\Delta} \overline{v} \mid \xi_{\overline{v}}$
- $\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_{\Sigma}$
- $\Sigma; \vdash \Gamma \rightsquigarrow \Gamma$
- $\xi = \xi_{\mathrm{e}} \sqcup \xi_{\overline{u}} \sqcup \xi_{\overline{v}}$
- $[\Gamma]_{\xi}, \overline{\mathit{eq} : {}^{\Delta}[u]^{\Gamma}_{\xi} = {}^{\Delta}[v]^{\Gamma}_{\xi}} \vdash_{s} {}^{*}[\mathrm{e}]^{\Gamma}_{\xi} : {}^{*}[t]^{\Gamma}_{\xi} \equiv \mathrm{e}^{\dagger}$

then $[\Sigma]_{\xi_\Sigma}; [\Gamma]_{\xi}, \overline{\mathit{eq} : {}^{\Delta}[u]^{\Gamma}_{\xi} = {}^{\Delta}[v]^{\Gamma}_{\xi}} \vdash \mathrm{e}^{\dagger} : {}^{*}[t]^{\Gamma}_{\xi}$

Proof. By induction over the structure of the unification algorithm, using Lemma 2. □

We also prove a failing unification $\mathsf{unify}(\overline{u}, \overline{v}) \equiv \bot$ suffices for our algorithm to produce a proof using the explosion principle to any desired type.

Lemma 4 (Failing Unification Repairs to Any Type)**.** *If all of the following holds:*

- *if* $\overline{u}$ *and* $\overline{v}$ *doesn't have type constructors*
- $\mathit{unify}(\overline{u}, \overline{v}) \equiv \bot$
- $\Sigma; \Gamma \vdash \overline{u} \rightsquigarrow_{\Delta} \overline{u} \mid \xi_u$
- $\Sigma; \Gamma \vdash \overline{v} \rightsquigarrow_{\Delta} \overline{v} \mid \xi_v$
- $\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_{\Sigma}$
- $\Sigma; \vdash \Gamma \rightsquigarrow \Gamma$

- $\xi = \xi_u \sqcup \xi_v$
- $[\Gamma]_\xi, \overline{\mathit{eq} : {}^{\Delta}[u]^{\Gamma}_{\xi} = {}^{\Delta}[v]^{\Gamma}_{\xi}} \vdash_s \mathit{False} : t \equiv \mathrm{e}^\dagger$

then $[\Sigma]_{\xi_\Sigma}; [\Gamma]_\xi, \overline{\mathit{eq} : {}^{\Delta}[u]^{\Gamma}_{\xi} = {}^{\Delta}[v]^{\Gamma}_{\xi}} \vdash \mathrm{e}^\dagger : t$

Proof. Straightforward induction over the structure of the unification $\mathsf{unify}(\overline{u}, \overline{v})$. Notice that $\overline{u}$ and $\overline{v}$ can't mention type constructors otherwise the repair function would get stuck trying to compare $T\ \overline{u'} = T\ \overline{v'}$, since type constructors aren't injective, in general, in GCIC, and unfortunately GSet is not expressive enough to deeply embed these sort of type constructors. □

3.3.7 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 3.3.1 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 $\Sigma \vdash \Gamma \rightsquigarrow \Gamma$, they are defined at Figure 3.13.

$\boxed{\Sigma; \Delta \vdash \Gamma \rightsquigarrow \Gamma}$

$$\frac{}{\Sigma; \Delta \vdash () \rightsquigarrow ()} \quad (\textsc{ctrans\_\_empty})$$

$$\frac{\Sigma; \Delta, a \vdash \Gamma \rightsquigarrow \Gamma}{\Sigma; \Delta \vdash a\Gamma \rightsquigarrow (a : \mathit{Set})\Gamma} \quad (\textsc{ctrans\_\_kind})$$

$$\frac{\Sigma; \Delta, (x : t) \vdash \Gamma \rightsquigarrow \Gamma \qquad \Sigma; \Delta \vdash t : * \rightsquigarrow_* t \mid \_}{\Sigma; \Delta \vdash (x : t)\Gamma \rightsquigarrow (x : t)\Gamma} \quad (\textsc{ctrans\_\_type})$$

Figure 3.12. Typing Context Translation

$\boxed{\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma}$

$$\frac{}{\vdash () \rightsquigarrow ()} \quad (\textsc{sigtrans\_\_empty})$$

$$\frac{\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma \qquad \mathsf{type}\ T\ \overline{a}\ := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \to T\ \overline{a}} \in \Sigma \qquad \left\{ \begin{array}{c} \Sigma; \vdash \forall \overline{ab_{\mathrm{i}}}.\overline{t_{\mathrm{i}} : *} \rightsquigarrow_* \Delta_{\mathrm{i}} \mid \xi_t \\ \xi_\Sigma(T_{K_{\mathrm{i}}}) = \xi_t \end{array} \right\}_{K_{\mathrm{i}}}}{\begin{array}{c} \vdash \Sigma; \mathsf{type}\ T\ \overline{a}\ := \overline{\mid K : \forall \overline{ab}.\ \overline{t} \to T\ \overline{a}} \rightsquigarrow \\ \Sigma; \mathsf{Inductive}\ T\ \overline{(a : Set)} :\ \mathrm{Set}\ := \overline{\mid K : \Delta \to T\ \overline{a}} \mid \xi_\Sigma \end{array}} \quad (\textsc{sigtrans\_\_adt})$$

$$\frac{\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma \qquad \mathsf{gadt}\ G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma \qquad \left\{ \begin{array}{c} \Sigma; \vdash \forall \overline{b_{\mathrm{i}}}.\overline{t_{\mathrm{i}} : *} \rightsquigarrow_* \Delta_{\mathrm{i}} \mid \xi_t \\ \Sigma; \overline{b_{\mathrm{i}}} \vdash \overline{v_{\mathrm{i}} : *} \rightsquigarrow_\Delta \overline{v_{\mathrm{i}}} \mid \xi_{\mathrm{i}} \\ \xi_\Sigma(G_{K_{\mathrm{i}}}) = \xi_t \sqcup \xi_{\mathrm{i}} \end{array} \right\}_{K_{\mathrm{i}}}}{\begin{array}{c} \vdash \Sigma; \mathsf{gadt}\ G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \rightsquigarrow \\ \Sigma; \mathsf{Inductive}\ G : \forall \overline{(a : GSet)},\ \mathrm{Set}\ := \overline{\mid K : \Delta \to G\ \overline{v}} \mid \xi_\Sigma \end{array}} \quad (\textsc{sigtrans\_\_gadt})$$

Figure 3.13. Datatype Declaration Context Translation

In order to translate type contexts Γ we have three rules:

1. CTRANS_EMPTY translates empty contexts into empty contexts;
2. CTRANS_KIND translates type variables to variables of type Set;
3. and CTRANS_TYPE translates a variable x of type t into a variable x of type t, given that t is a translation of t.

The translation of Datatype Signature Σ introduces a particular GSet constraint $\xi_\Sigma(T_{K_i})$ for each constructor of each datatype, since their variables are local. This translation is also performed by the three rules declared in Figure 3.13, and they are:

1. SIGTRANS_EMPTY translates empty signature contexts into empty signature contexts;
2. SIGTRANS_ADT If the first declaration is an ADT, then it translates parameter of its constructors $\forall\overline{ab}.\overline{t_i}$ into Δ_i and sets the ξ_Σ related with that constructor to the GSET-Context generated by the constructor parameters translation ξ_t. Finally, the parameters $\overline{a}$ are set as the parameters of the Inductive Definition $\overline{(a : Set)}$;
3. SIGTRANS_GADT behaves similarly, but it also needs to translate the indices related to the return type of each constructor v_i into v_i, of which the generated GSet constraint ξ_i will be joined with the GSet constraint of the constructor parameters ξ_t. This time, the parameters $\overline{a}$ are translated as indices of the Inductive Definition $\forall\overline{(a : GSet)}$;

Notice that for both variable contexts translations and declaration contexts translations, all names are maintained, be it the datatype name, variable name or constructor name.

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 3.3.1 (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 3.3.2 (Expression Translation Preserves Typing). *If* $\Sigma; \Gamma \vdash \mathrm{e} : t \rightsquigarrow \mathrm{e} \mid \xi$ *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* $[\Sigma]_{\xi_\Sigma}; [\Gamma]_\xi \vdash {}^*[\mathrm{e}]^\Gamma_\xi : {}^*[t]^\Gamma_\xi$, *assuming that* e *doesn't have pattern matchings over datatypes that uses user-defined types as indices*

Proof. By induction over the expression transpilation derivation $\Sigma; \Gamma \vdash \mathrm{e} : t \rightsquigarrow \mathrm{e} \mid \xi$. We show the case for pattern matching.

Case

$$\Sigma; \Gamma \vdash \begin{array}{l}\mathsf{match}\ \mathrm{e}\ \mathsf{with} \\ \overline{\mid K\overline{x} \to \mathrm{e} : t}\end{array} \rightsquigarrow \left|\begin{array}{l}\mathsf{match}\ \mathrm{e}\ \mathsf{in}\ G\ \overline{b}\ \mathsf{return}\ (\overline{b = u} \to t) \\ \overline{\mid K\overline{x} \Rightarrow \Gamma, (\overline{x} : \Delta) \vdash_s \lambda(\overline{h : v = u}).\ \mathrm{e}'_\mathrm{i}} \\ \mathsf{end}\ \overline{\mathsf{eq\_refl}}\end{array}\right| \xi$$

We have that

1. $\vdash \Sigma \rightsquigarrow \Sigma \mid \xi_\Sigma$
2. $\Sigma \vdash \Gamma \rightsquigarrow \Gamma$
3. $\Sigma; \Gamma \vdash t : * \rightsquigarrow_* t \mid \_$
4. $\Sigma; \Gamma \vdash \mathrm{e} : G\ \overline{u} \rightsquigarrow \mathrm{e} \mid \xi_\mathrm{e}$
5. $\Sigma; \Gamma \vdash G\ \overline{u} : * \rightsquigarrow_* G\ \overline{u} \mid \xi_G$

   $\Sigma; \Gamma \vdash u_\mathrm{i} : * \rightsquigarrow_\Delta u_\mathrm{i} \mid \xi_{u_\mathrm{i}}$, for each $u \in \overline{u}$

$$\xi_G = \bigsqcup \xi_{u_i}$$

6. gadt $G\ \overline{a}\ := \overline{\mid K : \forall \overline{b}.\ \overline{t} \to G\ \overline{v}} \in \Sigma$

7. $\xi = \xi_{\Sigma(G)} \sqcup \xi_G \sqcup \xi_e$

We want to show that

$$\Sigma; \Gamma \ \vdash \ \begin{array}{l} \mathsf{match}\ {}^*[\mathrm{e}]^\Gamma_\xi\ \mathsf{in}\ G\ \overline{b}\ \mathsf{return}\ {}^*[(\overline{b = u} \to t)]^\Gamma_\xi\ \mathsf{with} \\ \overline{\mid K\overline{x} \Rightarrow {}^*[\Gamma, (\overline{x} : \Delta) \vdash_s \lambda(\overline{h : v = u}).\ \mathrm{e}'_i]^{\Gamma,(\overline{x}\ :\ \Delta)}_\xi} \\ \mathsf{end}\ \overline{\mathsf{eq\_refl}} \end{array} \ : \ {}^*[t]^\Gamma_\xi$$

By applying the match typing rule we need to show that:

1. $\Sigma; [\Gamma]_\xi \vdash {}^*[\mathrm{e}]^\Gamma_\xi : G\ {}^\Delta[\overline{u}]^\Gamma_\xi$

2. $\Sigma; \Gamma \vdash {}^*[\overline{v = u} \to\ t]^\Gamma_\xi : \mathsf{Set}$

3. $\mathsf{Inductive}\ G : \Delta \to \mathsf{Set}\ := \overline{\mid K : \Delta \to G\ \overline{v}} \in \Sigma$

4. $\Sigma; {}^*[\Gamma]^\Gamma_\xi, \overline{x_i} : {}^*[\Delta_i]^\Gamma_\xi, \overline{h : {}^\Delta[v]^\Gamma_\xi = {}^\Delta[u]^\Gamma_\xi} \vdash \mathrm{e}^\dagger_i : {}^*[t]^\Gamma_\xi$, for each K_i

All of these are straightforward. The first one follows directly from the induction hypothesis generated by hypothesis 4. The second one is trivial from the kinding rules, using Theorem 3.3.1 (Type Translation Preserves Kinding). The third one is straightforward, noting that translation of declaration signatures is deterministic. The interesting part is the fourth one.

For each K_i there are two cases, either $unifies(\overline{u}, \overline{v})$ returns a valid substitution σ_i or it fails. We proceed the proof analyzing each case of the translation:

Case $unifies(\overline{v}, \overline{u}) = \bot$, therefore $\mathrm{e}^\dagger_i = [\Sigma]_{\xi_\Sigma}; [\Gamma]_\xi, \overline{x_i} : {}^*[\Delta_i]^\Gamma_\xi, \overline{h : {}^\Delta[v]^\Gamma_\xi = {}^\Delta[u]^\Gamma_\xi} \vdash_s \mathsf{False} : {}^*[t]^\Gamma_\xi$. This follows directly from the correctness of disunification Lemma 4. Note that we can apply this lemma because we have as hypothesis that we do not pattern match over datatypes that uses type families as indices, therefore $\overline{v}, \overline{u}$ cannot mention any type constructors.

Case $\sigma_i = unifies(\overline{v}, \overline{u}) \neq \bot$, therefore $\Sigma; \sigma_i(\Gamma, \overline{x_i} : \Delta_i) \vdash \mathrm{e}'_i : \sigma_i(t) \rightsquigarrow \mathrm{e}'_i$, then by induction hypothesis

$$\forall \Gamma', t_0. \Sigma; \vdash \sigma_{\mathrm{i}}(\Gamma, \overline{x_{\mathrm{i}}} : \Delta_{\mathrm{i}}) \rightsquigarrow \Gamma' \ \rightarrow$$
$$\Sigma; \Gamma, \overline{x_{\mathrm{i}}} : \sigma_{\mathrm{i}}(\Delta_{\mathrm{i}}) \vdash \sigma_{\mathrm{i}}(t) : * \rightsquigarrow t_0 \ \rightarrow$$
$$\Sigma; \Gamma' \vdash \mathrm{e}'_{\mathrm{i}} : t_0$$

Therefore we have that $\Sigma; \sigma_{\mathrm{i}}(\Gamma, \overline{x_{\mathrm{i}}} : \Delta_{\mathrm{i}}) \vdash \mathrm{e}'_{\mathrm{i}} : \sigma_{\mathrm{i}}(t)$, where σ_{i} is the direct translation of σ_{i}. Then by generalizing Lemma 2 we conclude that

$$\Sigma; \Gamma, \overline{x_{\mathrm{i}}} : \Delta_{\mathrm{i}}, \overline{h : v = u} \vdash \mathrm{e}^{\dagger}_{\mathrm{i}} : t$$

as we wanted. □

4. 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 3.3, 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**. We provide these benchmarks in the github repo branch that accompanies this thesis[6]. In particular, the micro-benchmark related to GADTs can be found in the following files:

- *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;
- *GSet_record.ml*: regular records and irrefutable patterns;

[6]↑https://github.com/pedrotst/coq-of-ocaml/tree/thesis/tests

Table 4.1. 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**

- *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 4.1. 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.

5. RELATED WORK

The source language of our translation, GADTML, is similar to $\lambda_{2,G\mu}$, first presented in [25]. 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, [36] 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 [36] does not need to construct proof terms witnessing type casts and the infeasibility of impossible branches.

Finally, developed independently and concurrently with our work, [7] appears to offer a definite resolution for the metatheory of GADTs. They propose the calculus $F^{=\text{ii}}_{\omega\mu}$, an extension of F_ω with higher-kinded iso-recursive types that satisfies injectivity and discriminability laws of GADTs. They prove type soundness with a novel normalization-by-evaluation technique. They also prove that GADTs adds expressive power to programming languages, when compared to regular ADTs. In contrast to GADTML, $F^{=\text{ii}}_{\omega\mu}$ uses internalized type equality in their calculus and provides a rigorous account for all the interesting properties that our work could was unable to complete.

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** [17], 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 [37]. One key implementation challenge for **hs-to-coq** not present in **coq-of-ocaml** was the difference in evaluation strategies between Haskell, which uses lazy evaluation, and Coq, which is strict. In practice, this is not a problem, as the tool targets a total subset of Haskell. Instead of turning off Coq's termination checker, **hs-to-coq** uses the axiom `unsafeFix : forall {A}, (A → A) → A` to encode functions that are not obviously terminating. While also unsound, this approach is more local than disabling the termination checker, as the axiom is only used in potentially non-terminating functions. **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.

Another source-to-source translater to Coq is **coq-of-rust**. However, since Rust does not support GADTs, and [7] proves that GADTs add expressive power to a language, then **coq-of-rust** shall not face the same translation issues related to GADTs.

CFML translates OCaml programs to Coq via characteristic formulae [38]. 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**) is a tool to validate translations between OCaml and PVS [39] 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 [40, 41] takes as input OCaml programs annotated with specifications in the GOSPEL language and outputs verification conditions. These conditions are then discharged by Why3 [42], 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 [43] to OCaml, Haskell, Scheme and JSON. This is, in some sense, the inverse of the problem we address here, as we go 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. [17] 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.

6. CONCLUSION

6.1 Future Work

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. This new universe should be sound, as long as it doesn't assume the excluded middle since type constructors are known to be unsound in its presence [44].

6.2 Conclusion

In this thesis, 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] J. Garrigue and J. L. Normand, "Adding GADTs to OCaml the direct approach," p. 29, 2011.

[2] J. Stolarek, S. Peyton Jones, and R. A. Eisenberg, "Injective type families for haskell," in *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, ser. Haskell '15, Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 118–128, ISBN: 9781450338080. DOI: 10.1145/2804302.2804314. [Online]. Available: https://doi.org/10.1145/2804302.2804314.

[3] G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones, "Gadts meet their match: Pattern-matching warnings that account for gadts, guards, and laziness," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015, Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 424–436, ISBN: 9781450336697. DOI: 10.1145/2784731.2784748. [Online]. Available: https://doi.org/10.1145/2784731.2784748.

[4] J. Garrigue and J. Le Normand, "GADTs and Exhaustiveness: Looking for the Impossible," en, *Electronic Proceedings in Theoretical Computer Science*, vol. 241, pp. 23–35, Feb. 2017, ISSN: 2075-2180. DOI: 10.4204/EPTCS.241.2. [Online]. Available: http://arxiv.org/abs/1702.02281.

[5] J. Cockx, "Dependent pattern matching and proof-relevant unification," Ph.D. dissertation.

[6] J. Cockx and A. Abel, "Elaborating dependent (co)pattern matching: No pattern left behind," en, *Journal of Functional Programming*, vol. 30, e2, 2020, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796819000182. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0956796819000182/type/journal_article.

[7] F. Sieczkowski, S. Stepanenko, J. Sterling, and L. Birkedal, "The essence of generalized algebraic data types," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, Jan. 2024, ISSN: 2075-2180. DOI: 10.1145/3632866. [Online]. Available: https://doi.org/10.1145/3632866.

[8] D. Patterson and A. Ahmed, "The next 700 compiler correctness theorems (functional pearl)," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jun. 2019. DOI: 10.1145/3341689. [Online]. Available: https://doi.org/10.1145/3341689.

[9] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jun. 2009, ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. [Online]. Available: https://doi.org/10.1145/1538788.1538814.

[10] G. Klein *et al.*, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220, ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. [Online]. Available: https://doi.org/10.1145/1629575.1629596.

[11] R. Gu *et al.*, "Deep specifications and certified abstraction layers," *SIGPLAN Not.*, vol. 50, no. 1, pp. 595–608, Jan. 2015, ISSN: 0362-1340. DOI: 10.1145/2775051.2676975. [Online]. Available: https://doi.org/10.1145/2775051.2676975.

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the fscq file system," in *Proceedings of the 25th Symposium on Operating Systems Principles*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 18–37, ISBN: 9781450338349. [Online]. Available: https://doi.org/10.1145/2815400.2815402.

[13] K. Bhargavan *et al.*, *Implementing and proving the tls 1.3 record layer*, Cryptology ePrint Archive, Report 2016/1178, https://ia.cr/2016/1178, 2016.

[14] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. V. Tassel, "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*, NLD: North-Holland Publishing Co., 1992, pp. 129–156, ISBN: 0444896864.

[15] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. Appel, "Vst-floyd: A separation logic tool to verify correctness of c programs," *Journal of Automated Reasoning*, vol. 61, pp. 367–422, 2018.

[16] G. Claret, *Coq of ocaml*, https://github.com/clarus/coq-of-ocaml, Accessed: 2021-09-09, 2021.

[17] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, "Total haskell is reasonable coq," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018, Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 14–27, ISBN: 9781450355865. DOI: 10.1145/3167092. [Online]. Available: https://doi.org/10.1145/3167092.

[18] T. C. D. Team, *The coq proof assistant*, version 8.13, Jan. 2021. DOI: 10.5281/zenodo.4501022. [Online]. Available: https://doi.org/10.5281/zenodo.4501022.

[19] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. -: The MIT Press, 2013, ISBN: 0262026651.

[20] N. Tabareau, É. Tanter, and M. Sozeau, "Equivalences for free: Univalent parametricity for effective transport," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jun. 2018. DOI: 10.1145/3236787. [Online]. Available: https://doi.org/10.1145/3236787.

[21] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[22] A. Chlipala, "Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl)," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473599. [Online]. Available: https://doi.org/10.1145/3473599.

[23] G. Gilbert, J. Cockx, M. Sozeau, and N. Tabareau, "Definitional proof-irrelevance without k," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. DOI: 10.1145/3290316. [Online]. Available: https://doi.org/10.1145/3290316.

[24] R. M. Burstall, D. B. MacQueen, and D. T. Sannella, "Hope: An experimental applicative language," in *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, ser. LFP '80, Stanford University, California, USA: Association for Computing Machinery, 1980, pp. 136–143, ISBN: 9781450373968. DOI: 10.1145/800087.802799. [Online]. Available: https://doi.org/10.1145/800087.802799.

[25] H. Xi, C. Chen, and G. Chen, "Guarded recursive datatype constructors," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '03, New Orleans, Louisiana, USA: Association for Computing Machinery, 2003, pp. 224–235, ISBN: 1581136285. DOI: 10.1145/604131.604150. [Online]. Available: https://doi.org/10.1145/604131.604150.

[26] C. Paulin-Mohring, "Inductive definitions in the system coq rules and properties," in *International Conference on Typed Lambda Calculi and Applications*, Springer, 1993, pp. 328–345.

[27] F. Pfenning and C. Paulin-Mohring, "Inductively defined types in the calculus of constructions," in *Mathematical Foundations of Programming Semantics*, M. Main, A. Melton, M. Mislove, and D. Schmidt, Eds., red. by G. Goos *et al.*, vol. 442, Series Title: Lecture Notes in Computer Science, New York, NY: Springer-Verlag, 1990, pp. 209–228. DOI: 10.1007/BFb0040259. [Online]. Available: https://link.springer.com/10.1007/BFb0040259.

[28] G. P. Huet, "The undecidability of unification in third order logic," *Information and Control*, vol. 22, no. 3, pp. 257–267, 1973, ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(73)90301-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001999587390301X.

[29] M. Sozeau, "Equations: A dependent pattern-matching compiler," in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 419–434, ISBN: 978-3-642-14052-5.

[30] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[31] C. Paulin-Mohring, *Introduction to the calculus of inductive constructions*, 2015.

[32] J. Cockx, D. Devriese, and F. Piessens, "Unifiers as equivalences: Proof-relevant unification of dependently typed data," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2016, Nara, Japan: Association for Computing Machinery, 2016, pp. 270–283, ISBN: 9781450342193. DOI: 10.1145/2951913.2951917. [Online]. Available: https://doi.org/10.1145/2951913.2951917.

[33] C. McBride, "Dependently typed functional programs and their proofs," 2000.

[34] W. J. Bowman, Y. Cong, N. Rioux, and A. Ahmed, "Type-preserving cps translation of Σ and Π types is not not possible," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158110. [Online]. Available: https://doi.org/10.1145/3158110.

[35] A. Ahmed and M. Blume, "Typed closure conversion preserves observational equivalence," *SIGPLAN Not.*, vol. 43, no. 9, pp. 157–168, Sep. 2008, ISSN: 0362-1340. DOI: 10.1145/1411203.1411227. [Online]. Available: https://doi.org/10.1145/1411203.1411227.

[36] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly, "System f with type equality coercions," in *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, ser. TLDI '07, Nice, Nice, France: Association for Computing Machinery, 2007, pp. 53–66, ISBN: 159593393X. DOI: 10.1145/1190315.1190324. [Online]. Available: https://doi.org/10.1145/1190315.1190324.

[37] J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, and S. Weirich, "Ready, set, verify! applying hs-to-coq to real-world haskell code (experience report)," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018. DOI: 10.1145/3236784. [Online]. Available: https://doi.org/10.1145/3236784.

[38] A. Charguéraud, "Program verification through characteristic formulae," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '10, Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 321–332, ISBN: 9781605587943. DOI: 10.1145/1863543.1863590. [Online]. Available: https://doi.org/10.1145/1863543.1863590.

[39] S. Owre, J. M. Rushby, and N. Shankar, "Pvs: A prototype verification system," in *International Conference on Automated Deduction*, Springer, 1992, pp. 748–752.

[40] M. Pereira and A. Ravara, "Cameleer: A deductive verification tool for ocaml," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., Cham: Springer International Publishing, 2021, pp. 677–689, ISBN: 978-3-030-81688-9.

[41] J.-C. Filliâtre *et al.*, "A toolchain to produce verified ocaml libraries," Jan. 2020, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01783851.

[42] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128, ISBN: 978-3-642-37036-6.

[43] P. Letouzey, "Extraction in coq: An overview," in *Conference on Computability in Europe*, Springer, 2008, pp. 359–369.

[44] L. de Moura, *Coq of ocaml*, https://github.com/leanprover/lean/issues/654, Accessed: 2022-09-21, 2022.